

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД
„ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА”

Навчально-науковий інститут математики та інформаційних технологій

Кафедра математики та інформатики

Ся Юншен

ДОСЛІДЖЕННЯ ІСНУЮЧИХ АЛГОРИТМІВ ПРОЦЕДУРНОЇ
ГЕНЕРАЦІЇ КОНТЕНТУ В ІГРОВИХ ДОДАТКАХ

Магістерська робота
за спеціальністю 122 "Комп'ютерні науки"

Особистий підпис – _____

Науковий керівник – _____ к.т.н., доцент Галина КОЗУБ

В.о. зав. кафедри – _____ д.т.н., професор Юрій КОЗУБ

Полтава – 2025

АНОТАЦІЯ

Ся Юншен

Тема: Дослідження існуючих алгоритмів процедурної генерації контенту в ігрових додатках

Спеціальність: 122 „Комп’ютерні науки”

Установа: ДЗ ЛНУ імені Тараса Шевченка, 2025р.

Кваліфікаційна робота містить: 72стор., 4 табл., 25 рис., 32 джерела, 2 додатки.

Об’єкт дослідження – процес програмної реалізації генерації контенту в ігровому додатку.

Предмет дослідження – технології створення ігрових додатків.

Мета роботи – дослідження і програмна реалізація генерації лабіринту в ігровому додатку з використанням сучасних засобів розробки і методів процедурної генерації контенту.

Результати роботи. Досліджено класифікації комп’ютерних ігор за ігровими платформами, за жанром, за змістом, а також за кількістю гравців та за видавничими критеріями. Розглянуто ключові етапи створення гри. Досліджено алгоритми створення лабіринтів, запропоновано алгоритми поєднання генерації таких елементів з генерацією самого лабіринту. Обрано модифікацію алгоритму Вілсона та визначено функціональні вимоги до відеогри «Розкрадач гробниць».

Виконано програмну реалізацію генерації лабіринту у середовищі Unity 3D користуючись інтегрованим середовищем розробки Visual Studio та мовою програмування C#.

Ключові слова: Unity 3D, C#, алгоритм Вілсона, Visual Studio, лабіринт.

ABSTRACT

Xia Yongsheng

Theme: Research into existing algorithms for procedural content generation in gaming applications

Specialty: 122 "Computer Science"

Institution: Taras Shevchenko National University of Luhansk, 2020

Qualification work contains: 72 pages, 25 figures, 32 sources, 2 appendices.

Object of research is the process of software implementation of content generation in a game application

Subject of research is the technology of creating game applications.

Purpose of the work is the study and software implementation of maze generation in a game application using modern development tools and methods of procedural content generation.

Results of the work. The classification of computer games by game platforms, by genre, by content, as well as by the number of players and by publishing criteria is studied. The key stages of game creation are considered. Algorithms for creating labyrinths have been studied, algorithms for combining the generation of such elements with the generation of the labyrinth itself have been proposed. A modification of Wilson's algorithm has been selected and functional requirements for the video game "Tomb Raider" have been determined.

A software implementation of the labyrinth generation has been performed in the Unity 3D environment using the Visual Studio integrated development environment and the C# programming language.

Keywords: Unity 3D, C#, Wilson's algorithm, Visual Studio, maze.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ	5
ВСТУП.....	6
РОЗДІЛ 1 АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ	8
1.1 Особливості процесу розробки ігрових додатків	8
1.2 Огляд та класифікація ігрових додатків	12
1.2.1 Опис класифікації ігор за ігровими платформами	12
1.2.2 Огляд класифікації комп'ютерних ігор за змістом.....	13
1.2.3 Опис класифікації ігор за видавничими критеріями	15
1.2.4 Опис класифікації ігор за кількістю гравців	16
1.3 Висновки до першого розділу	17
РОЗДІЛ 2 МЕТОДИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ КОНТЕНТУ В ІГРОВИХ ДОДАТКАХ.....	18
2.1 Класифікація та типізація лабіринтів.....	18
2.2 Алгоритми процедурної генерації контенту в ігрових додатках	23
2.3 Аналіз алгоритмів генерації лабіринтів.....	30
2.4 Розробка модифікованого методу генерації лабіринтів.....	33
2.5 Висновки до другого розділу	37
РОЗДІЛ 3 РОЗРОБКА ІГРОВОГО ДОДАТКУ	38
3.1 Інструментальні засоби реалізації проєкту	38
3.2 Вихідні дані проєкту	39
3.3 Архітектура програмного додатку	40
3.4 Генерування даних лабіринту.....	44
3.5 Генерація мешу лабіринту.....	48
3.6 Інтерактивні аспекти гри.....	55
3.7 Висновки до третього розділу	62
ВИСНОВКИ.....	63
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	64
ДОДАТКИ.....	67
Додаток А Код GameController.cs.....	67
Додаток В. Код MazeMeshGenerator.cs	69

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ

AI	– artificial intelligence;
BFS	– Breadth First Search;
BSP	– Binary Space Partitioning;
CBT	– Closed Beta Testing;
CPI	– cost per install;
DFS	– Depth First Search;
DLC	– набір ігрових елементів;
GUI	– graphical user interface;
HDFS	– Heuristic Depth First Search;
LTV	– lifetime value;
MMO	– Massively multiplayer online game;
OBT	– Open beta testing;
PBEM	– Play By Electronic Mail;
RW	– Random Walk;
RPG	– Role-Playing Game;
SSAO	– screen space ambient occlusion;
ІКТ	– інформаційно-комунікаційні технології;
ІС	– інтелектуальна система;
ІТ	– інформаційні технології;
ОС	– операційна система;
ОБ	– алгоритм Олдоса-Бродера;
ПЗ	– програмне забезпечення;
ПК	– персональний комп'ютер.

ВСТУП

У сучасному світі комп'ютерні технології дедалі більше інтегруються в усі аспекти життя людини – від повсякденних справ до професійної діяльності та дозвілля. Це зумовлює необхідність постійного освоєння нових навичок роботи з персональним комп'ютером. Коли базові знання комп'ютерної грамотності вже опановані, настає етап підвищення швидкості й оптимізації роботи. Персональний комп'ютер здатен виконувати значно більше завдань і швидше, ніж людина, що робить його незамінним інструментом у різних сферах, зокрема у створенні математичних моделей і процедурної генерації контенту для ігрових додатків.

З кожним роком ігрова індустрія демонструє стрімке зростання, збільшуючи свої прибутки як у світі, так і в Україні. У цьому контексті розробка ігрових додатків із застосуванням сучасних технологій стає вкрай актуальною.

Метою роботи є створення ігрового додатка з використанням сучасних засобів розробки та методів процедурної генерації лабіринтів. Для досягнення поставленої мети необхідно виконати такі завдання:

- провести аналіз ключових аспектів процесу розробки ігрових додатків;
- здійснити огляд та класифікацію ігрових додатків;
- вивчити існуючі підходи до створення ігрових додатків;
- дослідити методи генерації лабіринтів, що використовуються в ігрових додатках;
- спроектувати концепцію ігрового додатка;
- розробити ігровий додаток із використанням Unity 3D та мови програмування C#.

Об'єктом дослідження є процес програмної реалізації генерації маршрутів у межах ігрового додатку.

Предметом дослідження є методи та інструменти процедурної генерації контенту для ігрових додатків, а також їх інтеграція в ігрове середовище. У роботі також розглядаються технології створення ігрового додатка.

Методи дослідження:

- теоретичні методи: аналіз науково-технічної літератури та інтернет-

ресурсів з проблеми дослідження;

– емпіричні методи: оптимізації розробки засобів процедурної генерації контенту у концепції комп'ютерної гри.

Практичне значення полягає в програмній реалізації генерації лабіринту у ігровому додатку засобами Unity 3D та C#.

Особистий внесок – розроблено ігровий додаток "Розкрадач гробниць" з реалізацією генерації лабіринту.

Структура і обсяг роботи

Робота складається з вступу, трьох розділів, висновків, списку використаних джерел, додатків. Обсяг роботи становить 72 сторінки, обсяг використаної літератури – 32 джерела.

Перший розділ містить опис особливостей і етапи розробки ігрових додатків. Надано огляд класифікації ігрових додатків.

У другому розділі проводиться дослідження існуючих методів процедурної генерації контенту в ігрових додатках, наведено порівняння існуючих алгоритмів генерації та зміст модифікованого методу генерації.

У третьому розділі надано програмну реалізацію генерації лабіринту ігрового додатку «Розкрадач гробниць» у середовищі Unity 3D користуючись інтегрованим середовищем розробки Visual Studio та мовою програмування C#.

Додатки містять сертифікат впровадження, головні елементи коду сценарію ігрової розробки.

РОЗДІЛ 1 АНАЛІЗ СУЧАСНОГО СТАНУ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Особливості процесу розробки ігрових додатків

При проектуванні будь-якої гри необхідно створити концепцію гри, визначити технології розробки і визначити спосіб візуалізації.

Ігрова концепція полягає в основному задумі гри, тобто розробка правил гри і спосіб взаємодії гравця з ігровим контентом. Задум гри, набір персонажів, візуальні ефекти руху персонажів, розробка сценаріїв вимагають певної кваліфікації геймдизайнера.

Технологія розробки забезпечує технічну основу для реалізації гри. Основну роль у цьому відіграє ігровий рушій, що забезпечує функціонал, необхідний для створення гри. Використання інструментів рушія для створення ігрового контенту дозволяє дещо спростити процес реалізації ігрової механіки.

Візуальний стиль який формує зовнішній вигляд гри. За створення візуальної складової відповідає команда художників, яка охоплює різні напрямки комп'ютерної графіки. Концепт-художники працюють над унікальним виглядом ігрового світу, 3D-художники моделюють персонажів, аніматори додають їм рухи, а фахівці з ігрового оточення створюють декорації, рівні та спецефекти.

Усі ці три елементи об'єднуються для досягнення головної мети – створення ігрового процесу, що забезпечує розвагу або розвиток гравця.

Взаємодія гравця з ігровим персонажем має власний термін – геймплей, що відображає механіку і динаміку гри, спрямовану на залучення й утримання інтересу гравця.

Геймплей – це те, що відрізняє комп'ютерну гру від пасивних форм розваг, таких як книги чи кіно. На відміну від функцій звичайного спостерігача геймплей має можливість користувачу взаємодіяти з контентом гри, приймати участь у розвитку сценарію гри.

При розробці геймплея можна виділити два етапи базовий та рівневий.

До базового геймплея відноситься набір правил і механік, що доступні гравцю. Фактично це основа всього задуму гри, що дозволяє в подальшому розвивати дизайн гри на наступних етапах.

На наступному етапі виникає завдання дизайнера рівнів створити локації, які забезпечують різноманітні ігрові ситуації, де кожна базова механіка може бути застосована. За допомогою організації простору, об'єктів і параметрів віртуального світу дизайнер формує геймплей рівнів, пропонуючи гравцю новий і унікальний досвід взаємодії з грою.

У різних жанрах ігор рівні можуть мати унікальні особливості.

Для стратегічних ігор це карти, які створюють геймплей через особливості ландшафту, розташування ресурсів і гравців.

Для ігор-перегонів це траси з перешкодами, різними маршрутами.

Для пригодницьких ігор геймплей має створювати на шляху гравця головоломки, перешкоди, пастки, ворогів.

Попри відмінності у структурі та назвах (карти, місії, зони, етапи, завдання), рівні в іграх виконують одну спільну функцію – розширюють і урізноманітнюють основний геймплей, пропонуючи нові виклики та сценарії для гравця.

Процес розробки гри можна поділити на наступні етапи [4]. Альфа, Пре-продакшн, Софт-ланч, Хард-ланч, Підтримка.

Рис. 1.1. Етапи розробки гри [4]

На стартовому етапі продюсером формуються набір умов для запуску проєкту та список вимог до нього. Завдання продюсера – розробити та погодити з керівництвом концепцію проєкту, а також визначити склад команди розробників.

На цьому етапі продюсер створює концепцію проєкту, що включає дані про цільову аудиторію, сетинг і жанр гри, основні елементи геймплею, підходи

до розробки, вибрані інструменти й технології, ринки збуту, підтримувані платформи, терміни виконання, бюджет і вимоги до команди розробників.

На етапі пре-продакшу визначаються ризики, що можуть впливати на реалізацію проекту. Для усунення або мінімізації ризиків, пов'язаних з грою, зазвичай проектується прототип гри, розробляється деталізований план проекту.

Після схвалення прототипу настає етап софт-ланча, на якому продюсером та командою ухвалюються план розробки першої повноцінної версії ігрового застосунку. Перевірка працездатності зазвичай доручається обраній групі користувачів. Зібрана статистика дозволяє оцінити потенціал ігрового проекту.

При позитивній оцінці наступає етап хард-ланч. На цьому етапі розробляється повна версія гри для глобальної реалізації.

Коли статистика проекту стабілізується на прийнятному рівні і проект більше не потребує значних доопрацювань і розширень, його передають на етап підтримки. Для цього формують команду підтримки, яка часто є частиною основної команди розробників. Команда підтримки поступово вдосконалює гру за допомогою дрібних оновлень та доповнень. Ці ітерації можуть тривати до того моменту, поки підтримка гри буде доцільною для керівництва. Якщо подальше обслуговування проекту стає економічно недоцільним, проект закривається.

Технічний погляд на розробку передбачає наступні етапи [26]: Концептування (Concept), Прототипування (Prototyping), Вертикальний зріз (Vertical Slice), Виробництво контенту (Content Production, Friends & Family / CBT (Closed Beta Testing)), Soft Launch / OBT (Open Beta Testing), Soft Launch / OBT (Open Beta Testing), Release.

Concept. На цьому етапі команда розробляє концепцію гри та початково опрацьовує ігровий дизайн. Геймдизайнер документує свої ідеї, що дозволяє виконавцю чітко розуміти завдання щодо реалізації продукту. Тестувальник має зрозуміти, що саме і як потрібно тестувати.

Важливо, щоб усі проєктні та продуктові документи постійно оновлювались на всіх етапах розвитку проєкту. Для ефективного використання документації слід використовувати спеціалізовані інструменти.

Prototyping. Прототип створюється для перевірки основного ігрового процесу, тестування різних гіпотез, ігрових механік та ключових технічних аспектів. Важливо, щоб прототип містив тільки необхідні елементи для перевірки і був розроблений в стислі терміни. Він має бути простим у реалізації, оскільки після досягнення поставлених цілей його слід відкинути.

Vertical Slice. Створення мінімальної повноцінної версії, яка включає повністю реалізований основний ігровий процес. Важливо, щоб висока якість була досягнута тільки для тих елементів гри, що значно впливають на загальне сприйняття продукту. Всі основні функції гри мають бути присутні, хоча б в чорновому вигляді. Для забезпечення повноцінного ігрового процесу реалізується мінімальний, але достатній набір контенту, наприклад, один рівень або локація.

Content Production. Створення ігрового контенту для реалізації на велику аудиторію. Цей процес потребує розробки всіх функцій проекту, удосконалення елементів проекту.

Friends & Family / CBT (Closed Beta Testing). Етап закритого бета тестування лояльною аудиторією. Основними завданнями на цьому етапі є пошук та виправлення геймдизайнерських помилок, вирішення проблем ігрової логіки та усунення критичних багів. В грі вже реалізовані всі основні функції, створено достатньо контенту для тривалої гри, а також налагоджено збір та аналіз статистики. Тестування проводиться за планом, включаючи стрес-тести з реальними гравцями.

Soft Launch / OBT (Open Beta Testing) – на цьому етапі триває тестування гри вже на великій аудиторії. Відбувається оптимізація для високих навантажень, і гра повинна бути готова до прийому великого трафіку. Впроваджено систему білінгу, і починаються прийом платежів. На цьому етапі завершується розробка нових функцій: відбувається feature freeze, програмісти припиняють реалізацію нових можливостей і фокусуються на налагодженні та оптимізації наявних функцій. Геймдизайнери, продюсери і аналітики аналізують дані з СВТ і оцінюють ефективність монетизації. До цього етапу інфраструктура

проекту повинна бути повністю функціональною: працюють сайт, групи в соцмережах, канали залучення користувачів та підтримка клієнтів.

Release – офіційний реліз гри, мета якого – отримання прибутку. Команда розробки продовжує виправляти технічні помилки, що виникають під час експлуатації, і оптимізує продукт. Геймдизайнери проводять точне налаштування геймплею під реальну ситуацію в грі, а також реалізують нові внутрішньоігрові функції для підтримки монетизації. Крім того, ведеться розробка та інтеграція нового контенту для підтримки інтересу гравців.

1.2 Огляд та класифікація ігрових додатків

Фахівці з ігрових технологій в освітній практиці пропонують таку класифікацію ігор: *за областю діяльності* (фізичні, інтелектуальні, трудові, соціальні, психологічні); *за характером педагогічного процесу* (навчальні, тренінгові, контролюючі, узагальнюючі, пізнавальні, виховні, розвивальні, репродуктивні, продуктивні, творчі, комунікативні, діагностичні, профорієнтаційні, психотехнічні); *за ігровою методикою* (предметні, сюжетні, рольові, ділові, імітаційні, драматизації); *за предметною областю* (математичні, хімічні, біологічні, фізичні, екологічні, музичні, театральні, літературні, трудові, технічні, виробничі, фізкультурні, спортивні, військово-прикладні, туристичні, народні, суспільнознавчі, управлінські, економічні, комерційні); *за ігровим середовищем* (без предметів, з предметами, настільні, кімнатні, вуличні, на місцевості, комп'ютерні, телевізійні, технічні, із засобами пересування).

1.2.1 Опис класифікації ігор за ігровими платформами

Комп'ютерна гра — форма розвиваючої-розважальної взаємодії користувача і комп'ютера, що імітує у віртуальному просторі життєві і уявні ситуації. Основним способом поділу відеоігор на категорії є поділ по платформа, який вказує, на якому пристрої можна запустити ту чи іншу гру. Якщо у

користувача немає платформи, для якої призначена гра, то і пограти в неї неможливо.

Персональний комп'ютер (ПК, РС, ноутбук, нетбук). Перелік найбільш популярних серій комп'ютерних ОС: Windows (від фірми Microsoft), Mac OS (від фірми Apple), Linux (безкоштовна ОС, що розроблюється світовою інтернет-спільнотою).

Ігрова консоль або приставка (PS, Xbox, Nintendo). До найбільш популярних поколінь консолей можна віднести: Sony PlayStation (PSP, PSOne, PS2 – PS5), Microsoft Xbox (Xbox, Xbox 360, Xbox One), Nintendo 3DS, Wii.

Мобільний пристрій: телефон, планшет, кишеньковий комп'ютер (КПК, PDA). На застарілих телефонах для гри є java-додатки, на сучасних телефонах ігри запускаються під мобільними ОС: Windows Mobil і Android. Для поширення мобільних ігор створені цілі глобальні сервіси типу App Store, Google Play.

Планшет, сенсорний мобільний телефон. Окремою категорією йдуть планшети і телефони з можливістю сенсорного введення (натисканням пальцями по екрану). Особливий спосіб введення даних дозволяє створювати ігри з унікальним геймплеєм, що використовують ці особливості, наприклад, малювання на екрані, нахил пристроїв для зміни гравітації в грі та ін.

Ігровий автомат. Браузерна або флеш-гра (віртуальна інтернет платформа), *браузерні ігри* - ігри, здатні запускатися у вікні браузера. Особливий пристрій браузерних ігор дозволяє грати в них з будь-якого пристрою, який може підключатися до інтернету. Всі популярні браузери: Google Chrome, Opera, FireFox, Edge, Safari – підтримують запуск невеликих програм прямо усередині інтернет-сторінок [1].

1.2.2 Огляд класифікації комп'ютерних ігор за змістом

Існує велика кількість ігрових жанрів і кожен з них характеризується певними властивостями. Для того, щоб зрозуміти до якого жанру буде

відноситися будь-яка гра, потрібно «розкласти» гру на її складові частини і визначити їх зв'язки один з одним.

Виділяють 3 основних групи: рольова, бойовик і стратегія. Оскільки існують різні оцінки приналежності гри до того чи іншого жанру не визначені, то однозначно жанр конкретного проєкту може бути визначений у різний спосіб. Крім того, одна і та гра може бути віднесена до декількох жанрів.

Загалом можна навести класифікацію ігор за жанром наступним чином:

1. Action: 3D-шутер, «бродилки-стрілялки» (шутери від першої і третьої особи; «криваві/м'ясні» шутери; тактичні шутери); файтінги (побий їх усіх; слешер; аркада; стелс-екшен).

2. Симулятори/менеджери: технічні; аркадні; спортивні; спортивний менеджер; економічні.

3. Стратегії: стратегії за схемою ігрового процесу

4. Пригоди – текстова пригодницька гра.

5. Графічний квест: головоломки; пригодницький бойовик; симулятор побачень; візуальна новела.

6. Музичні ігри – ритмічні ігри.

7. Ролеві ігри – тактичні Role-Playing Game (RPG).

8. Головоломки, логічні, пазли.

9. Традиційні і настільні.

10. Текстові – ігри в псевдографіці.

Класифікація за сеттінгом. Сеттінг (від англ. «Setting») – ігровий світ, в якому відбувається дія художньо твору. В створенні ігрові світів використовуються міфологічні історії та персонажі, генеруються вигадані персонажі, вигадані всесвіти.

За часом дії (історична епоха): зародження життя; ера динозаврів; зародження цивілізацій; середньовіччя; епоха колонізації; епоха; минулі війни; наш час; інформаційна епоха; освоєння космосу.

За умовами всередині світу: місце з певною культурою; певна кліматична зона; наявність катаклізму; хоррор (лякає, напружена атмосфера).

Класифікація за метою гри. **Казуальна** (повсякденна) гра. Гра побудована так, що її можна тимчасово перервати в будь-який момент, а потім продовжити. Часто процес гри розділений на невеликі рівні (приклад: Angry Birds, Plants vs Zombies).

Гра-пісочниця (творчі можливості, вибір цілей). Ігри без сюжету і цілей. Основою гри-пісочниці є різноманітні ігрові можливості, які гравець може застосовувати на власний розсуд. Досить часто пісочниці це не окремі ігри, а спеціальні режими в сюжетних іграх (приклад: Grand Thief Auto, Minecraft, SimSity).

Гра-змагання (дуель, чемпіонат, суперництво). Гра, в якій гравці змагаються між собою за статус переможця (приклад: StarCraft 2, Counter Strike, Battlefield).

Гра на проходження (виконання цілей, сюжет). У сюжетній грі всі ігрові завдання взаємопов'язані між собою, слідує одна за одною, утворюючи лінію сюжету.

Навчальна гра (отримання нових знань). Це загальні ігри для дітей або вузько спеціалізовані симулятори для дорослих. В процесі гри в ігровій формі подається інформація для вивчення.

Хардкорна (дуже складна) гра. Гра, створена спеціально для досвідчених гравців, для випробування їх ігрових навичок.

Опис класифікації ігор за видавничими критеріями

Класифікація за бюджетом розробки. Від того, які фінанси вкладені в гру, залежить її зовнішній вигляд, опрацювання деталей, різноманітність ігрових можливостей.

Професійна гра (ігри із середнім бюджетом). Ігри, створювані професіоналами, але без використання величезних бюджетів. В таких іграх

менше лиску, обмежене число спецефектів, різноманітних декорацій, але в цілому, відчувається професійний підхід.

Інді-гра (незалежна гра, малобюджетний клас). Ігри, створювані одним автором або невеликою групою. Інді-автори – це частіше за все професіонали в своїй справі, у них виходять відмінні гри. Єдиний мінус – при створенні інді-ігор не витрачаються кошти на покупку сторонніх фахівців, все створюється самостійно.

Любительська гра (безкоштовна гра з мінімальною якістю). Зараз існує безліч ігрових движків, що значно спрощують процес створення гри. Створенням гри може зайнятися навіть студент або школяр, що впевнено розбирається в комп'ютерах, але, наприклад, нічого не знаючий про ігровий баланс, дизайн та стилі. В результаті, найчастіше виходять ігри, в яких дуже багато недоліків.

Класифікація за видавничим форматом. **Оригінальна гра.** Будь-яка гра, що розповідає про якусь нову історію, що не копіює повністю старі ідеї з інших ігор, є оригінальною. Перша частина в серіях ігор так само є оригіналом.

Чергова гра в ігровій серії (сіквел, приквел, рімейк). Ігри, що отримали достатню популярність у гравців, продовжують розвиватися далі. Творці залишають старих героїв, інші улюблені ігрові елементи, і переносять їх в нові ігрові ситуації – так виходить продовження ігор.

Доповнення до гри. Менше за обсягом продовження гри, яке може працювати тільки в комплекті з оригінальною грою.

Скачуваний контент до гри (DLC). DLC - набір ігрових елементів (герої, зброї, костюми, корисні предмети, рівні, вороги), які не входять до стандартного набору гри, але які можна докупити у розробників за окрему плату та додати в гру. Будь-які неофіційні доповнення до гри, створені не розробниками, а гравцями-фанатами, називаються модифікаціями (скорочено – моди) [2].

1.2.4 Опис класифікації ігор за кількістю гравців

Гра без участі гравців (Zero Player Game). У деяких іграх є можливість налаштувати комп'ютерних супротивників так, що вони можуть битися між собою, а сам гравець навіть не братиме участі в грі.

Одиночна гра (сінглплеєр, англ. Singleplayer). Тип гри, в якій ігровий процес розрахований на одного гравця.

Спільна гра на одному пристрої (Hotseat, Splitscreen). Гра, в якій можуть брати участь одразу кілька гравців на одному пристрої.

Спільна гра на одному пристрої по черзі (Hotseat). Почергова гра можлива в іграх з покроковим режимом гри. Гравці здійснюють свої ходи по черзі, використовуючи один і той самий комп'ютер або консоль.

Гра, розрахована на багато користувачів (мультиплеєр, Multiplayer). Тип гри, в якій можуть брати участь відразу кілька гравців. Кожен гравець входить в гру через свій пристрій (комп'ютер, консоль, мобільний пристрій).

Масова онлайн гра (Massively multiplayer online game, ММО). Тип гри, в якій може брати участь величезна кількість людей (десятки і сотні тисяч гравців). Такі онлайн ігри побудовані за принципом «клієнт-сервер». Основна частина гри розташовується на спеціальному потужному комп'ютері, який постійно підключений до інтернету (сервер).

Розрахована на багато користувачів оффлайн-гра (PBEM - Play By Electronic Mail, «ігри по електронній пошті»)). Існує особлива категорія онлайн-ігор, в яких підключення до інтернету необхідно лиш на короткий проміжок часу, тільки для того, щоб передати інформацію про свій хід.

1.3 Висновки до першого розділу

На підставі аналізу та дослідження класифікації сучасних ігрових систем можна зробити висновок, що генерація контенту при розробці ігрового застосунку потребує достатньо великих обсягів витрат часу. Автоматизація створення контенту дозволяє підвищити ефективність процесу розробки, та зменшує вимоги до задіяних ресурсів.

РОЗДІЛ 2 МЕТОДИ ПРОЦЕДУРНОЇ ГЕНЕРАЦІЇ КОНТЕНТУ В ІГРОВИХ ДОДАТКАХ

Одним із важливих етапів розробки гри є створення її рівнів, які в більшості випадків представлені картами, що є лабіринтами. Оскільки лабіринти широко використовуються для побудови ігрових локацій, гравець практично завжди стикається з ними під час гри.

Всі локації будуть так чи інакше схожі між собою, що зменшує інтерес до гри. Тому ручне створення лабіринтів є неефективним варіантом. З цієї причини розробники ігор часто вдаються до процедурної генерації лабіринтів, щоб кожне нове проходження гри відбувалося на ново згенерованій карті.

Для збільшення інтересу користувачів та підвищення інтерактивності в іграх, що побудовані на лабіринтах, часто додаються предмети, з якими гравець може взаємодіяти. Для спрощення і покращення розробки геймплея доцільно поєднати генерацію таких предметів із процесом створення лабіринту.

Іншою проблемою, з якою стикаються розробники при створенні лабіринтів, є визначення місця розташування виходу. Це завдання також варто автоматизувати, щоб вихід з'являвся в різних випадкових точках на кожному рівні. Оскільки лабіринт генерується автоматично, вихід також слід генерувати під час його створення. Для підвищення інтересу гравця, вихід повинен розташовуватися якомога далі від входу.

Класифікація та типізація лабіринтів

Лабіринт – це структура (зазвичай у двовірному або тривірному просторі), що складається з заплутаних шляхів, які ведуть до виходу або в глухі кути. В давньогрецькій та римській традиції лабіринт уявлявся як великий простір, що включає численні зали, кімнати, двори та проходи, розташовані за складним і заплутаним планом виходу [26].

Лабіринти можна поділити на різними параметрами. Зокрема, це вимірність, топологія, мозаїка, маршрутизація [7, 22].

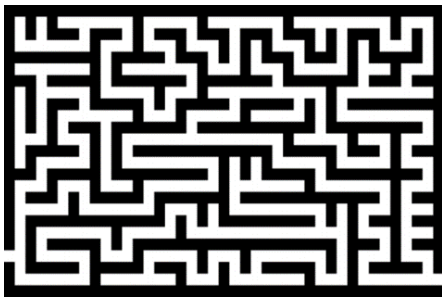
Вимірність визначає кількість вимірів простору, в якому розташований лабіринт.

2D (двовимірний). Більшість лабіринтів, чи то на папері, чи в реальному світі, є двовимірними, і їх можна зобразити на аркуші паперу, де можна рухатися по схематичному плану, не перекриваючи інші проходи (рис.2.1, а).

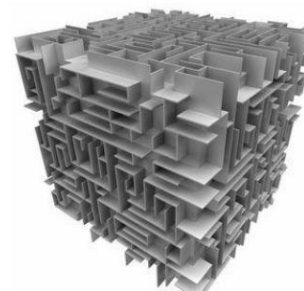
3D (тривимірний). Для побудови тривимірного лабіринту зазвичай використовуються набори двовимірних лабіринтів з додаванням спроможності переходу між ними (рис.2.1, б).

Higher-dimensions (багатовимірні). Це можуть бути лабіринти у 4D або з ще більшою кількістю вимірів. Вони зображуються як 3D-лабіринти, до яких додаються спеціальні "портали", через які можна переміщатися в 4-е вимірювання (рис.2.1, в).

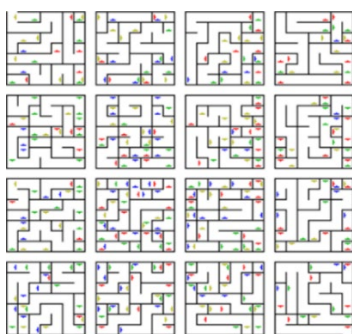
Weave (переплетені). Це лабіринти з перекриттям проходів. (рис.2.1, г).



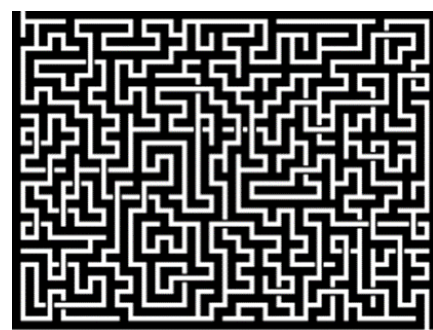
а) 2D лабіринт



б) 3D лабіринт



в) 4D лабіринт



г) Weave лабіринт

Рис. 2.1. Вимірні лабіринти

Топологія визначає спосіб неперервного з'єднання геометричних просторів для створення лабіринту. Лабіринти з незвичними з'єднаннями стін називаються Planair (рис.2.2).

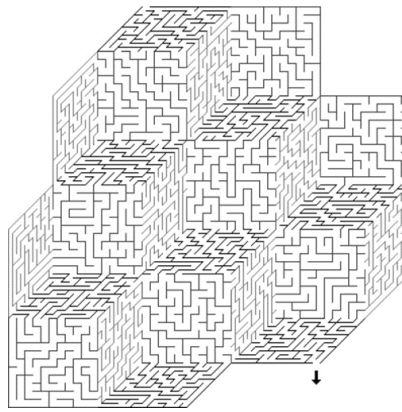


Рис. 2.2. Planair лабіринт

Мозаїка (рис.2.3). визначає спосіб складання лабіринту з окремих клітин певної геометрії.

Ортогональний складається прямокутників з проходами між клітинами, що перетинаються під прямим кутом.

Лабіринт Delta складається з з'єднаних трикутників, кожен з яких може мати до трьох проходів, що ведуть до інших комірок (рис. 2.3, а).

Лабіринт Sigma формується з з'єднаних шестикутників, де кожна комірка може мати до шести проходів (рис. 2.3, б).

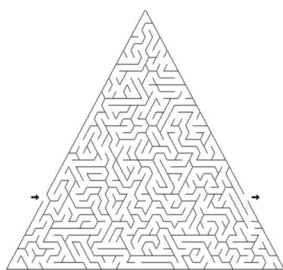
Лабіринт Theta формується з концентричних кіл. Вхід та вихід зазвичай розташовуються в центрі та на зовнішньому контурі лабіринта. Комірки зазвичай мають чотири можливих з'єднання, але кількість проходів може зрости, якщо в зовнішніх кільцях є більше клітин (рис. 2.3, в).

Лабіринт Upsilon формується з чотирикутників та восьмикутників, що з'єднані між собою (рис. 2.3, г).

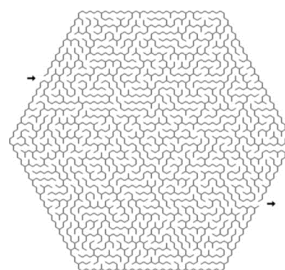
Лабіринт Zeta фактично складається з прямокутників з прямокутними та діагональними переходами. (рис. 2.3, д).

Лабіринт Crack – безформний лабіринт без постійної мозаїки, в якому стіни або проходи знаходяться в випадкових точках (рис. 2.3, ж).

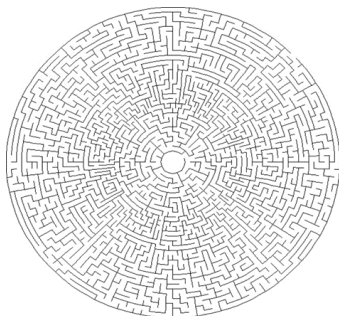
Лабіринт Fractal створюється вкладених один в один клітинок, які є лабіринтами, які, в свою чергу, також містять лабіринт. Такий лабіринт можна створити використовуючи вкладення багато разів (рис. 2.3, з).



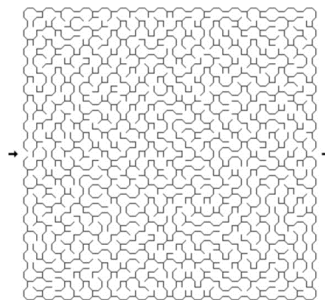
а) Лабіринт Delta



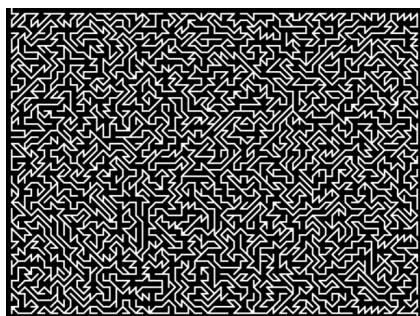
б) Лабіринт Sigma



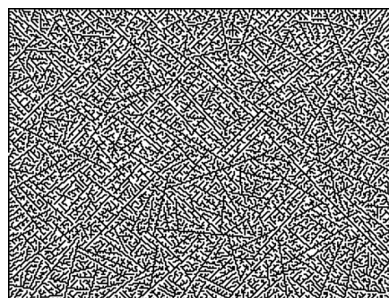
в) Лабіринт Theta



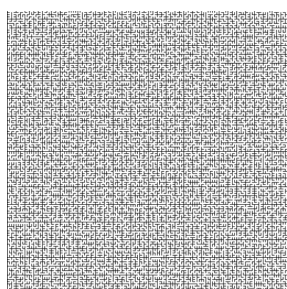
г) Лабіринт Upsilon



д) Лабіринт Zeta



ж) Лабіринт Crack



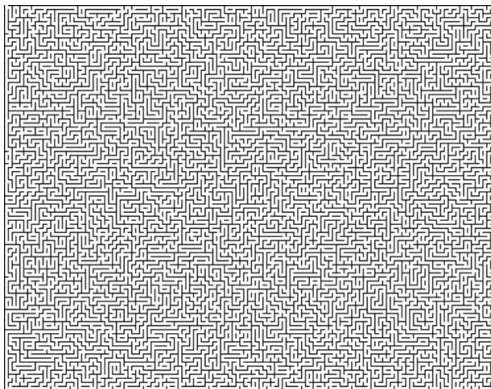
з) Лабіринт Fractal

Рис. 2.3 Мозаїка лабіринтів

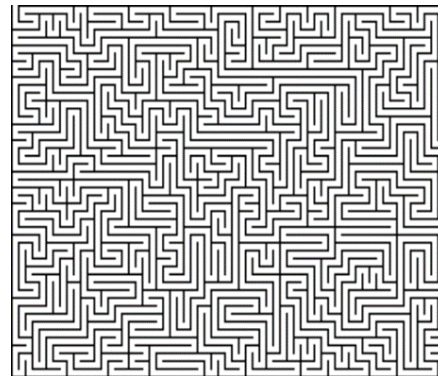
Маршрутизація. Клас маршрутизації визначає спосіб генерації лабіринту. Лабіринти можуть мати різні типи проходів (рис. 2.4):

Досконалий – лабіринт без замкнутих циклів, витків або недоступних ділянок. З кожної точки завжди є принаймні один шлях до іншої точки, і лабіринт має хоча б одне розв'язання. У термінах комп'ютерних наук такий лабіринт можна описати як кістякове дерево, що складається з клітин і вершин.

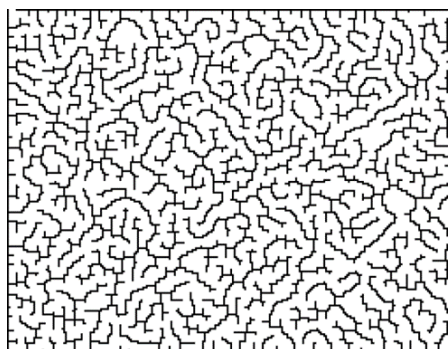
Коса – лабіринт без тупиків, де проходи переплітаються один з одним, створюючи коло (тому його називають "коса"). Цей тип лабіринту змушує гравця витрачати час, рухаючись по колу, замість того щоб потрапляти в глухий кут. Добре продуманий косий лабіринт може бути значно складнішим, ніж досконалий лабіринт тієї ж величини (рис. 2.4, а).



а) Лабіринт-коса



б) Унікурсальний лабіринт



в) Розріджений лабіринт

Рис. 2.4. Маршрутизація лабіринтів

Унікурсальний – лабіринт без перехресть, що складається з одного довгого змієподібного проходу. Проходити такий лабіринт не надто складно, якщо тільки гравець не заблукає на середині шляху та не змушений буде повертатися назад (рис.2.4, б).

Розріджений – у цьому лабіринті не кожна клітина має прохід, деякі проходи відсутні (рис.2.4, в). Такі лабіринти можуть включати недоступні області і є своєрідною протилежністю лабіринту-косі.

2.2 Алгоритми процедурної генерації контенту в ігрових додатках

Алгоритм двійкового дерева

Алгоритм (BSP – Binary Space Partitioning) призначений для створення випадкового маршруту з кожної клітини поля. Після цього обробляється лише одна клітина за раз, що дозволяє генерувати лабіринти теоретично нескінченного розміру, зберігаючи тільки фінальний результат (лабіринт), без потреби зберігати будь-яку додаткову інформацію [6,13].

Цей метод має два побічні ефекти: лабіринти часто мають виражене діагональне зміщення і відсутність тупиків у відповідному напрямку; також утворюються два порожніх коридори з боків лабіринту. Коли алгоритм доходить до кінця рядка чи стовпця, він змушений продовжити шлях у єдиному напрямку, створюючи порожні «кордони».

У побудованому випадковому двійковому дереві з кожної комірки існує єдиний шлях у до кореня.

Етапи алгоритму полягають к обранні початкової клітки, обрання випадкового напрямку в межах поля, перейти до наступної комірки.

Алгоритм має просту реалізацію, дозволяє швидко генерувати лабіринти. Проте складність створюваних алгоритмів низька.

Приклад роботи алгоритму двійкового дерева наведено на рис. 2.5.

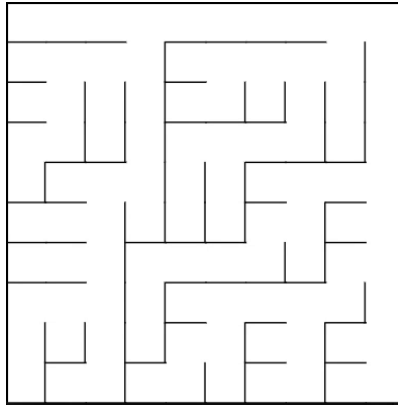


Рис. 2.5. Робота алгоритму BSP

Алгоритм Еллера

Алгоритм Еллера – це математичний генератор, що дозволяє створювати лабіринти, у яких між кожними двома точками існує єдиний шлях, тобто лабіринти не містять циклів.

«Алгоритм полягає у циклі додавання нових рядків, кожен з яких має однакову кількість клітинок, визначену в початковий момент. Клітинки належать множинам, що контролюють можливість проходу між ними. Під час генерації поточного рядка, клітинки однієї множини з'єднуються між собою, тоді як клітинки з різних множин залишаються ізольованими. Кожна клітинка може мати або не мати праву та нижню стінку. Загалом, стінки генеруються випадковим чином, але з дотриманням правил, які забезпечують відсутність циклів у лабіринті» [27].

Приклад згенерованого алгоритму наведено на рис. 2.6.



Рис. 2.6. Приклад згенерованого лабіринту [26]

Алгоритм Sidewinder

За алгоритмом Sidewinder лабіринт генерується по одному рядку за раз: для кожної клітинки випадковим чином визначається, чи вирізати прохід вправо. Вирізання проходу дозволяє коригувати процес горизонтального формування лабіринту. Прохід на інший рядок формується з випадково обраної клітини актуального рядка.

На відміну від алгоритму двійкового дерева, де прохід завжди йде вгору від лівої клітинки горизонтального проходу, в алгоритмі Sidewinder напрямок підйому обирається випадково з будь-якої клітинки цього проходу. Якщо в лабіринті двійкового дерева по верхньому і лівому краю є один довгий прохід, то в лабіринті Sidewinder лише верхній край має один довгий прохід.

Як і в двійковому дереві, лабіринт Sidewinder можна вирішити без помилок і детерміновано, рухаючись знизу вгору, оскільки в рядку є лише один прохід, що веде вгору [15].

Формальний алгоритм має складається з таких етапів:

1. Вибір початкового ряду.
2. Вибір початкової клітинки в ряду.
3. Ініціалізація порожньої множини.
4. Додавання клітинки до множини.
5. Визначення потреби прокладання проходу вправо.

За позитивного рішення, перейти до нової клітинки і зробити її поточною.

Повторити кроки 3-6.

За негативної відповіді, вибрати випадкову клітинку з множини і прокласти шлях вгору. Перейти до наступного ряду і повторити кроки 2-5.

6. Продовжувати, поки не буде оброблений кожен ряд.

Цей алгоритм дозволяє генерувати нескінченні лабіринти, наявність лише одного порожнього коридору та складніший малюнок порівняно з алгоритмом двійкового дерева. Однак серед недоліків варто відзначити більш складну реалізацію, відсутність тупиків при зміщенні [15].

Приклад роботи алгоритму Sidewinder наведено на рис. 2.7.

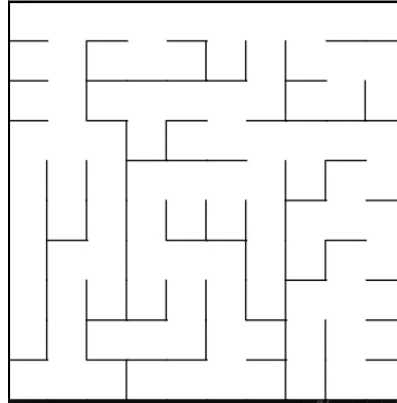


Рис. 2.7. Робота алгоритму Sidewinder [26]

Алгоритм Олдоса-Бродера

Алгоритм може бути реалізований як метод додавання стін. При такому підході алгоритм працює майже в два рази швидше, оскільки телепортація дозволяє швидше потрапляти до віддалених частин лабіринту [5].

Алгоритм Олдоса-Бродера є надзвичайно простим і здатний створювати дуже заплутані лабіринти. Це пояснюється тим, що алгоритм виконує лише три основні дії:

1. Вибір випадкової клітинки на полі.
2. Перехід до будь-якої сусідньої клітинки.
3. Якщо ця клітинка ще не була переглянута, зруйнувати стінку між клітинками і повернутися до кроку 2. Якщо ж клітинка вже була відвідана, просто перейти до кроку 2.

Утворений лабіринт є максимально непередбачуваним.

Основний недолік алгоритму — це тривалий час, необхідний для побудови лабіринту. Навіть для досить невеликого лабіринту цей процес може зайняти значний час. Крім того, алгоритм не підтримує генерацію нескінченних лабіринтів і стає менш ефективним до кінця генерації [13].

Приклад згенерованого алгоритму наведено на рис. 2.8.

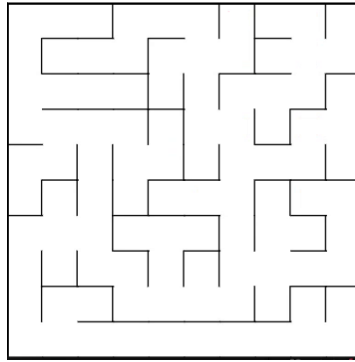


Рис. 2.8. Згенерований лабіринт [26]

Алгоритм Вілсона

Алгоритм Вілсона є вдосконаленою версією алгоритму Олдоса-Бродера.

Пам'ять, яку використовує цей алгоритм, дорівнює розміру лабіринту. Алгоритм Вілсона як і алгоритм Олдоса-Бродера витрачає значний час на пошук першого випадкового шляху до початкової клітинки.

У разі додавання стін алгоритм працює вдвічі швидше, оскільки вся стіна кордону вже є частиною лабіринту, і тому початкові стіни приєднуються значно швидше.

Алгоритм Вілсона генерує повністю випадкові лабіринти без будь-якого зсуву [16].

У алгоритмі Вілсона обрану випадкову вершину, яка не належить до кістяка, і додаю до дерева. На наступному етапі через випадкову вершину, що не належить до кістяка, проводиться обхід графа до досягнення вже доданої вершини дерева. Вершини новоутвореного підграфа додаються до кістяка.

Недоліки: складна реалізація; повільний старт генерації; більші вимоги до пам'яті, ніж у алгоритму Олдоса-Бродера.

Приклад роботи алгоритму Вілсона наведено на рис. 2.9.

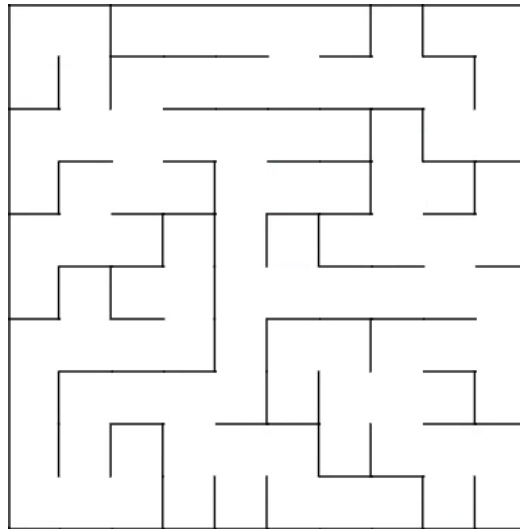


Рис. 2.9. Робота алгоритму Вілсона [26]

Алгоритм Прима

Алгоритм Прима є рандомізованою версією класичного алгоритму Прима для побудови мінімального кістякового дерева в неорієнтованому зваженому графі. Замість того, щоб вибирати найбільш оптимальний шлях переходу до наступної комірки, в алгоритмі Прима для генерації лабіринтів випадковим чином обирається комірка для подальшого руху. Алгоритм зберігає список сусідніх комірок і потребує пам'яті, пропорційну розмірам лабіринту.

Алгоритм починається з вибору довільної комірки і поступово розширює лабіринт, додаючи сусідні комірки, при цьому стежить, щоб нові комірки не з'єднувалися з уже відвіданими частинами лабіринту. Рандомізований підхід замінює вибір найменшої ваги на випадкове додавання нових комірок до лабіринту [9].

Вихідна точка алгоритму вибір сітки з заповненими стінами. Обираючи комірку, фіксується її приналежність до лабіринту та додаються її стіни до списку стін. Поки список стін не порожній обирається випадкову стіну зі списку. Якщо одна з двох комірок, які розділяє стіна, не була відвідана, то зробити стіну прохідною і позначити невідвідану комірку як частину лабіринту. Стіни сусідніх комірок додаються у список стін.

Лабіринт створений за алгоритмом Прима наведено на рис. 2.10.

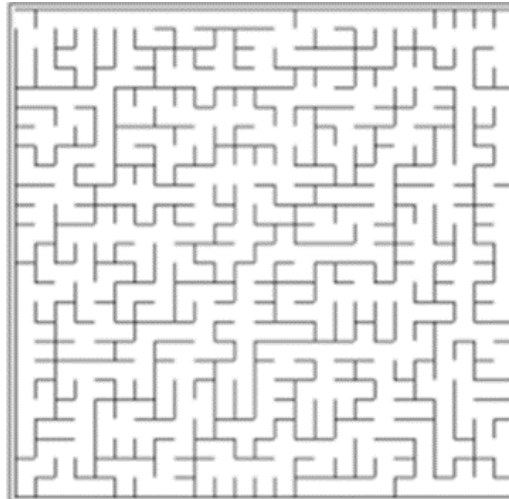


Рис. 2.10. Лабіринт за алгоритмом Прима

Алгоритм Крускала

Алгоритм Крускала для створення лабіринту є рандомізованою версією класичного алгоритму Крускала, який застосовується для побудови мінімального кістякового дерева в зваженому графі. Особливість цього алгоритму полягає в тому, що він не «будує» лабіринт, розширюючи його як дерево, а натомість випадковим чином вирізає сегменти проходів по всьому лабіринту. В результаті цього підходу створюється ідеальний лабіринт.

Рандомізована версія алгоритму змінює перший крок традиційного циклу, де замість вибору ребра з найменшою вагою, випадковим чином обирається ребро з набору. Алгоритм вимагає зберігання структури даних для лабіринту, а також здатності випадковим чином перераховувати всі ребра між комірками [14].

Після того, як всі ребра зібрані в набір і визначено унікальний ідентифікатор для кожної підмножини, алгоритм вибирає випадкове ребро, перевіряє, чи належать його комірки різним підмножинам, і якщо так, об'єднує ці підмножини, встановлюючи однакові ідентифікатори для обох частин лабіринту.

Приклад роботи алгоритму Крускала наведено на рис. 2.11.

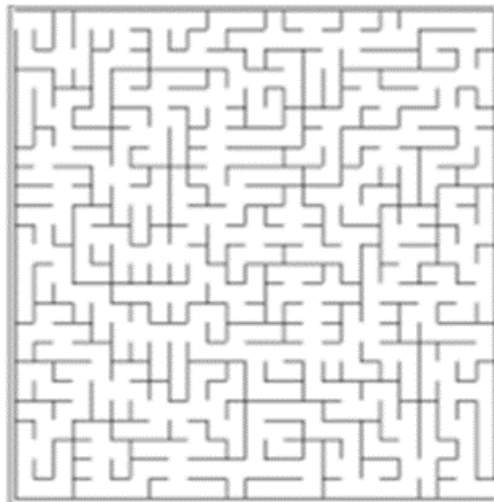


Рис. 2.11. Робота алгоритму Крускала

2.3 Аналіз алгоритмів генерації лабіринтів

Для аналізу характеристик алгоритмів використовуються агентів пошуку. Агент и пошуку відрізняються реалізацією пошук рішення.

Агент випадкового пошуку (Random Walk – RW) рухається до випадково обраного сусіда. Агент пошуку в глибину (Depth First Search – DFS) продовжує шлях, доки не натрапить на глухий кут, після чого повертається до першого вузла з невідвіданими сусідами. Агент евристичного пошуку в глибину (Heuristic Depth First Search – HDFS) обирає напрямки на основі простої евристики. Агент пошуку в ширину (Breadth First Search – BFS) застосовує принципи BFS [8] для розв'язання лабіринту, але замість дослідження початкових вузлів він одразу переходить до кінця лабіринту.

Експериментальний аналіз тимчасової складності алгоритмів, метою якого була оцінка якості їх результатів та проаналізовано співставлення часу виконання алгоритмів генерації лабіринту до розмірів лабіринту проведений у роботі [26]. Результат представлений на рис. 2.12.

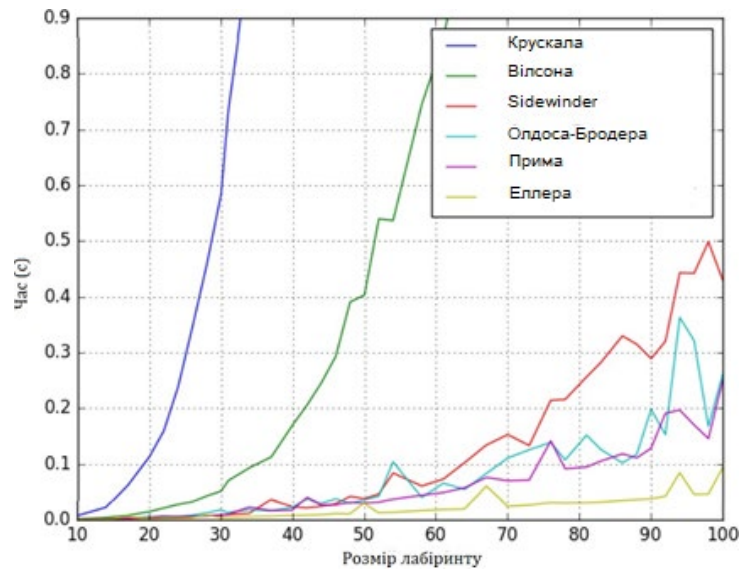


Рис. 2.12. Співвідношення часу виконання алгоритму до розмірів лабіринту

Складність лабіринту визначається його розміром, кількістю перетинів, кількістю тупиків.

У таблиці 2.1 наведено кількісні характеристики лабіринтів (100×100), створених з різними алгоритмами кожного типу.

Наведені дані демонструють, що деякі лабіринти, створені за допомогою схожих алгоритмів, мають подібну поведінку. Зокрема, алгоритми Олдоса-Бродера і Вілсона, які базуються на методах побудови однорідного кістякового дерева, мають майже однакову кількість перехрещень і тупиків. Алгоритми Еллера і Sidewinder вирізняються тим, що вони мають значно нижчі значення кількості перетинів та тупиків порівняно з іншими лабіринтами, оскільки вони базуються на DFS. Алгоритми Крускала і Прима, що походять від методів пошуку на графах, не демонструють таких чітких характеристик, як попередні групи.

Основним критерієм складності лабіринту є кількість кроків, які агент здійснює під час його проходження. Крок визначається як перехід від поточної клітини до сусідньої (таблиця 2.2).

Оцінки складності лабіринтів наведені в наступних таблицях 2.3-2.4.

Таблиця 2.1 – Середня кількість перетинів і тупиків в лабіринтах

Алгоритм	Кількість перетинів (x1000)	Кількість тупиків (x1000)	Ранг
Прима	2946	3559	1
Крускала	2654	3058	2
Олдоса-Бродера	2577	2933	3
Вільсона	2576	2932	4
Sidewinder	920	939	5
Еллера	869	898	6

Таблиця 2.2 – Середня кількість кроків необхідна для проходження лабіринту

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Sidewinder	7,3М	5,3k	13,3k	3,5k	1
Олдоса-Бродера	4,3М	7,1k	12,6k	3,1k	2
Вільсона	4,3М	7,2k	12,4k	3,0k	3
Еллера	9,1М	2,2k	12,7k	2,7k	4
Крускала	4,0М	6,8k	12,5k	2,8k	5
Прима	2,3М	5,6k	15,4k	1,7k	6

Таблиця 2.3 – Середня кількість відвіданих перетинів для кожного агента

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Олдоса-Бродера	1.8М	2.9k	5.1k	850	1
Вільсона	1,7М	2,9k	5,0k	844	2
Крускала	1,7М	2,9k	5,2k	800	3
Прима	1,1М	2,7k	7,3k	567	4
Sidewinder	1,0М	750	1.8k	337	5
Еллера	1.2М	232	2,0k	211	6

Таблиця 2.4 – Середня кількість відвіданих тупиків для кожного агента

Алгоритм	RW	DFS	HDFS	BFS	Ранг
Олдоса-Бродера	0,6М	1023	1863	823	1
Вілсона	0,6М	1035	1840	816	2
Краскала	0,6М	1030	1927	769	3
Прима	0,4М	974	2759	535	4
Еллера	0,4М	75	681	190	5
Sidewinder	0,4М	249	626	307	6

Ранг кількості відвіданих тупиків приблизно схожий з рангом кількості відвіданих перетинів.

З'ясувавши ранг алгоритмів, з'являється можливість визначення характеристик, які розрізняють різні рівні складності алгоритмів.

Кількість перетинів взаємопов'язана зі складністю лабіринтів. Більша кількість перетинів означає, що лабіринт складніше, і що є більше шансів пропустити правильний шлях.

Значення рівня складності в залежності від типу алгоритму:

2.4 Розробка модифікованого методу генерації лабіринтів

На основі аналізу, проведеного в попередньому розділі, найбільш ефективними для використання є алгоритми Олдоса-Бродера та Вілсона. Тому виникає необхідність детально розглянути їхню суть.

Алгоритм Олдоса-Бродера будує лабіринт шляхом вибору випадкових невідвіданих комірок. Усі клітинки поділяються на два типи: невідвідані та ті, що вже додані до лабіринту.

Алгоритм виконується за такими кроками:

1. Ініціалізувати лабіринт, обравши випадкову початкову комірку.
2. Обрати одну з сусідніх до поточної комірок (залежно від розташування, їх може бути 2, 3 або 4).

3. Якщо вибрана комірка ще не є частиною лабіринту, додати її до лабіринту, видаливши стінку між нею та поточною. Інакше повторити вибір сусідньої комірки.

4. Зупинити алгоритм, якщо всі клітинки відвідані. Інакше повернутися до кроку 2.

Алгоритм працює виключно із сусідніми комірками. Наприклад, якщо початкова комірка — (2,2), то її сусідньою може бути, наприклад, комірка (4,2). Якщо (4,2) додається до лабіринту, алгоритм повторює вибір сусідньої клітинки вже для (4,2). Однак через те, що алгоритм не запам'ятовує відвідані комірки, сусідньою для (4,2) може знову виявитися (2,2). Таке «блукання» між уже відвіданими комірками є основним недоліком алгоритму: зі збільшенням розмірів лабіринту час його побудови може стати нескінченним.

Наприклад, на полі розміром 101×101 , якщо залишиться лише дві невідвідані комірки, ймовірність випадково потрапити до них буде майже нульовою.

Алгоритм Вілсона усуває цей недолік. Усі невідвідані комірки додаються до спеціального списку, а цикли, що можуть утворитися, видаляються. Циклом вважається замкнений шлях, який починається і закінчується в одній і тій самій комірці лабіринту.

Алгоритм Вілсона виконується за такими етапами:

1. Додати всі клітинки лабіринту до списку невідвіданих.
2. Обрати довільну початкову клітинку, додати її до лабіринту та видалити зі списку невідвіданих.
3. Випадково вибрати клітинку зі списку невідвіданих.
4. Вибрати випадкову сусідню клітинку для поточної.
5. Якщо сусідня клітинка вже належить до лабіринту:
Додати всі пройдені під час поточної ітерації клітинки до лабіринту.
Прибрати стінки між цими клітинками.
Видалити їх зі списку невідвіданих.
- 5.1. Якщо утворився цикл:

Видалити цикл, повернувши всі клітинки, які його утворили, до списку невідвіданих.

Продовжити алгоритм із клітинки, з якої розпочався цикл.

5.2. Інакше перейти до сусідньої клітинки й продовжувати виконувати крок 4, доки не виконається крок 5.

6. Якщо список невідвіданих порожній, завершити роботу. Інакше перейти до кроку 3.

Видалення циклів: Припустимо, рух розпочався з клітинки (3,3). Алгоритм Вілсона знаходить множину невідвіданих клітин A , остання з яких приводить назад до (3,3), утворюючи цикл. У такому випадку всі клітинки із множини A знову позначаються як невідвідані, а алгоритм продовжується з точки (3,3).

Однак це не гарантує, що при подальшій роботі з тієї ж точки (3,3) цикл A (можливо, із тих самих клітинок) не утвориться знову.

Алгоритм Вілсона працює значно ефективніше за Олдоса-Бродера, однак вимагає додаткової пам'яті для зберігання списку невідвіданих клітин. Видалення циклів також уповільнює роботу, але забезпечує коректність побудови лабіринту.

Завдяки своїм характеристикам алгоритм Вілсона найкраще підходить для виконання поставленого завдання. Однак у базовій версії він генерує лабіринт лише з одним проходом, тому виникає необхідність у його модифікації.

Таку модифікацію дозволяє зробити ще одна особливість: для алгоритму не важлива форма поля лабіринту, тому можна генерувати лабіринт частинами. Іншими словами, можна задати будь-які обмеження на індекси двовимірного масиву. Наприклад, будувати лабіринт алгоритмом Вілсона в разі, коли індекси задовольняють деяким обмеженням, наприклад, $5 < i < 21$, а $j > i / 2$, де i відповідає за рядки лабіринту, а j – за стовпці.

Примітно, що у лабіринті, побудованому в обмеженому нерозривному просторі, також буде прохід з будь-якої комірки в будь-яку.

Тому було введено поняття циклової частини лабіринту – це частина лабіринту, яка утворює замкнутий простір. Відповідно, крім самої циклової

частини, позначеної як 1, поле розбивається ще на дві частини – частина всередині циклу і поза ним.

Циклічну частину можна задати наступним чином:

$$A = \left\{ \left\lfloor \frac{N}{8} \right\rfloor < i < \left\lfloor N - \frac{N}{8} \right\rfloor \right\}$$

$$B = \left\{ \left\lfloor \frac{N}{2} - 3 \right\rfloor < i < \left\lfloor \frac{N}{2} + 3 \right\rfloor \right\}$$

Використовуємо простір A/B .

Модифікований алгоритм Вілсона складається з таких етапів:

1. Виділити циклічну частину в початковому лабіринті (рис. 2.13, а).
2. За допомогою базового алгоритму Вілсона згенерувати лабіринт у межах циклічної частини (рис. 2.13, б).
3. Використовуючи базовий алгоритм Вілсона, побудувати лабіринти всередині циклу (рис. 2.13, в) і поза ним (рис. 2.13, г).
4. Створити прорізи на межі між циклічною та зовнішньою частинами лабіринту, зробивши заздалегідь визначену кількість проходів (рис. 2.13, д).

Принцип створення прорізів: Якщо межа області має форму прямокутника, можна зробити k прорізів на двох сусідніх сторонах прямокутника, а потім додати ще k прорізів на двох інших сторонах, розташованих діаметрально протилежно. У результаті завжди утворюється парна кількість прорізів.

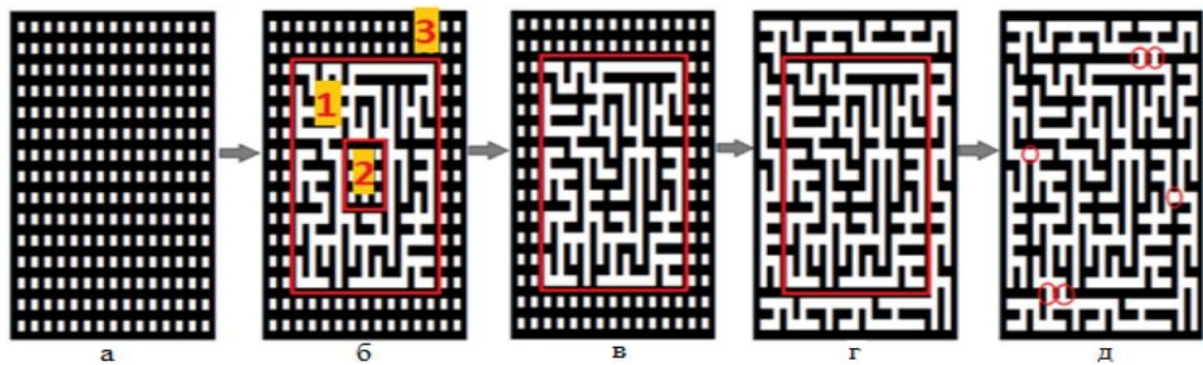


Рис. 2.13. Модифікований алгоритм Вілсона

Виконано модифікацію алгоритму Вілсона, який працює набагато ефективніше тому, що включає наявність засобів видалення циклів, яке уповільнює його час. Цей алгоритм не вимагає додаткових витрат пам'яті для зберігання списку невідвіданих клітин.

2.5 Висновки до другого розділу

В розділі проведено дослідження класифікації та типізації існуючих лабіринтів. Розглянуто алгоритми процедурної генерації контенту в ігрових додатках, та вплив їх модифікацій на утворення ігрового простору. Щоб зрозуміти значення налаштувань, порівняно існуючі алгоритми генерації лабіринтів та проаналізовано властивості лабіринтів і їх тимчасові характеристики. Також досліджено використання агентів пошуку, які допомагають зрозуміти, наскільки складно вирішити той чи інший лабіринт та оцінено атрибути складності лабіринтів.

РОЗДІЛ 3 РОЗРОБКА ІГРОВОГО ДОДАТКУ

3.1 Інструментальні засоби реалізації проєкту

Основним інструментом для реалізації проєкту є ігровий рушій Unity 3D. Unity – це багатоплатформне середовище для створення двовимірних і тривимірних додатків та ігор, яке працює під операційними системами Windows та macOS. Програми, створені за допомогою Unity, сумісні з Windows, macOS, Windows Phone, Android, iOS, Linux, а також із консолями Wii, PlayStation 3 і Xbox 360. Крім того, є можливість розробки веб-додатків за допомогою спеціального модуля Unity для браузерів. Unity підтримує DirectX і OpenGL [18].

Особливості Unity:

- Можливість створення сценаріїв на C#, JavaScript (модифікована версія) та Boo (діалект Python із строгою типізацією для платформи .NET). У кожній із мов внесені модифікації для підтримки всіх функцій внутрішньої скриптові мови UnityScript.
- Інтеграція ігрового рушія із середовищем розробки, що дозволяє тестувати гру безпосередньо в редакторі.
- Просте керування ресурсами через функцію Drag&Drop і настроюваний інтерфейс редактора.
- Реалізована система наслідування об'єктів.
- Підтримка імпорту з широкого спектра форматів.
- Вбудована підтримка мережових функцій.
- Інструменти для спільної роботи над проєктом, зокрема система контролю версій (Version Control).

Для написання і налагодження коду використовується середовище розробки під ОС Windows – Microsoft Visual Studio. Це середовище повністю інтегроване з Unity3D і операційною системою Windows. Для створення графічних об'єктів, які використовуються в генерації лабіринту, – використовується графічний редактор Blender. Він надається для розробки

моделей у форматі, відповідному для Unity 3D на безкоштовній основі.

3.2 Вихідні дані проєкту

У більшості алгоритмів створюються "ідеальні" щільні лабіринти, тобто такі, у яких є тільки один вірний шлях і немає петель. Вони схожі на лабіринти, які публікуються в газетних розділах "пазли" (рис. 3.1).

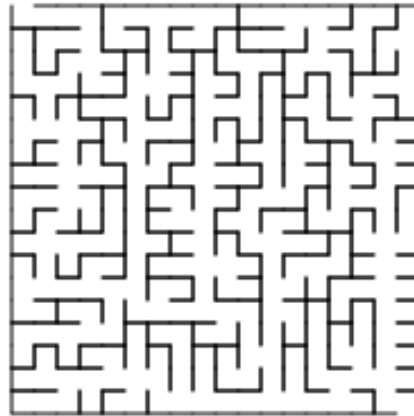


Рис. 3.1. Приклад пазла-лабіринту

Однак в більшість ігор приємніше грати, коли лабіринти неідеальні і в них є петлі (рис. 3.2). Вони повинні бути великими і складатися з відкритих просторів, а не з вузьких звивистих коридорів. Це особливо справедливо для жанру rogue-like, в якому процедурні рівні є не стільки "лабіринтами", а скоріше підземеллями.

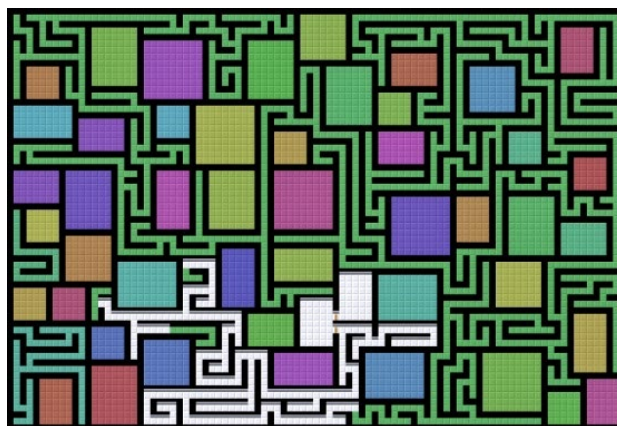


Рис. 3.2. Лабіринт з петлями

В роботі реалізовано один з алгоритмів генерації лабіринтів, обраний для

реалізації лабіринтів в грі з мінімальною кількістю зусиль. Такий підхід добре працює в класичних іграх з лабіринтами, тому можна використовувати його для створення лабіринтів в грі під назвою "Розкрадач гробниць".

У цій грі кожен рівень – це новий лабіринт, в якому захована скриня зі скарбом. Однак у гравця не так багато часу на його пошуки і втечу до того, як повернуться стражники. На кожному рівні є обмеження в часі і користувач може грати, поки його не зловлять. Набрані бали залежать від кількості вкрадених скарбів.



Рис. 3.3. Приклад початку гри

Проект має наступний вміст:

- папка Graphics, в якій міститься вся необхідна для гри графіка;
- сцена Scene – вихідна, що містить гравця і UI;
- папка Scripts , що містить скрипти.

3.3 Архітектура програмного додатку

Початок – додавання в сцену порожнього проекту. Вибрано GameObject – Create Empty, названо його Controller і розташовано в (X: 0, Y: 0, Z: 0). Цей об'єкт буде просто точкою приєднання скриптів, керуючих грою.

В папці Scripts проекту створено скрипт C # з назвою GameController, а потім створено ще один скрипт і названо його MazeConstructor. Перший скрипт буде керувати грою в цілому, а другий – займатися генеруванням лабіринту.

Всі рядки в GameController замінено наступним кодом:

```
using System;
using UnityEngine;

[RequireComponent(typeof(MazeConstructor))]           // 1

public class GameController : MonoBehaviour
{
    private MazeConstructor generator;

    void Start()
    {
        generator = GetComponent<MazeConstructor>();    // 2
    }
}
```

Згідно номеру коментаря в коді:

1. Атрибут `RequireComponent` забезпечує додавання компонента `MazeConstructor` при додаванні цього скрипту до `GameObject`.
2. Приватна змінна зберігає посилання, що повертається `GetComponent()`.

`MazeConstructor` включає в себе наступний код:

```
using UnityEngine;

public class MazeConstructor : MonoBehaviour
{
    //1
    public bool showDebug;

    [SerializeField] private Material mazeMat1;
    [SerializeField] private Material mazeMat2;
    [SerializeField] private Material startMat;
    [SerializeField] private Material treasureMat;

    //2
    public int[,] data
    {
        get; private set;
    }

    //3
    void Awake()
    {
        data = new int[,]
        {
            {1, 1, 1},
            {1, 0, 1},
            {1, 1, 1}
        }
    }
}
```

```

    };
}

public void GenerateNewMaze(int sizeRows, int sizeCols)
{
    // заглушка
}
}

```

Згідно номеру коментаря в коді:

1. Перелічені поля доступні в Inspector. ShowDebug перемикає відображення налагодження, а різні посилання Material є матеріалами для генерації моделей. Атрибут SerializeField відображає поле в Inspector, навіть незважаючи на те, що модифікатор доступу змінної є private.

2. Далі йде опис властивості data. Оголошення модифікатору доступу робить його read-only за межами класу. Таким чином, дані лабіринту неможливо буде змінювати ззовні.

3. Остання частина коду знаходиться в Awake(). Функція ініціалізує data з масивом 3 x 3 з одиниць, що оточують нуль. 1 означає стіну, а 0 – порожній простір, тобто сітка за замовчуванням виглядає як оточена стіною кімната.

Для відображення даних лабіринту і перевірки того, як він виглядає, в MazeConstructor додано наступний метод:

```

void OnGUI()
{
    //1
    if (!showDebug)
    {
        return;
    }

    //2
    int[,] maze = data;
    int rMax = maze.GetUpperBound(0);
    int cMax = maze.GetUpperBound(1);

    string msg = "";

    //3
    for (int i = rMax; i >= 0; i--)
    {
        for (int j = 0; j <= cMax; j++)
        {
            if (maze[i, j] == 0)

```

```

        {
            msg += "....";
        }
        else
        {
            msg += "==" ;
        }
    }
    msg += "\n";
}

//4
GUI.Label(new Rect(20, 20, 500, 500), msg);
}

```

Розглянемо кожен з прокоментованих розділів:

1. Код перевіряє, чи включено відображення налаштування.
2. Ініціалізація кількох локальних змінних: локальна копія збереженого лабіринту, максимальний рядок і стовпець.
3. Два вкладених циклу проходять по рядках і стовпцях двовірного масиву. Для кожного рядка або стовпця масиву код перевіряє збережене значення і додає "...." або "==" в залежності від того, чи дорівнює значення нулю. Також після проходження по всіх стовпцях в рядку код додає новий рядок.
4. Нарешті, GUI.Label() виводить створюваний рядок. У проєкті за замовчуванням використовується 3D система GUI виведення даних гравцеві, але 2D система простіше для створення швидких налагоджувальних повідомлень.

При включеному атрибуті ShowDebug для компонента MazeConstructor, якщо натиснути Play, на екрані відобразяться збережені дані лабіринту, які є лабіринтом за замовчуванням (Рис 3.4).



Рис. 3.4. Збережені дані лабіринту за замовчуванням

3.4 Генерування даних лабіринту

В кінці методу Start() скрипта GameController фігурує наступний рядок, що буде викликати метод GenerateNewMaze():

```
generator.GenerateNewMaze(13, 15);
```

Числа 13 і 15 – це параметри методу, що визначають розміри лабіринту – кількість рядків і стовпців сітки. В даному випадку параметри задані літералами. Потенційно можна кастомізувати або рандомізувати визначення параметрів.

Скрипт MazeDataGenerator містить однойменний клас, який інкапсулює логіку генерування даних, і буде використовуватися в MazeConstructor.

Відкрито новий скрипт і замінено все на наступний код:

```
using System.Collections.Generic;
using UnityEngine;

public class MazeDataGenerator
{
    public float placementThreshold;

    public MazeDataGenerator()
    {
        placementThreshold = .1f; // 1
    }

    public int[,] FromDimensions(int sizeRows, int sizeCols) // 2
    {
        int[,] maze = new int[sizeRows, sizeCols];
        return maze;
    }
}
```

Даний клас не буде наслідувати MonoBehaviour. Він не буде використовуватися безпосередньо як компонент, а використання у додатку обмежено MazeConstructor-ом. Тому функціонал MonoBehaviour тут не потрібен.

PlacementThreshold буде використовуватися алгоритмом генерування даних для визначення того, чи є порожнім простір. У конструкторі класу цієї змінної призначається значення за замовчуванням, але вона позначена як public, щоб інший код міг керувати налаштуванням лабіринту, що генерується.

В MazeConstructor було додано кілька розділів коду, щоб він міг викликати метод FromDimensions. На початку класу оголошена приватна змінна для зберігання генератора даних:

```
private MazeDataGenerator dataGenerator;
```

Надалі створюється екземпляр змінної в методі Awake(), зберігши генератор в нову змінну додаванням наступного рядка вгорі методу Awake().

```
dataGenerator = new MazeDataGenerator();
```

Після створення змінної проводиться виклик FromDimensions() в GenerateNewMaze(), передаючи розмір сітки і зберігаючи отримані дані. В GenerateNewMaze() заглушка замінюється наступним кодом:

```
if (sizeRows % 2 == 0 && sizeCols % 2 == 0)
{
    Debug.LogError("Odd numbers work better for dungeon size.");
}

data = dataGenerator.FromDimensions(sizeRows, sizeCols);
```

У вищенаведеному коді додано попередження про те, що краще використовувати для розмірів непарні числа, тому що згенерований лабіринт буде оточений стінами.

При запуску гри можна побачити порожні дані лабіринту з правильними розмірами (рис 3.5).

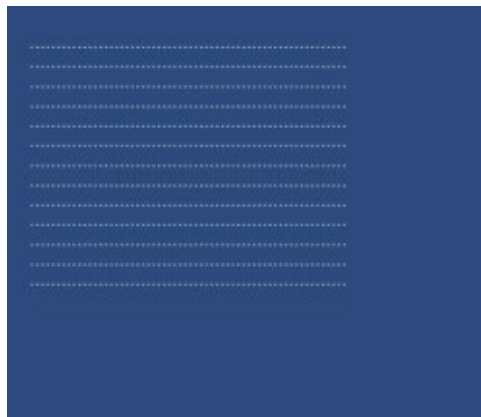
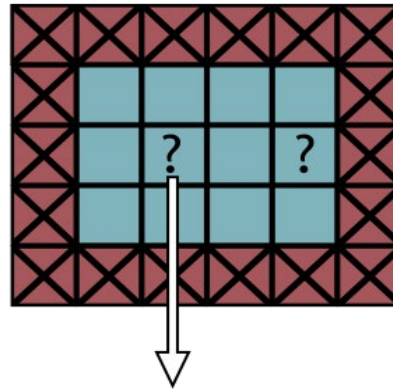


Рис. 3.5. Тестові дані лабіринту

На основі процедур збереження і відображення даних лабіринту в методі FromDimensions() реалізовано алгоритм генерації лабіринту (рис. 3.6).



1. Decide if there should be a wall here.
2. Pick an adjacent space to get a wall.

Рис. 3.6. Ілюстрація алгоритму

Описаний вище алгоритм обходить кожну другу клітинку в сітці (не кожну клітинку), розташовуючи стіну і вибираючи сусідній простір для блокування. Програмований у проєкті алгоритм злегка відрізняється від проілюстрованого, він також вирішує, чи потрібно пропускати простір, що може призводити до виникнення в лабіринті відкритих просторів. Оскільки алгоритм не повинен зберігати багато інформації або знати багато про іншу частину лабіринту, наприклад, про точки розгалуження, за якими потрібно пройти, то код можна дещо спростити.

Для реалізації цього алгоритму генерації лабіринту в FromDimensions() з MazeDataGenerator було додано наступний код:

```
int rMax = maze.GetUpperBound(0);
int cMax = maze.GetUpperBound(1);

for (int i = 0; i <= rMax; i++)
{
    for (int j = 0; j <= cMax; j++)
    {
        //1
        if (i == 0 || j == 0 || i == rMax || j == cMax)
        {
            maze[i, j] = 1;
        }

        //2
        else if (i % 2 == 0 && j % 2 == 0)
        {
            if (Random.value > placementThreshold)
```

```

{
    //3
    maze[i, j] = 1;

    int a = Random.value < .5 ? 0 : (Random.value < .5 ? -
1 : 1);
    int b = a != 0 ? 0 : (Random.value < .5 ? -1 : 1);
    maze[i+a, j+b] = 1;
}
}
}
}
}

```

Код отримує кордони 2D-масиву, а потім обходить його:

1. Для кожної сітки алгоритм спочатку перевіряє, чи виходить поточна клітинка за межі сітки (тобто знаходиться якийсь з індексів на кордоні масиву). Якщо це так, то він ставить стіну, привласнюючи значення 1.

2. Далі код перевіряє, чи діляться координати на 2 без залишку, щоб виконувати дії в кожній клітині. Також присутня додаткова перевірка на описане вище значення `placementThreshold` для пропуску випадковим чином цієї комірки та продовження обходу масиву.

3. Нарешті, код привласнює значення 1 поточній клітині випадково обраній сусідній клітині. Код використовує кілька операцій для додавання до індексу масиву 0, 1 або -1, отримуючи таким чином індекс сусідньої комірки.

При повторному відображенні даних лабіринту можна переконатися в успішності виконання процедури (рис. 3.7).

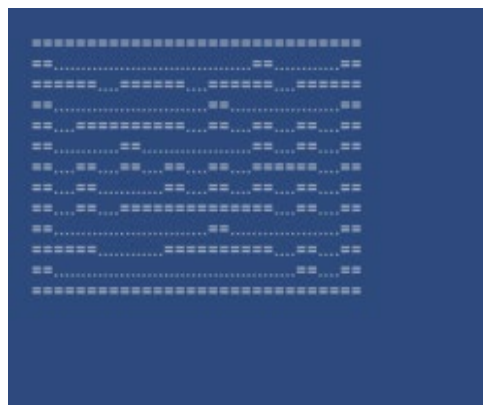


Рис. 3.7. Успішне виконання процедури

3.5 Генерація мешу лабіринту

Після генерування всіх даних лабіринту можна на підставі цих даних побудувати меш (в комп'ютерній графіці – набір вершин і багатокутників, що визначають форму тривимірного об'єкта).

Для цієї мети створено новий скрипт MazeMeshGenerator. Так само, як MazeDataGenerator інкапсулював логіку генерування лабіринту, MazeMeshGenerator буде містити логіку генерування мешу і буде використовуватися MazeConstructor для виконання цього етапу генерування лабіринту.

Для демонстрації роботи і опису взаємодії компонентів було створено текстурований чотирикутник, а потім поетапно модифіковано код для генерування всього лабіринту.

З початку потрібно прив'язати матеріали, які будуть застосовуватися до згенерованого мешу. Вибрати у вікні Project папку Graphics, вибрати у вікні Hierarchy Controller, щоб відобразити в Inspector компонент Maze Constructor.

Перетягнути матеріали з папки Graphics в слоти матеріалів Maze Constructor. Використовуючи floor-mat для Material 1 і wall-mat для Material 2, а start і treasure повинні бути перетягнуті у відповідні слоти.

Завершивши всі необхідні зміни в редакторі Unity в скрипт MazeMeshGenerator вноситься наступний код:

```
using System.Collections.Generic;
using UnityEngine;

public class MazeMeshGenerator
{
    // generator params
    public float width;      // how wide are hallways
    public float height;     // how tall are hallways

    public MazeMeshGenerator()
    {
        width = 3.75f;
        height = 3.5f;
    }

    public Mesh FromData(int[,] data)
    {
        Mesh maze = new Mesh();
```



```

//1
List<Vector3> newVertices = new List<Vector3>();
List<Vector2> newUVs = new List<Vector2>();
List<int> newTriangles = new List<int>();

// corners of quad
Vector3 vert1 = new Vector3(-.5f, -.5f, 0);
Vector3 vert2 = new Vector3(-.5f, .5f, 0);
Vector3 vert3 = new Vector3(.5f, .5f, 0);
Vector3 vert4 = new Vector3(.5f, -.5f, 0);

//2
newVertices.Add(vert1);
newVertices.Add(vert2);
newVertices.Add(vert3);
newVertices.Add(vert4);

//3
newUVs.Add(new Vector2(1, 0));
newUVs.Add(new Vector2(1, 1));
newUVs.Add(new Vector2(0, 1));
newUVs.Add(new Vector2(0, 0));

//4
newTriangles.Add(2);
newTriangles.Add(1);
newTriangles.Add(0);

//5
newTriangles.Add(3);
newTriangles.Add(2);
newTriangles.Add(0);

maze.vertices = newVertices.ToArray();
maze.uv = newUVs.ToArray();
maze.triangles = newTriangles.ToArray();

return maze;
}
}

```

Два поля у верхній частині класу, `width` і `height`, аналогічні `placementThreshold` з `MazeDataGenerator` – це значення, які в конструкторі задаються за замовчанням і використовуються кодом генерування мешу.

Основна частина коду знаходиться всередині `FromData()` – це метод, який `MazeConstructor` викликає для генерування мешу. В даному випадку цей код просто створює єдиний чотирикутник для демонстрації своєї роботи.

На ілюстрації (рис. 3.8) показано з чого створений чотирикутник:

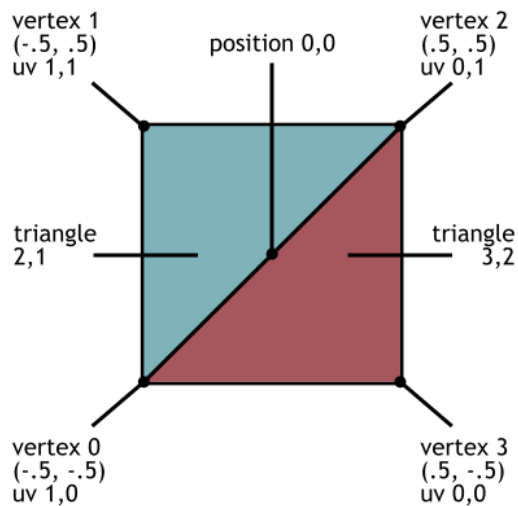


Рис. 3.8. Приклад генерації мешу

Загалом код визначає, що:

- меш складається з трьох списків: вершин, координат UV і трикутників;
- список вершин зберігає позицію кожної вершини;
- перераховані координати UV відповідають вершинам в цьому списку;
- трикутники є індексами в списку вершин (тобто "цей трикутник складається з вершин 0, 1 і 2");
- створюються два трикутника; чотирикутник складається з двох трикутників.

MazeConstructor повинен створити екземпляр MazeMeshGenerator, а потім викликати метод генерування мешу. Також він повинен відображати меш. Наступним кодом додано приватне поле для зберігання генератору мешу:

```
private MazeMeshGenerator meshGenerator;
```

Створено його екземпляр в Awake(), зберігши генератор мешу в новому полі додаванням наступного рядка в верхній частині методу Awake():

```
meshGenerator = new MazeMeshGenerator();
```

Код методу DisplayMaze () наведено далі:

```
private void DisplayMaze()
{
    GameObject go = new GameObject();
```

```

go.transform.position = Vector3.zero;
go.name = "Procedural Maze";
go.tag = "Generated";

MeshFilter mf = go.AddComponent<MeshFilter>();
mf.mesh = meshGenerator.FromData(data);

MeshCollider mc = go.AddComponent<MeshCollider>();
mc.sharedMesh = mf.mesh;

MeshRenderer mr = go.AddComponent<MeshRenderer>();
mr.materials = new Material[2] {mazeMat1, mazeMat2};
}

```

Сам по собі меш – це деякі дані. Він невидимий, поки не призначений певному об'єкту (якщо конкретніше, то MeshFilter об'єкта) в сцені. Тому DisplayMaze() не тільки викликає MazeMeshGenerator.FromData(), але і вставляє цей виклик посередині створення екземпляра нового GameObject, задаючи тег Generated, додаючи MeshFilter і згенерований меш, додаючи MeshCollider для колізій з мешем, і, нарешті, додаючи MeshRenderer і матеріали.

При такій компіляції MazeMeshGeneratorі під час запуску додатку можна отримати побудований, повністю затекстурований чотирикутник (рис. 3.9).

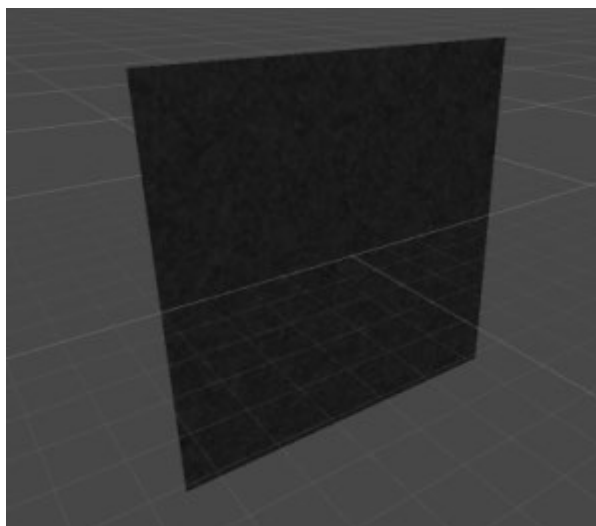


Рис. 3.9. Генерація чотирикутника у грі

Для роботи з лабіринтом в код FromData() треба внести значні зміни:

```

public Mesh FromData(int[, ] data)
{
    Mesh maze = new Mesh();
    //3
    List<Vector3> newVertices = new List<Vector3>();
    List<Vector2> newUVs = new List<Vector2>();

    maze.subMeshCount = 2;
    List<int> floorTriangles = new List<int>();
    List<int> wallTriangles = new List<int>();

    int rMax = data.GetUpperBound(0);
    int cMax = data.GetUpperBound(1);
    float halfH = height * .5f;

    //4
    for (int i = 0; i <= rMax; i++)
    {
        for (int j = 0; j <= cMax; j++)
        {
            if (data[i, j] != 1)
            {
                // floor
                AddQuad(Matrix4x4.TRS(
                    new Vector3(j * width, 0, i * width),
                    Quaternion.LookRotation(Vector3.up),
                    new Vector3(width, width, 1)
                ), ref newVertices, ref newUVs, ref floorTriangles);

                // ceiling
                AddQuad(Matrix4x4.TRS(
                    new Vector3(j * width, height, i * width),
                    Quaternion.LookRotation(Vector3.down),
                    new Vector3(width, width, 1)
                ), ref newVertices, ref newUVs, ref floorTriangles);

                // walls on sides next to blocked grid cells

                if (i - 1 < 0 || data[i-1, j] == 1)
                {
                    AddQuad(Matrix4x4.TRS(
                        new Vector3(j * width, halfH, (i-.5f) * width),
                        Quaternion.LookRotation(Vector3.forward),
                        new Vector3(width, height, 1)
                    ), ref newVertices, ref newUVs, ref wallTriangles);
                }

                if (j + 1 > cMax || data[i, j+1] == 1)
                {
                    AddQuad(Matrix4x4.TRS(
                        new Vector3((j+.5f) * width, halfH, i * width),
                        Quaternion.LookRotation(Vector3.left),
                        new Vector3(width, height, 1)
                    ), ref newVertices, ref newUVs, ref wallTriangles);
                }
            }
        }
    }
}

```

```

        if (j - 1 < 0 || data[i, j-1] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3((j-.5f) * width, halfH, i * width),
                Quaternion.LookRotation(Vector3.right),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }

        if (i + 1 > rMax || data[i+1, j] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3(j * width, halfH, (i+.5f) * width),
                Quaternion.LookRotation(Vector3.back),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }
    }
}

maze.vertices = newVertices.ToArray();
maze.uv = newUVs.ToArray();

maze.SetTriangles(floorTriangles.ToArray(), 0);
maze.SetTriangles(wallTriangles.ToArray(), 1);

maze.RecalculateNormals();

return maze;
}
//1, 2
private void AddQuad(Matrix4x4 matrix, ref List<Vector3> newVertices,
    ref List<Vector2> newUVs, ref List<int> newTriangles)
{
    int index = newVertices.Count;

    // corners before transforming
    Vector3 vert1 = new Vector3(-.5f, -.5f, 0);
    Vector3 vert2 = new Vector3(-.5f, .5f, 0);
    Vector3 vert3 = new Vector3(.5f, .5f, 0);
    Vector3 vert4 = new Vector3(.5f, -.5f, 0);

    newVertices.Add(matrix.MultiplyPoint3x4(vert1));
    newVertices.Add(matrix.MultiplyPoint3x4(vert2));
    newVertices.Add(matrix.MultiplyPoint3x4(vert3));
    newVertices.Add(matrix.MultiplyPoint3x4(vert4));

    newUVs.Add(new Vector2(1, 0));
    newUVs.Add(new Vector2(1, 1));
    newUVs.Add(new Vector2(0, 1));
    newUVs.Add(new Vector2(0, 0));

    newTriangles.Add(index+2);
    newTriangles.Add(index+1);

```

```

newTriangles.Add(index);

newTriangles.Add(index+3);
newTriangles.Add(index+2);
newTriangles.Add(index);
}

```

Код генерування чотирикутника переміщений в окремий метод `AddQuad()` для його повторного виклику для підлоги, стелі і стін кожної клітинки сітки.

1. Останні три параметри `AddQuad()` – це список вершин, UV і трикутників. Перший рядок методу отримує індекс, з якого потрібно починати. При додаванні нових чотирикутників індекс буде збільшуватися.

2. Перший параметр `AddQuad()` – це матриця перетворень. По суті, положення / поворот / масштаб може зберігатися у вигляді матриці, а потім застосовуватися до вершин. Саме це робить виклик `MultiplyPoint3x4()`. Таким чином, код генерування чотирикутника можна використовувати для підлог, стель, стін та ін. Достатньо лише змінювати використовувану матрицю перетворень.

3. Списки для вершин UV і трикутників створюються в верхній частині `FromData()`. Об'єкт `MeshUnity` може мати безліч підмешів з різними матеріалами на кожному, тобто кожен список трикутників є окремим підмешем. Ми оголошуємо два підмеша, щоб можна було призначити різні матеріали підлоги та стін.

4. Після цього виконується прохід по 2D-масиву і створення чотирикутників для підлоги, стелі і стін в кожній комірці сітки. Кожній комірці потрібна підлога і стеля, крім того, виконуються перевірки сусідніх комірок на необхідність стін. `AddQuad()` викликається кілька разів, але кожен раз з іншою матрицею перетворень і різними списками трикутників, використовуваними для підлоги і стін.

При запуску проєкту можна побачити, як генерується весь меш згідно даних налагодження (рис. 3.10).

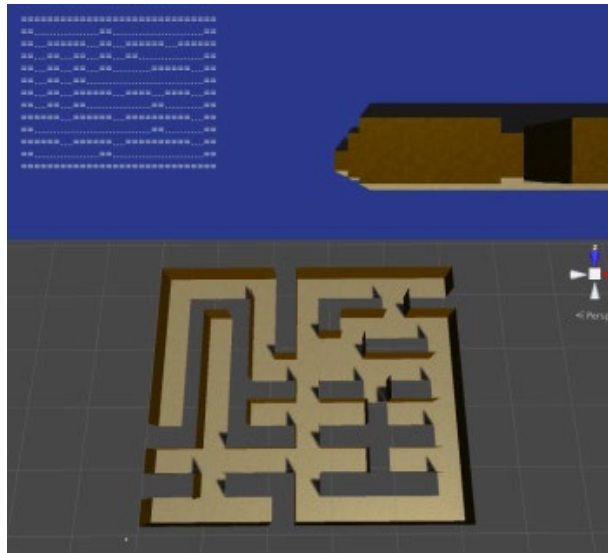


Рис. 3.10. Результат генерації лабіринту

3.6 Інтерактивні аспекти гри

У проєкті знаходяться два скрипти, сцена з гравцем і UI, а також вся графіка для гри з лабіринтом. Скрипт `FpsMovement` – це одностороння версія контролера персонажа, а `TriggerEventRouter` – це допоміжний код, для зручності роботи з тригерами гри.

У сцені відповідно налаштований гравець, у якого є компонент `FpsMovement` і до камери приєднане направлене джерело світла. Крім того, у вікні `Lighting Settings` відключені скайбокси і навколишнє освітлення. Нарешті, в сцені є полотно UI з мітками для рахунку і часу.

У `MazeConstructor` було додано наступні властивості для зберігання розмірів і координат:

```
public float hallWidth
{
    get; private set;
}
public float hallHeight
{
    get; private set;
}

public int startRow
{
    get; private set;
}
```

```

public int startCol
{
    get; private set;
}

public int goalRow
{
    get; private set;
}
public int goalCol
{
    get; private set;
}

```

Для подальшої реалізації ігрового процесу було створено два методи. Перший – це `DisposeOldMaze()`, що видаляє існуючий лабіринт. Метод знаходить всі об'єкти з тегом `Generated` і знищує їх.

```

public void DisposeOldMaze()
{
    GameObject[] objects = GameObject.FindGameObjectsWithTag("Generated");
    foreach (GameObject go in objects) {
        Destroy(go);
    }
}

```

Другий – метод `FindStartPosition()`. Код починає обхід з 0,0 і проходить по всіх даних лабіринту, поки не знаходить відкритий простір. Потім ці координати зберігаються як початкова позиція лабіринту.

```

private void FindStartPosition()
{
    int[,] maze = data;
    int rMax = maze.GetUpperBound(0);
    int cMax = maze.GetUpperBound(1);

    for (int i = 0; i <= rMax; i++)
    {
        for (int j = 0; j <= cMax; j++)
        {
            if (maze[i, j] == 0)
            {
                startRow = i;
                startCol = j;
                return;
            }
        }
    }
}

```



```
}
```

Аналогічно, FindGoalPosition() по суті робить те ж саме, тільки починає з максимальних значень і виконує зворотний відлік.

```
private void FindGoalPosition()
{
    int[,] maze = data;
    int rMax = maze.GetUpperBound(0);
    int cMax = maze.GetUpperBound(1);

    // loop top to bottom, right to left
    for (int i = rMax; i >= 0; i--)
    {
        for (int j = cMax; j >= 0; j--)
        {
            if (maze[i, j] == 0)
            {
                goalRow = i;
                goalCol = j;
                return;
            }
        }
    }
}
```

PlaceStartTrigger() і PlaceGoalTrigger() розміщують об'єкти в сцені в позиціях початку і цілі. Їх колайдер є тригером. До них застосовується відповідний матеріал, а потім додається TriggerEventRouter. Цей компонент отримує функцію обробки події, яка викликається, коли щось входить в обсяг тригера.

```
private void PlaceStartTrigger(TriggerEventHandler callback)
{
    GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);
    go.transform.position = new Vector3(startCol * hallWidth, .5f, startRow
* hallWidth);
    go.name = "Start Trigger";
    go.tag = "Generated";

    go.GetComponent<BoxCollider>().isTrigger = true;
    go.GetComponent<MeshRenderer>().sharedMaterial = startMat;

    TriggerEventRouter tc = go.AddComponent<TriggerEventRouter>();
    tc.callback = callback;
}
```

```
private void PlaceGoalTrigger(TriggerEventHandler callback)
```

```

{
    GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);
    go.transform.position = new Vector3(goalCol * hallWidth, .5f, goalRow *
hallWidth);
    go.name = "Treasure";
    go.tag = "Generated";

    go.GetComponent<BoxCollider>().isTrigger = true;
    go.GetComponent<MeshRenderer>().sharedMaterial = treasureMat;

    TriggerEventRouter tc = go.AddComponent<TriggerEventRouter>();
    tc.callback = callback;
}

```

Метод GenerateNewMaze() модифіковано наступним чином:

```

public void GenerateNewMaze(int sizeRows, int sizeCols,
    TriggerEventHandler startCallback=null, TriggerEventHandler goalCallback
=null)
{
    if (sizeRows % 2 == 0 && sizeCols % 2 == 0)
    {
        Debug.LogError("Odd numbers work better for dungeon size.");
    }

    DisposeOldMaze();

    data = dataGenerator.FromDimensions(sizeRows, sizeCols);

    FindStartPosition();
    FindGoalPosition();

    // store values used to generate this mesh
    hallWidth = meshGenerator.width;
    hallHeight = meshGenerator.height;

    DisplayMaze();

    PlaceStartTrigger(startCallback);
    PlaceGoalTrigger(goalCallback);
}

```

Переписаний GenerateNewMaze() викликає нові методи ігрового процесу для таких операцій, як видалення старого мешу і розташування тригерів.

Код в GameController був замінений наступним чином:

```

using System;
using UnityEngine;

```

```

using UnityEngine.UI;

[RequireComponent(typeof(MazeConstructor))]

public class GameController : MonoBehaviour
{
    //1
    [SerializeField] private FpsMovement player;
    [SerializeField] private Text timeLabel;
    [SerializeField] private Text scoreLabel;

    private MazeConstructor generator;

    //2
    private DateTime startTime;
    private int timeLimit;
    private int reduceLimitBy;

    private int score;
    private bool goalReached;

    //3
    void Start() {
        generator = GetComponent<MazeConstructor>();
        StartNewGame();
    }

    //4
    private void StartNewGame()
    {
        timeLimit = 80;
        reduceLimitBy = 5;
        startTime = DateTime.Now;

        score = 0;
        scoreLabel.text = score.ToString();

        StartNewMaze();
    }

    //5
    private void StartNewMaze()
    {
        generator.GenerateNewMaze(13, 15, OnStartTrigger, OnGoalTrigger);

        float x = generator.startCol * generator.hallWidth;
        float y = 1;
        float z = generator.startRow * generator.hallWidth;
    }
}

```

```

    player.transform.position = new Vector3(x, y, z);

    goalReached = false;
    player.enabled = true;

    // restart timer
    timeLimit -= reduceLimitBy;
    startTime = DateTime.Now;
}

//6
void Update()
{
    if (!player.enabled)
    {
        return;
    }

    int timeUsed = (int)(DateTime.Now - startTime).TotalSeconds;
    int timeLeft = timeLimit - timeUsed;

    if (timeLeft > 0)
    {
        timeLabel.text = timeLeft.ToString();
    }
    else
    {
        timeLabel.text = "TIME UP";
        player.enabled = false;

        Invoke("StartNewGame", 4);
    }
}

//7
private void OnGoalTrigger(GameObject trigger, GameObject other)
{
    Debug.Log("Goal!");
    goalReached = true;

    score += 1;
    scoreLabel.text = score.ToString();

    Destroy(trigger);
}

private void OnStartTrigger(GameObject trigger, GameObject other)
{

```

```

        if (goalReached)
        {
            Debug.Log("Finish!");
            player.enabled = false;

            Invoke("StartNewMaze", 4);
        }
    }
}

```

1. Додано серіалізовані поля для об'єктів в сцені.
2. Додано декілька приватних змінних для відстеження таймера і балів гри, а також того, чи досягнута мета в лабіринті.
3. Start() використовує новий метод (StartNewGame()), який не просто викликає GenerateNewMaze().
4. StartNewGame() використовується для запуску всієї гри спочатку, а не для перемикання рівнів всередині гри. Таймеру присвоюються початкові значення, рахунок скидається, після чого створюється лабіринт.
5. StartNewMaze() переходить до нового рівня, не перезапускаючи заново всю гру. Крім створення нового лабіринту, цей метод розташовує гравця в початковій точці, скидає мету і знижує ліміт часу.
6. Update() перевіряє, чи активний гравець, а потім оновлює час, що залишився на проходження рівня. Після завершення часу гравець деактивується і починається нова гра.
7. OnGoalTrigger() і OnStartTrigger() – функції обробки подій, що передаються TriggerEventRouter в MazeConstructor. OnGoalTrigger() записує, що ціль була знайдена, а потім збільшує кількість очок. OnStartTrigger() перевіряє, чи знайдена ціль, і якщо це так, то деактивує гравця і запускає новий лабіринт.

Остаточний результат розробки інтерактивної частини ігрового проєкту проілюстровано на прикладі одного з етапів гри (рис. 3.11)



Рис. 3.11. Скріншот ігрового процесу (знайдено скриню зі скарбом)

3.7 Висновки до третього розділу

Проведено апробацію запропонованих рішень у ході застосування методів генерації лабіринтів. Проведено загальний огляд особливостей реалізації у середовищі Unity 3D, виконана програмна реалізація генерації лабіринту у середовищі Unity 3D користуючись інтегрованим середовищем розробки Visual Studio та мовою програмування C#. На засадах вищезазначених інструментів створено ігровий додаток "Розкрадач гробниць". Описано особливості ігрового процесу та логіки обробки даних.

У планах подальшого розвитку додатку можлива розробка:

- дизайну об'єктів (кожен рівень буде відрізнятися місцевістю проходження та виглядом скрині зі скарбом);
- нових об'єктів (вороги, двері, декор);
- ігрових звуків;
- вікна налаштувань;
- мережевої версії з рейтингами і статистикою гравців.

Код програми легкий, його можна редагувати, розширювати і модернізувати під потреби розробника або користувача.

ВИСНОВКИ

У рамках випускної кваліфікаційної роботи було виконано загальний огляд і аналіз сучасного стану розглянутої проблеми. Зокрема приділено увагу особливостям процесу розробки ігрових додатків, загальному огляду та класифікації видів лабіринтів та огляду сфери використання лабіринтів у формуванні гравального простору.

Проведено дослідження існуючих методів генерації лабіринтів, серед яких були розглянуті алгоритми двійкового дерева, Прима, Крускала, Олдоса-Бродера, Вілсона, Еллера, Sidewinder. Основну увагу приділено порівнянню даних алгоритмів, за результатами якого було запропоновано модифікований метод генерації лабіринтів, основою якого є алгоритм Вілсона. Для алгоритму редагування шляхів в лабіринті методи були протестовані і досліджені на ефективність, виявлені їх недоліки.

Здійснено апробацію запропонованих рішень у ході застосування методів генерації лабіринтів. Проведено загальний огляд особливостей реалізації у середовищі Unity 3D. Виконано програмну реалізацію генерації лабіринту у середовищі Unity 3D. Розроблено ігровий додаток " Розкрадач гробниць" і описано особливості ігрового процесу та логіки обробки даних.

Гра знаходиться на стадії доробки, в планах у подальшому розвитку додатку, планується розробка нових об'єктів (вороги, двері, декор) та вікна налаштувань.

Код програми легкий в розширенні і модернізації під потреби розробника або користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bethke, E. Game development and production [Text] / E. Bethke. – Wordware Publishing, Inc, 2003.– 415 с. – ISBN 1-55622-951-8.
2. Создание игр. Один в поле воин! Игрострой GCUP. URL: <http://gcup.ru>.
3. Video games starting to get serious. [Electronic resource] / Steve Barberich. URL: http://www.gfzette.net/stries/083107/businew11739_32356.html.
4. App2Top. Этапы разработки игры для мобильных платформ. URL: <https://app2top.ru/columns/e-tapy-razrabotki-igry-dlya-mobil-ny-h-p-40118.html>.
5. David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 296–303. ACM, 1996.
6. Habr. Алгоритм Дейкстры. Поиск оптимальных маршрутов на графе. URL: <https://habr.com/ru/post/111361/>.
7. Lee, C.Y., An Algorithm for Path Connections and Its Applications, IRE Transactions on Electronic Computers, vol. EC-10, number 2, 1961 – P. 364-365.
8. М Tim Jones. Artificial Intelligence: A Systems Approach: A Systems Approach. Jones & Bartlett Learning, 2015.
9. Maze generations: Algorithms and Visualizations. URL: <https://medium.com/analytics-vidhya/maze-generations-algorithms-and-visualizations-9f5e88a3ae37>.
10. Studfiles. Модификация волнового алгоритма. URL: <https://studfile.net/preview/1382785/page:25/>.
11. Studfiles. Поиск в ширину. URL: <https://studfile.net/preview/1399243/>.
12. Suvitruf's Blog: Gamedev suffering. Обход препятствий: волновой алгоритм (Алгоритм Ли). URL: <https://suvitruf.ru/2012/05/13/1176/volnovoj-algorithm-algorithm-li/>
13. The Buckblog. Maze Generation: Binary Tree algorithm URL: <http://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm.html>.
14. The Buckblog. Maze Generation: Kruskal's Algorithm. URL: <http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm.html>.

15. The Buckblog. Maze Generation: Sidewinder algorithm URL: <http://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm.html>
16. The Buckblog. Maze Generation: Wilson's algorithm URL: <http://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm.html> .
17. The Buckblog. Maze Generation: Aldous-Broder algorithm. URL: <http://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm.html> .
18. Will Goldstone. «Unity Game Development Essentials». October 2009. 316p.
19. Басараб М.А., Домрачева А.Б., Купляков В.М. Алгоритмы решения задачи быстрого поиска пути. Инженерный журнал: наука и инновации, 2013, в ып. № 11. URL:<http://engjournal.ru/catalog/it/hidden/1054.html>.
20. Визуальный язык ДРАКОН.. URL: <https://drakon.su/algorithmy/a-star>.
21. Герасимов В.Н., Михайлов Б.Б. Решение задачи управления движением мобильного робота при наличии динамических препятствий. *Вестник МГТУ им. Н.Э. Баумана. Приборостроение. Спецвыпуск "Робототехнические системы"*. 2012. № 6. С. 83-92.
22. Германн, Керн. Лабиринт: основные принципы, гипотезы, интерпретации. *Лабиринты мира*. СПб.: Азбука-классика, 2007. С. 7-33.
23. ДСТУ 3008:2015. Інформація та документація. Звіти у сфері науки і техніки. Структура і правила оформлювання. – Чинний від 22.06.2015. – Київ: ДП «УкрНДНЦ», 2016. – 31 с.
24. Краснов Е.С. Методика оценки алгоритмов поиска пути в лабиринте *Известия тульского государственного университета. Технические науки*, выпуск 11, часть 2. С. 179– 187.
25. Кристиан Нейгел. «С# 5.0 и платформа .NET 4.5 для профессионалов – Professional C# 5.0 and .NET 4.5.». М.: «Диалектика», 2013. 1440 с. – ISBN 978-5-8459-1850-5
26. Лабиринты: классификация, генерирование, поиск решений URL: https://ai-news.ru/2019/03/labirinty_klassifikaciya_generirovanie_poisk_reshenij.
27. Сахнов К. Семь этапов создания игры: от концепта до релиза. URL:

<https://habrahabr.ru/company/miip/blog/308286/> .

28. Соловьёв И. Выделение контуров зданий и распознавание служебных символов для трехмерной реконструкции объектов.

URL:<https://www.graphicon.ru/html/2013/papers/290-293.pdf> .

29. Федотова, Д. Э., Семенов Ю. Д., Чижик К. Н. CASE-технологии: Практикум [Текст] / М. : "Горячая Линия – Телеком", 2005. 160 с.

30. Хокинг Дж. «Unity в действии. Мультиплатформенная разработка на C#» – СПб.: Питер, 2016. – 336 с.

31. Штовба С. Д. Муравьиные алгоритмы. Exponenta Pro. 2003. №4. С. 70-75.

32. Жанры компьютерных игр (общая схема). Компьютерные игры как искусство. URL: <https://gamesisart.ru/TableJanr.html> .

ДОДАТКИ

Додаток А Код GameController.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(MazeConstructor))]

public class GameController : MonoBehaviour
{
    [SerializeField] private FpsMovement player;
    [SerializeField] private Text timeLabel;
    [SerializeField] private Text scoreLabel;

    private MazeConstructor generator;

    private DateTime startTime;
    private int timeLimit;
    private int reduceLimitBy;

    private int score;
    private bool goalReached;

    // Use this for initialization
    void Start() {
        generator = GetComponent<MazeConstructor>();
        StartNewGame();
    }

    private void StartNewGame()
    {
        timeLimit = 80;
        reduceLimitBy = 5;
        startTime = DateTime.Now;

        score = 0;
        scoreLabel.text = score.ToString();

        StartNewMaze();
    }

    private void StartNewMaze()
    {
        generator.GenerateNewMaze(13, 15, OnStartTrigger, OnGoalTrigger);

        float x = generator.startCol * generator.hallWidth;
        float y = 1;
        float z = generator.startRow * generator.hallWidth;
        player.transform.position = new Vector3(x, y, z);

        goalReached = false;
    }
}

```

```

        player.enabled = true;

        // restart timer
        timeLimit -= reduceLimitBy;
        startTime = DateTime.Now;
    }

    // Update is called once per frame
    void Update()
    {
        if (!player.enabled)
        {
            return;
        }

        int timeUsed = (int)(DateTime.Now - startTime).TotalSeconds;
        int timeLeft = timeLimit - timeUsed;

        if (timeLeft > 0)
        {
            timeLabel.text = timeLeft.ToString();
        }
        else
        {
            timeLabel.text = "TIME UP";
            player.enabled = false;

            Invoke("StartNewGame", 4);
        }
    }

    private void OnGoalTrigger(GameObject trigger, GameObject other)
    {
        Debug.Log("Goal!");
        goalReached = true;

        score += 1;
        scoreLabel.text = score.ToString();

        Destroy(trigger);
    }

    private void OnStartTrigger(GameObject trigger, GameObject other)
    {
        if (goalReached)
        {
            Debug.Log("Finish!");
            player.enabled = false;

            Invoke("StartNewMaze", 4);
        }
    }
}

```

Додаток В. Код MazeMeshGenerator.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MazeMeshGenerator
{
    // generator params
    public float width;      // how wide are hallways
    public float height;     // how tall are hallways

    public MazeMeshGenerator()
    {
        width = 3.75f;
        height = 3.5f;
    }

    public Mesh FromData(int[,] data)
    {
        Mesh maze = new Mesh();

        List<Vector3> newVertices = new List<Vector3>();
        List<Vector2> newUVs = new List<Vector2>();

        // multiple materials for floors and walls
        maze.subMeshCount = 2;
        List<int> floorTriangles = new List<int>();
        List<int> wallTriangles = new List<int>();

        int rMax = data.GetUpperBound(0);
        int cMax = data.GetUpperBound(1);
        float halfH = height * .5f;

        for (int i = 0; i <= rMax; i++)
        {
            for (int j = 0; j <= cMax; j++)
            {
                if (data[i, j] != 1)
                {
                    // floor
                    AddQuad(Matrix4x4.TRS(
                        new Vector3(j * width, 0, i * width),
                        Quaternion.LookRotation(Vector3.up),
                        new Vector3(width, width, 1)
                    ), ref newVertices, ref newUVs, ref floorTriangles);

                    // ceiling
                    AddQuad(Matrix4x4.TRS(
                        new Vector3(j * width, height, i * width),
                        Quaternion.LookRotation(Vector3.down),
                        new Vector3(width, width, 1)
                    ), ref newVertices, ref newUVs, ref floorTriangles);

                    // walls on sides next to blocked grid cells
                }
            }
        }
    }
}

```

```

        if (i - 1 < 0 || data[i-1, j] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3(j * width, halfH, (i-.5f) * width),
                Quaternion.LookRotation(Vector3.forward),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }

        if (j + 1 > cMax || data[i, j+1] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3((j+.5f) * width, halfH, i * width),
                Quaternion.LookRotation(Vector3.left),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }

        if (j - 1 < 0 || data[i, j-1] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3((j-.5f) * width, halfH, i * width),
                Quaternion.LookRotation(Vector3.right),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }

        if (i + 1 > rMax || data[i+1, j] == 1)
        {
            AddQuad(Matrix4x4.TRS(
                new Vector3(j * width, halfH, (i+.5f) * width),
                Quaternion.LookRotation(Vector3.back),
                new Vector3(width, height, 1)
            ), ref newVertices, ref newUVs, ref wallTriangles);
        }
    }
}

maze.vertices = newVertices.ToArray();
maze.uv = newUVs.ToArray();

maze.SetTriangles(floorTriangles.ToArray(), 0);
maze.SetTriangles(wallTriangles.ToArray(), 1);

maze.RecalculateNormals();

return maze;
}

private void AddQuad(Matrix4x4 matrix, ref List<Vector3> newVertices,
    ref List<Vector2> newUVs, ref List<int> newTriangles)
{
    int index = newVertices.Count;

```

```

// corners before transforming
Vector3 vert1 = new Vector3(-.5f, -.5f, 0);
Vector3 vert2 = new Vector3(-.5f, .5f, 0);
Vector3 vert3 = new Vector3(.5f, .5f, 0);
Vector3 vert4 = new Vector3(.5f, -.5f, 0);

newVertices.Add(matrix.MultiplyPoint3x4(vert1));
newVertices.Add(matrix.MultiplyPoint3x4(vert2));
newVertices.Add(matrix.MultiplyPoint3x4(vert3));
newVertices.Add(matrix.MultiplyPoint3x4(vert4));

newUVs.Add(new Vector2(1, 0));
newUVs.Add(new Vector2(1, 1));
newUVs.Add(new Vector2(0, 1));
newUVs.Add(new Vector2(0, 0));

newTriangles.Add(index+2);
newTriangles.Add(index+1);
newTriangles.Add(index);

newTriangles.Add(index+3);
newTriangles.Add(index+2);
newTriangles.Add(index);
    }
}

```