Міністерство освіти і науки України
Національний технічний університет
«Дніпропетровська політехніка»

Інститут електроенергетики
Факультет інформаційних технологій
Кафедра інформаційних технологій та комп'ютерної інженерії

ПОЯСНЮВАЛЬНА ЗАПИСКА
Кваліфікаційної роботи ступеня бакалавра

Студента Латкіна Дмитра Олексійовича

академічної групи 126-19-1
спеціальності 126 «Інформаційні системи та технології»
за освітньо-професійною програмою
«Інформаційні системи та технології»
на тему Створення гри з процедурною генерацією поля на основі A* алгоритму у середовищі Unity.

Керівники	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікавційної роботи	проф. Олевський В.І			
розділів:				

Рецензент	проф. Лактіонов І.С.			

Нормоконтролер	проф. Коротенко Г.М.			

Дніпро
2023

ЗАТВЕРДЖЕНО:
завідувач кафедри

інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ Гнатушенко В.В.
(підпис) (прізвище, ініціали)

«_____»_____2023 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня _____бакалавр _____
(бакалавра, спеціаліста, магістра)

студенту _____Латкін Д.О._____ **академічної групи** ____126-19-1____
(прізвище та ініціали) (шифр)

спеціальності _____126 « Інформаційні системи та технології »_____

за освітньою-професійною програмою _____

«Інформаційні системи та технології»_____

на тему ____Створення гри з процедурною генерацією поля на основі A* алгоритму у середовищі Unity._____

затверджену наказом ректора НТУ «Дніпровська політехніка» від 16.05.2023 р. № 350-с

Розділ	Зміст	Термін виконання
Розділ 1	Ознайомлення з матеріалами сайтів, літературою та виконання пошуку технологій і формування відповідних рішень	
Розділ 2 та 3	Проектування та реалізація проектних рішень	01.04.2023 – 20.06.2023

Завдання видано _____ _____Олевський В.І_____
(підпис керівника) (прізвище, ініціали)

Дата видачі _____01.02.2023 р_____

Дата подання до екзаменаційної комісії _____20.06.2023 р _____

Прийнято до виконання _____ _____Латкін Д.О._____
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка : 57с., 14 рис., 3 додатка, 18 джерел.

Об’єкт розробки: процедурно згенероване поле в середовищі Unity з використанням мови програмування C#.

Мета роботи: вдосконалення методів процедурної генерації з використанням алгоритму пошуку A*.

У вступі наведено інформацію про стан проблеми, здійснено аналіз та запропоновані кроки, які будуть вжиті під час створення проекту.

В першому розділі наведені основні відомості та положення про існуючі алгоритми пошуку шляху в графі та проблеми, з якими може зіткнутися розробник при реалізації механізму процедурної генерації ігрового контенту.

У другому розділі наведена теоретична складова проекту, який вирішує поставлену задачу, встановлені вимоги до розробки, розглянута реалізація A* алгоритму, обґрунтування проектних рішень, список проведених робіт та діаграма класів створюваної програми.

У третьому розділі наведено практичне рішення поставленої задачі з процедурної генерації ігрового поля, а також розглянуті класи, з яких складається програма та методи цих класів, огляд готового проекту.

Практичне значення кваліфікаційної роботи полягає у використанні методу пошуку шляху у графі A* для процедурної генерації ігрового поля, що пропонує альтернативний підхід для розробника під час реалізації механізму процедурної генерації.

Розроблене технологічне рішення може бути запроваджено в галузі електронного розважального бізнесу, а саме для розробки продуктів з елементами процедурної генерації ігрового проля або світу.

ПРОЦЕДУРНА ГЕНЕРАЦІЯ, A* АЛГОРИТМ, UNITY, МОВА ПРОГРАМУВАННЯ C#.

REVIEW

Explanatory note: 57 p., 14 figures, 3 appendices, 18 sources.

Object of development: a procedurally generated field in the Unity environment using the C# programming language.

Purpose: to improve the methods of procedural generation using the A* search algorithm.

The introduction provides information about the state of the problem, analyses and suggests the steps to be taken during the project.

The first section provides basic information and provisions about existing graph search algorithms and the problems that a developer may face when implementing a procedural game content generation mechanism.

The second section presents the theoretical component of the project that solves the task, establishes the requirements for development, considers the implementation of the A* algorithm, justification of design decisions, a list of works performed and a class diagram of the created program.

The third chapter presents a practical solution to the task of procedural generation of the playing field, as well as the classes that make up the programme and the methods of these classes, and an overview of the finished project.

The practical significance of the qualification work lies in the use of the method of finding a path in column A* for the procedural generation of the playing field, which offers an alternative approach for the developer when implementing the procedural generation mechanism.

The developed technological solution can be implemented in the field of electronic entertainment business, namely for the development of products with elements of procedural generation of a game board or world.

PROCEDURAL GENERATION, A* ALGORITHM, UNITY, C# PROGRAMMING LANGUAGE.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАДАЧІ	9
1.1 Поширені проблеми в процедурній генерації поля	11
1.1.1 Проблема недостатньої унікальності в процедурній генерації поля	12
1.1.2 Проблема балансу ігрової складності в рамках процедурної генерації ігорового поля	14
1.1.3 Проблема ефективності обчислень в рамках процедурної генерації ігорового поля	15
1.1.4 Проблема консистентності генерації при генерації ігрового поля	16
1.1.5 Проблема відсутності контролю в рамках процедурної генерації ігрового поля	18
1.2 Методи процедурної генерації	19
1.3 Методи шукання шляху в графі	20
1.3.1 Алгоритм Дейкстри	21
1.3. 2 Алгоритм BFS	22
1.3.3 Алгоритм DFS	23
1.3.4 Алгоритм Лі (Lee algorithm)	24
1.3.5 Що таке метод шукання шляху A*	25
1.4 Середовище розробки Unity	26
ВИСНОВКИ ЗА РОЗДІЛОМ 1	28
РОЗДІЛ 2 РОЗРОБКА ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ ПОЛЯ НА ОСНОВІ A* АЛГОРИТМУ У СЕРЕДОВИЩІ UNITY З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ C#	30
2.1 Основна ідея проекту	30
2.2 Основні вимоги до розробки	31
2.3 Реалізація алгоритму A* для пошуку шляху на процедурно генерованому полі.	32
2.4 Генерація випадкового поля з можливістю задання параметрів, таких як розмір поля, складність, розташування перешкод тощо.	33
2.5 Обгрунтування вибору технології розробки та мови програмування	34
2.6 Проблема використання СУБД	36
2.7 Види та етапи робіт	37
2.8 Загальний опис та структура розробки	38
2.9 Опис діаграми класів	41
ВИСНОВКИ ЗА РОЗДІЛОМ 2	43

РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОЕКТУ РОЗРОБКИ ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ ПОЛЯ НА ОСНОВІ A* АЛГОРИТМУ У СЕРЕДОВИЩІ UNITY З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ C# 44
3.2 Функціонал програми 45
3.2.1 Ігровий інтерфейс 45
3.2.2 Система візуалізації 46
3.2.3 Засоби огляду ігрової мапи 46
3.2.4 Процедурна генерація ігрового поля 49
3.3 Реалізовані класи в програмному коді 49
3.3.1 PlayFieldGeneration 49
3.3.2 PathFinder 51
3.3.3 CameraMovement 53
3.3.4 CameraRotation 53
3.4 Результат роботи 55
ВИСНОВКИ ЗА РОЗДІЛОМ 3 60
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 62
Додаток А PlayFieldGeneration.cs 64
Додаток Б PathFinder.cs 76
Додаток В 81
В. 1 CameraMovement.cs 81
В.2 CameraRotation.cs 81
В.3 MainMenu.cs 82
В.4 Tile.cs 83
Додаток Г ВІДГУК 84
Додаток Д РЕЦЕНЗІЯ 86

ВСТУП

У даному проекті ставиться завдання розробити гру з процедурною генерацією поля на основі A* алгоритму у середовищі Unity. Головною метою проекту є створення ігрового середовища, де гравець зможе досліджувати унінікальні та викликові рівні, які автоматично генеруються кожного разу при початку гри.

Цей проект має прямий зв'язок з об'єктом діяльності фахівців у галузі розробки відеоігор. Процедурна генерація полів та використання A* алгоритму є ключовими компонентами багатьох сучасних ігор, які створюються з метою надати гравцям унікальний та захоплюючий геймплей.

Сучасний стан проблеми вказує на значний прогрес у галузі процедурної генерації ігрових полів та розвитку A* алгоритму. Але все ще існують виклики, такі як забезпечення проходимості поля та створення цікавих та різноманітних рівнів, що потребують додаткових досліджень та розробок.

Метою проекту є реалізація ефективного алгоритму процедурної генерації поля, який забезпечуватиме гравцю постійно змінні та цікаві рівні. Крім того, ми прагнемо створити ігрове середовище, яке буде викликати захоплення та інтерес гравців, надаючи їм неповторний досвід гри кожного разу.

Актуальність теми полягає в поєднанні двох сильних аспектів - процедурної генерації та використання A* алгоритму, які сприяють створенню ігор зі складним та непередбачуваним геймплеєм. Ця комбінація дозволяє розширити межі традиційного геймдизайну та створити незабутні ігрові враження для гравців.

Задачі, які будуть вирішуватись у процесі розробки, включають:

- Реалізацію процедурної генерації поля з використанням різноманітних алгоритмів та технік.
- Впровадження A* алгоритму для пошуку шляху в згенерованому полі.
- Оптимізація алгоритмів та коду для забезпечення ефективності та продуктивності гри.
- Тестування та налагодження проекту для забезпечення якості та готовності до релізу.

Цей проект має великий потенціал у галузі розробки відеоігор та може стати вагомим внеском у покращення геймдизайну та геймплею.

РОЗДІЛ 1
АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАДАЧІ

У даному розділі проводиться аналіз стану області рішення задачі, що пов'язана з розробкою гри з процедурною генерацією поля на основі A* алгоритму[1-3] у середовищі Unity[18] з використанням мови програмування C#[5]. Цей аналіз дозволяє отримати загальне уявлення про сучасний рівень розвитку даної області, ідентифікувати наявні проблеми та визначити можливі прогалини в знаннях або технічні протиріччя.

Аналіз області розпочинається з огляду літератури, наукових публікацій, академічних досліджень та інших джерел, пов'язаних з процедурною генерацією ігрових полів та використанням A* алгоритму. Вивчення цих джерел дозволяє ознайомитись з існуючими методами, підходами та розробками у цій галузі.

Далі проводиться аналіз аналогів - ігор, які використовують процедурну генерацію поля та A* алгоритм. Дослідження таких ігор дозволяє виявити їх сильні та слабкі сторони, зрозуміти, які аспекти можна вдосконалити та які нові ідеї можна внести у власний проект.

Також здійснюється огляд технічних протиріччя та нездійснених вимог до виробів чи розробок, пов'язаних з процедурною генерацією поля та використанням A* алгоритму. Цей аналіз виявляє потенційні труднощі, які можуть виникнути під час реалізації проекту та надає напрямок для подальших досліджень та розробок.

Аналіз стану області є важливим етапом у розробці проекту, оскільки він дозволяє побудувати наявні знання та досвід на основі попередніх досягнень. Він також надає основу для подальшого формулювання мети проекту та постановки конкретних задач, які будуть розглянуті в наступних розділах.

Огляд проектних рішень та аналіз стану області рішення задачі є важливим етапом у проектуванні будь-якого проекту, включаючи створення гри з процедурною генерацією поля на основі A* алгоритму у середовищі Unity. Цей етап дозволяє розглянути наявні підходи, методи та рішення, які використовуються у схожих проектах, і визначити, які можливості та виклики стоять перед проектом.

Аналіз стану області починається з огляду існуючих методів процедурної генерації поля. Вивчення різних методів дозволяє визначити найбільш ефективний та підходящий для даного проекту метод, що забезпечить генерацію цікавих та унікальних ігрових полів.

Також важливим елементом аналізу є розгляд A* алгоритму та його можливих варіантів реалізації. Дослідження різних підходів до використання A* алгоритму допомагає вибрати найбільш оптимальний варіант, що забезпечує ефективну швидкість пошуку шляху і точність у грі.

Далі вивчається середовище Unity та його функціональні можливості. Розгляд інструментів та можливостей Unity допомагає визначити найбільш оптимальний набір інструментів для реалізації проекту і забезпечити його успішну інтеграцію з обраною платформою.

Крім того, аналіз стану області включає огляд існуючих ігор з процедурною генерацією поля та дослідження їх сильних та слабких сторін. Це дозволяє виявити нові ідеї, вдосконалити концепцію гри та забезпечити унікальний досвід для гравців.

В результаті аналізу стану області формулюються конкретні завдання проекту, які будуть розглянуті і вирішені в наступних розділах. Це можуть бути завдання з розробки ефективного алгоритму процедурної генерації, оптимізації швидкості та продуктивності гри, розробки привабливого та зручного інтерфейсу, а також підтримки різних платформ для максимального охоплення аудиторії.

Аналіз стану області рішення задачі надає необхідну основу для подальшого розроблення проекту та допомагає визначити оптимальні рішення, що відповідають потребам та очікуванням користувачів.

1.1 Поширені проблеми в процедурній генерації поля

Процедурна генерація[12] ігрового поля є потужним інструментом, який дозволяє створювати велику кількість унікальних ігрових світів з мінімальними зусиллями. Однак, разом з цим потенціалом вона також вносить свої власні виклики та проблеми. У цьому дослідженні будуть розглянуті поширені проблеми, пов'язані з процедурною генерацією ігрового поля, і будуть запропоновані шляхи їх вирішення. Про проблеми з якими може бути пов’язано процедурне генерування контенту розглядають та згадують у роботах на ці теми, це наводить на думку про широкий спектр потенційних проблем та делікатних аспектів.

Проблема недостатньої унікальності в процедурній генерації поля:

Однією з головних проблем є недостатня унікальність генерованих полів. Якщо алгоритми генерації не забезпечують достатню варіативність, то гравці можуть швидко втомитися від повторюваних ситуацій і втратити інтерес до гри. Розглянуті будуть методи покращення унікальності генерованих полів та забезпечення більшого рівня варіативності.

Проблема балансу ігрової складності в рамках процедурної генерації ігрового поля:

Генерація поля може призводити до небалансу в геймплеї. Неправильно розміщені перешкоди або ресурси можуть зробити гру занадто легко або надто складно. Розглянуті будуть методи вирішення цієї проблеми та підходи до забезпечення належного рівня геймплею.

Проблема ефективності обчислень в рамках процедурної генерації ігрового поля:

При генерації великих ігрових полів можуть виникати технічні проблеми, такі як надмірне використання ресурсів, довгі часи завантаження

або некероване використання пам'яті. Розглянуті будуть методи оптимізації процедурної генерації поля та засоби покращення ефективності обчислень.

Проблема консистентності генерації при генерації ігрового поля:

Генерація поля може викликати проблеми з консистентністю, коли елементи поля не взаємодіють належним чином або не узгоджуються з геймплеєм. Це може призвести до нелогічних або нереалістичних ситуацій у грі. Розглянуті будуть методи забезпечення консистентності генерації та підходи до вирішення цієї проблеми.

Проблема відсутності контролю в рамках процедурної генерації ігрового поля:

Процедурна генерація поля може призводити до втрати контролю над геймплеєм. Гравці можуть зіткнутися з непередбачуваними або несправедливими ситуаціями, що впливають на їх досвід гри. Розглянуті будуть способи встановлення контролю над процедурною генерацією поля та забезпечення бажаного рівня випробування та викликів.

У подальших підрозділах [1.1.1 – 1.1.5] будуть розглянуті ці проблеми детальніше, а також запропоновані рішення для кожної з них, щоб забезпечити високоякісний і захоплюючий геймплей у процедурно генерованих ігрових світах.

1.1.1 Проблема недостатньої унікальності в процедурній генерації поля

Проблема недостатньої унікальності в процедурній генерації поля полягає в тому, що генеровані об'єкти, території або рівні можуть бути недостатньо різноманітними або схожими один на одного. Це може призводити до відчуття монотонності або програвати відчуттю свіжості та цікавості в грі.

Одна з причин цієї проблеми полягає в обмеженості алгоритмів генерації. Деякі алгоритми можуть мати обмежений набір правил або шаблонів, які використовуються для створення об'єктів або рівнів. Це може

призводити до повторюваності та обмеженої варіативності у генерованому контенті.

Ще одна причина - недостатнє врахування контексту або розуміння потреб гравців. Іноді генерований контент може бути технічно унікальним, але не відповідати очікуванням або не створювати цікавих ситуацій на ігровому полі. Наприклад, якщо генеровані рівні в грі завжди мають однакову структуру або розміщення ворогів, це може призвести до відчуття передбачуваності та нудьги для гравців.

Проблема недостатньої унікальності також може виникати з недостатньої варіативності або обмежених можливостей в процедурних алгоритмах. Якщо генератори не враховують достатню кількість параметрів або правил, вони можуть продукувати контент, який схожий один на одного. Це може стати проблемою, особливо в іграх з великим обсягом контенту або в довгострокових іграх, де гравці багато часу проводять у вигаданому світі.

Щоб вирішити ці проблеми, розробники можуть використовувати додаткові техніки та підходи. Наприклад, вони можуть поєднувати кілька алгоритмів генерації для створення більш різноманітного контенту. Використання випадковості, вагових коефіцієнтів та адаптивних алгоритмів також може допомогти створити більш унікальний та цікавий контент. Важливо проводити тестування та збирати відгуки від гравців для виявлення проблем недостатньої унікальності та вчасно вносити корективи в алгоритми генерації.[16]

1.1.2 Проблема балансу ігрової складності в рамках процедурної генерації ігорового поля

Проблема балансу ігрової складності в рамках процедурної генерації [14] ігрового поля виникає з тієї причини, що процес генерації є автоматичним і покладається на алгоритми, що створюють випадкові елементи гри. В результаті цього можуть виникати ситуації, коли гра стає надто легкою або надто складною для гравця, або ж з'являються неприродні або некоректні геймплейні ситуації.

Одна з основних проблем полягає в тому, що процедурно згенероване поле може бути надто одноманітним або, навпаки, надто непередбачуваним. Якщо генерація не забезпечує достатню різноманітність елементів поля або геймплейних ситуацій, гра може стати монотонною та нудною для гравця. З іншого боку, якщо генерація занадто випадкова і не враховує правила балансу, то можуть виникати ситуації, коли гравець стикається з непрохідними або надто складними випробуваннями, що може призвести до фрустрації та втрати інтересу до гри.[16]

Інша проблема полягає в недостатній унікальності та взаємодії елементів на процедурно згенерованому полі. Якщо елементи поля не взаємодіють або не мають достатньої варіативності, то гра може стати передбачуваною та стриманою в своїх можливостях. Гравець може швидко розгадати систему та знайти оптимальні шляхи досягнення успіху, що призведе до втрати цікавості до гри.[17]

Крім того, проблема балансу виникає із суперечності між різними елементами гри, що можуть бути згенеровані процедурно. Наприклад, якщо один тип ворогів має надто сильні характеристики, а інший - надто слабкі, то гра стає небалансованою. Такі дисбаланси можуть призвести до нерівних умов для гравця та непропорційної складності в різних частинах гри.

Для вирішення проблеми балансу ігрової складності в рамках процедурної генерації ігрового поля розробники повинні ретельно налаштовувати алгоритми генерації таким чином, щоб забезпечити достатню

різноманітність, взаємодію та вирівняність елементів поля. Вони можуть використовувати алгоритми балансування, які аналізують статистику гри та автоматично коригують параметри для забезпечення оптимальної складності. Також важливим є залучення гравців до тестування гри, збір їхнього фідбеку та врахування їхніх пропозицій для поліпшення балансу ігрової складності.

1.1.3 Проблема ефективності обчислень в рамках процедурної генерації ігорового поля

Проблема ефективності обчислень[8] в рамках процедурної генерації ігорового поля виникає внаслідок потреби в обробці великої кількості даних та ресурсоємних обчислень під час генерації ігрового світу. Оскільки процес генерації відбувається в реальному часі або перед початком гри, оптимізація ефективності стає критично важливою для забезпечення плавної та швидкої генерації, що впливає на ігровий досвід.

Одна з основних проблем полягає в обчислювальній складності алгоритмів генерації. Якщо алгоритми використовують складні математичні операції або потребують ітераційного проходження через велику кількість даних, то це може призвести до значних затрат обчислювальних ресурсів. Наприклад, генерація складного терену або великих масивів об'єктів може вимагати значних обчислювальних потужностей та тривати значну кількість часу.

Ще однією проблемою є забезпечення оптимального використання ресурсів, таких як пам'ять та процесор. Під час процедурної генерації ігрового поля можуть виникати потреби у зберіганні та обробці великої кількості даних, наприклад, для збереження інформації про кожен елемент світу або для розрахунку фізичних властивостей об'єктів. Це може призводити до надмірного використання пам'яті та перевантаження процесора, що негативно впливає на продуктивність гри.

Додатковою проблемою є потреба в генерації даних в реальному часі або перед початком гри. Якщо генерація займає надто багато часу, то це

може призвести до довгих затримок перед початком гри або зниження швидкості фреймрейту під час геймплею. Гравці вимагають миттєвої відгуку та швидкості дій, тому ефективність обчислень є важливим аспектом процедурної генерації.

Для вирішення проблеми ефективності обчислень у процедурній генерації ігорового поля розробники можуть застосовувати різні стратегії та техніки. Оптимізовані алгоритми генерації, які зменшують обчислювальну складність та використання ресурсів, можуть бути використані для прискорення процесу генерації. Використання кешування даних[12], мультипотокового обчислення[12] та паралельних обчислень[12] можуть також поліпшити ефективність. Крім того, можливо використовувати техніки, такі як ліниве обчислення[9], динамічне завантаження[9] та оптимізоване управління ресурсами[9], щоб забезпечити ефективне використання обчислювальних ресурсів.

Усі ці підходи мають на меті збільшити швидкість генерації ігрового поля, зменшити споживання ресурсів та забезпечити плавну та швидку генерацію.

1.1.4 Проблема консистентності генерації при генерації ігрового поля

Проблема консистентності генерації[15] в рамках процедурної генерації ігрового поля відноситься до питання, як забезпечити, щоб генерований ігровий світ був логічним, збалансованим та зрозумілим для гравця. Основна мета полягає в тому, щоб генеровані об'єкти, ландшафт, рівні або інші елементи максимально відповідали запланованому ігровому досвіду та внутрішнім правилам гри.

Одна з проблем полягає в забезпеченні логічної сполучності між різними частинами ігрового світу. Якщо різні області чи рівні генеруються незалежно один від одного, можуть виникнути нелогічні переходи або розриви в структурі ігри. Наприклад, гравець може виявити, що у двох

сусідніх регіонах або рівнях цілком різний рівень складності, типи ворогів або розміщення об'єктів. Це може порушити поглинання користувачів у ігровий світ та створити відчуття нелогічності в грі.

Інша проблема пов'язана з балансуванням генерації. Генеративні алгоритми можуть не завжди забезпечити правильне балансування складності гри. Наприклад, в деяких випадках гра може генерувати занадто легкі або надто складні рівні, що може призвести до нудьги або фрустрації гравця. Крім того, можуть виникати нелогічні комбінації об'єктів або розміщення ресурсів, що порушує геймплейну рівновагу і може впливати на гравців.

Також проблема консистентності генерації включає у себе важливий аспект - створення генеративних правил та обмежень, які забезпечують, що генеровані об'єкти або рівні відповідають заданим параметрам та вимогам гри. Недостатня чіткість цих правил може призвести до непередбачуваних результатів або негативно вплинути на геймплей.

Для вирішення проблеми консистентності генерації розробники можуть застосовувати різні підходи. Наприклад, використання правил та обмежень[13], що контролюють генерацію, може забезпечити більшу послідовність і логічність. Адаптивні алгоритми генерації можуть змінювати свою поведінку в залежності від контексту або геймплейних факторів, що дозволяє краще узгоджувати генеровані елементи з вимогами гри. Також важливо проводити тестування та зворотний зв'язок з гравцями для виявлення нелогічностей і недоліків у генерованому вмісті та вносити необхідні зміни для поліпшення консистентності.

1.1.5 Проблема відсутності контролю в рамках процедурної генерації ігрового поля

Проблема відсутності контролю[13] в рамках процедурної генерації ігрового поля полягає в тому, що процес генерації відбувається без

достатнього впливу або нагляду розробника, що може призводити до небажаних результатів і непередбачуваних проблем.

Одна з основних проблем відсутності контролю полягає в забезпеченні високої «ігрової» якості ігрового поля. Без достатнього контролю розробник не може гарантувати, що генеровані елементи будуть відповідати запланованому ігровому досвіду та намірам гри. Наприклад, гра може генерувати рівні з недостатньою кількістю викликів до дії або занадто простими ворогами, що може призвести до нудьги або втрати інтересу гравця.

Інша проблема пов'язана з естетичними аспектами генерації. Без контролю розробник не може впливати на вигляд ігрового поля, його візуальні ефекти, архітектуру, освітлення тощо. Це може призвести до неприємного або неякісного візуального досвіду для гравця та порушити загальну атмосферу гри.

Крім того, відсутність контролю може призводити до непередбачуваності і некерованості процесу генерації. Розробник не може гарантувати, що генерація буде досліджувати усі можливі варіанти або не потрапить у тупикові ситуації, які можуть призвести до некоректності чи нестабільності гри.

Для вирішення проблеми відсутності контролю важливо впроваджувати механізми контролю та обмежень в процес генерації. Це можуть бути правила, обмеження чи параметри, що регулюють процес генерації та впливають на результат. Розробники також можуть використовувати комбінацію створення випадковостей та попередньої обробки, щоб досягти керованої непередбачуваності. Також важливо проводити тестування та зворотний зв'язок з гравцями для виявлення проблем і вдосконалення процесу генерації.

1.2 Методи процедурної генерації

Процедурна генерація контенту в відеоіграх дозволяє створення різного контенту при незмінних правилах, що створює можливість нескінченного перегравання для користувача та унікального ігрового досвіду кожного разу.

1. Методи генерації за допомогою шуму[16]: ці методи використовують математичні алгоритми, щоб створити набір випадкових чисел, які надають вигляду поля. Зазвичай ці методи використовуються для створення нерівномірних, органічних форм, таких як гірські хребти або річки.

2. Методи генерації на основі клітин[16]: ці методи розбивають поле на клітинки та заповнюють кожну клітинку випадковими значеннями. Ці методи зазвичай використовуються для створення рівномірних, ретельно структурованих форм, таких як лабіринти.

3. Методи генерації на основі правил[16]: ці методи використовують правила, щоб визначити, які елементи можуть бути додані на поле та як вони можуть бути розміщені. Наприклад, ці методи можуть вимагати, щоб кожен коридор мав певну ширину, або щоб кожна кімната мала певну кількість дверей.

4. Методи генерації на основі шаблонів[16]: ці методи використовують попередньо визначені шаблони, які можуть бути змішані та змінені, щоб створити унікальне поле. Наприклад, можуть бути визначені шаблони для кімнат, коридорів та інших елементів, які можуть бути змішані, щоб створити унікальні комбінації.

1.3 Методи шукання шляху в графі

Існує безліч методів пошуку шляху у графах, кожен з яких має свої переваги та недоліки залежно від конкретної задачі, розміру графа та вхідних даних. Найвідоміші методи пошуку шляху в графі:

1. Алгоритм Дейкстри[7]: це один з найпростіших методів пошуку шляху, який дозволяє знайти найкоротший шлях між двома вершинами. Він працює зваженими графами, де кожній дуги графа відповідає деякий ваговий коефіцієнт.

2. Алгоритм A*: цей алгоритм поєднує в собі ідеї алгоритмів Дейкстри та Хібарда, що дозволяє досягнути ефективної роботи при пошуку шляху в графах. Він використовує heuristics (евристику) для покращення швидкості пошуку шляху.

3. Алгоритм BFS[4]: це алгоритм пошуку шляху в ширину, який працює з невагованими графами та знаходить найкоротший шлях в графі між двома вершинами.

4. Алгоритм DFS[6]: це алгоритм пошуку шляху в глибину, який працює з невагованими графами та знаходить всі можливі шляхи між двома вершинами.

5. Алгоритм Лі[11]: цей алгоритм використовується для пошуку шляху в лабіринті або на полі, де можна рухатися в чотирьох напрямках (вгору, вниз, ліворуч, праворуч). Він використовує відстань Менхеттену для підрахунку вартості кожної вершини в графі.

1.3.1 Алгоритм Дейкстри

Алгоритм Дейкстри є одним з основних алгоритмів для пошуку найкоротшого шляху в графі з невід'ємними вагами ребер. Винайдений голландським математиком Едсгером Дейкстрою у 1956 році, цей алгоритм є ефективним та широко застосовується в різних областях, включаючи маршрутне планування, мережеві протоколи та графічні програми.

Алгоритм Дейкстри працює на основі принципу жадібної стратегії, що означає, що на кожному кроці він обирає найближчу вершину та розглядає всі можливі шляхи до сусідніх вершин. Під час виконання алгоритму зберігається інформація про найкоротшу відому довжину шляху до кожної вершини та попередню вершину, з якої можна дістатися до поточної вершини з найкоротшим шляхом.

Основний крок алгоритму Дейкстри полягає у виборі вершини з найменшою відстанню до початкової вершини та оновленні відстаней до сусідніх вершин. Кожна вершина має асоційовану вагу, яка представляє вартість подорожі до цієї вершини. Якщо нова відстань до сусідньої вершини коротша за поточну, то оновлюється відстань та попередня вершина.

Алгоритм продовжується, поки не буде оброблено всі вершини графу або поки не буде досягнуто кінцевої вершини. Після закінчення алгоритму можна відновити найкоротший шлях, переходячи від кінцевої вершини до початкової за допомогою попередніх вершин.

Основною перевагою алгоритму Дейкстри є його ефективність, коли граф має невелику кількість вершин та ребер. У найгіршому випадку, коли граф є повним, алгоритм має складність O*(V^2), де V - кількість вершин. Однак, застосування піраміди чи черги з пріоритетами дозволяє покращити часову складність до O*((V + E) log V), де E - кількість ребер.

Алгоритм Дейкстри є надзвичайно корисним у випадках, коли необхідно знайти найкоротший шлях між двома вершинами у графі. Він

допомагає розв'язати багато задач, включаючи планування маршрутів у транспортних системах, пошук найкоротшого часу доставки товарів або оптимізацію мережевих протоколів.

1.3. 2 Алгоритм BFS

Алгоритм BFS (Breadth-First Search) або пошук в ширину є одним з основних алгоритмів обходу графів. Він використовується для пошуку найкоротшого шляху в невагованих графах, а також для вирішення інших задач, таких як визначення компонент зв'язності, виявлення циклів, перебору всіх вершин графа та інших.

Алгоритм BFS розпочинається з заданої початкової вершини графа і розглядає всі сусідні вершини цієї вершини перед переходом до наступного рівня вершин. Він просувається "в ширину" по графу, відвідуючи всі вершини на поточному рівні, перш ніж переходити до наступного рівня.

Алгоритм BFS використовує чергу (queue) для зберігання вершин, які потрібно обробити. Початкова вершина додається до черги, а потім починається основний цикл алгоритму. На кожному кроці з черги вибирається вершина, її сусідні вершини додаються до черги, і вони відвідуються. Цей процес продовжується, доки черга не порожня.

Під час виконання алгоритму BFS також зберігається інформація про відвідані вершини та їх відстань від початкової вершини. Це дає змогу визначити найкоротший шлях від початкової вершини до будь-якої іншої вершини, якщо такий шлях існує.

Алгоритм BFS гарантує знаходження найкоротшого шляху в невагованих графах, оскільки він обробляє вершини у порядку їх віддаленості від початкової вершини. Він також забезпечує повний обхід всього графа, якщо він є зв'язним.

Часова складність алгоритму BFS становить O*(V + E), де V - кількість вершин, E - кількість ребер у графі. Просторова складність також становить

O(V + E), оскільки необхідно зберігати інформацію про відвідані вершини та чергу.

Алгоритм BFS може бути застосований в багатьох областях, таких як аналіз мереж, графічні ігри, штучний інтелект, робототехніка та інші, де необхідно вирішувати задачі, пов'язані з графами та шляхами.

1.3.3 Алгоритм DFS

Алгоритм DFS (Depth-First Search) або пошук в глибину є одним з основних алгоритмів обходу графів. Він використовується для вирішення різних задач, таких як пошук шляхів, виявлення циклів, перебор всіх вершин графа, компонент зв'язності та інших.

Алгоритм DFS розпочинається з заданої початкової вершини графа і глибоко проникає в глибину графа, відвідуючи всі сусідні вершини по одному. Коли досягається вершина, яка не має невідвіданих сусідів, алгоритм повертається назад до попередньої вершини і продовжує обхід графа.

Алгоритм DFS використовує стек (stack) для зберігання вершин, які потрібно обробити. Початкова вершина додається до стеку, а потім починається основний цикл алгоритму. На кожному кроці зі стеку вибирається вершина, її сусідні вершини додаються до стеку, і вони відвідуються. Цей процес продовжується, доки стек не порожній.

Алгоритм DFS також використовує мітки для відслідковування відвіданих вершин і уникнення зациклення при наявності циклів у графі. Під час виконання алгоритму відвідані вершини можуть бути помічені, щоб уникнути повторного відвідування.

За допомогою алгоритму DFS можна здійснити пошук шляхів у графі, визначити наявність циклів, знайти компоненти зв'язності та інші характеристики графа. Однак, важливо враховувати, що алгоритм DFS не гарантує знаходження найкоротшого шляху між вершинами.

Часова складність алгоритму DFS залежить від кількості вершин (V) та ребер (E) у графі і становить O*(V + E). Просторова складність також становить O*(V + E), оскільки необхідно зберігати інформацію про відвідані вершини та стек.

Алгоритм DFS може бути використаний у багатьох областях, включаючи графічні ігри, вирішення задач штучного інтелекту, генетичний аналіз, виявлення шаблонів у тексті, аналіз даних та інші.

1.3.4 Алгоритм Лі (Lee algorithm)

Алгоритм Лі (Lee algorithm) або алгоритм пошуку найкоротшого шляху в лабіринті є ефективним методом для знаходження найкоротшого шляху між двома точками у двовимірному масиві або лабіринті. Вперше був запропонований Лі Мой Чжоном у 1961 році.

Алгоритм Лі використовує пошук в ширину (BFS) для пошуку найкоротшого шляху. Він працює наступним чином:

Ініціалізується двовимірний масив або лабіринт, де кожна клітина має стан "прохідна" або "стіна". Точка початку і точка кінця також визначаються.

Створюється черга (queue) для зберігання координат клітин, які потрібно перевірити.

Початкова точка додається до черги. Позначається, що вона має відстань 0.

Запускається цикл, поки черга не порожня. На кожній ітерації вибирається вершина з черги.

Перевіряється кожна сусідня клітина, яка є прохідною і ще не була відвідана. Якщо така клітина знайдена, вона додається до черги, а її відстань встановлюється як відстань вибраної вершини плюс 1. Це означає, що шлях до сусідньої клітини складається з поточного шляху плюс один крок.

Коли досягнута кінцева точка, алгоритм закінчує роботу. Найкоротший шлях може бути відновлений, проходячи від кінця до початку та обираючи сусідні клітини з меншою відстанню.

Алгоритм Лі гарантує знаходження найкоротшого шляху, оскільки він використовує пошук в ширину і обходить всі доступні клітини у порядку збільшення відстані від початку. Це означає, що якщо існує шлях до кінцевої точки, алгоритм обов'язково його знайде.

Часова складність алгоритму Лі залежить від розміру лабіринту і може досягати O(N), де N - загальна кількість клітин у лабіринті. Просторова складність також становить O(N), оскільки необхідно зберігати інформацію про відвідані клітини та чергу.

Алгоритм Лі може бути використаний у багатьох областях, де потрібно знаходити найкоротший шлях, наприклад, у навігаційних системах, робототехніці, симуляціях та ігровій розробці.

1.3.5 Що таке метод шукання шляху A*

Метод шукання шляху A* (проноситься як "Ей зірка" англ. A-star) є алгоритмом шуку найкоротшого шляху між двома вузлами на графі з ваговими ребрами. Алгоритм використовує комбінацію двох функцій, щоб визначити оптимальний шлях: функції g(x) і h(x).

Функція g(x) оцінює довжину шляху від початкового вузла до поточного вузла x. Функція h(x) оцінює відстань від вузла x до кінцевого вузла. Алгоритм використовує значення g(x) та h(x) для кожного вузла, щоб оцінити, який вузол потрібно вибрати на кожному кроці для досягнення кінцевого вузла з мінімальною вартістю шляху.

Алгоритм A* використовує структуру даних, називану відкритий список (Open List), щоб зберігати вузли, які ще не були розглянуті, та структуру даних, називану закритий список (Closed List), щоб зберігати вузли, які були вже розглянуті. Алгоритм продовжує розглядати вузли з відкритого списку в порядку, що визначається значенням функції f(x) = g(x) + h(x), доки не буде знайдений кінцевий вузол або не будуть досліджені всі можливі вузли.

Алгоритм A* створений для найшвидшого пошуку найкоротшого шляху, саме тому ми і будемо його використовувати для вирішення нашої задачі, адже нас не цікавіть ні кількість таких шляхів, ні опції більш довші.

1.4 Середовище розробки Unity

Unity є інтегрованою середовищем розробки (IDE) для розробки ігор та інших інтерактивних додатків. Воно містить в собі ряд інструментів, що допомагають створювати графічні об'єкти, налаштовувати фізику, управляти аудіо та відео, розміщувати об'єкти на сцені та багато іншого.

Unity підтримує багато мов програмування, включаючи C#, UnityScript (раніше відомий як JavaScript) та Boo.[10] Розробники можуть використовувати будь-яку з цих мов для написання скриптів, що взаємодіють з об'єктами у грі.

Окрім цього, Unity має інтегрований двигун фізики, який дозволяє симулювати реалістичну фізику об'єктів у грі, а також підтримує різні платформи, такі як Windows, macOS, Android, iOS, Xbox, PlayStation та багато інших.

За допомогою Unity, розробники можуть створювати ігри різних жанрів, таких як шутери, RPG, платформери, інді-ігри тощо. Додатково, Unity має широкий екосистему плагінів та інструментів, що допомагають розробникам зосередитися на творчому процесі та скоротити час розробки.

Також можна відокремить такі переваги середовища розробки Unity:

- Швидкість розробки: Unity має вбудований інтерфейс, який дозволяє швидко та просто створювати ігрові об'єкти та їх поведінку. Це дозволяє розробникам зосередитися на самому геймплеї та забезпечити швидкий процес розробки.
- Велика спільнота: Unity має велику та активну спільноту розробників, які надають безкоштовні ресурси, підтримку та поради.

Це дозволяє швидко вирішувати проблеми, які виникають під час розробки, а також знайти нові ідеї та підходи до роботи.

- Висока продуктивність: Unity забезпечує високу продуктивність та ефективне використання ресурсів. Це дозволяє створювати великі та складні ігри, які можуть працювати на різних платформах.
- Наявність великої кількості різноманітних плагінів та розширень, що дозволяє розширити функціональність середовища розробки.

ВИСНОВКИ ЗА РОЗДІЛОМ 1

Методи процедурної генерації є потужним інструментом у розробці відео ігор, але вони можуть викликати деякі проблеми. У цьому розділі було розглянуто декілька типових проблем, пов'язаних з процедурною генерацією в ігровій індустрії.

Недостатня унікальність може призводити до відчуття повторюваності та одноманітності у генерованому вмісті, що може негативно вплинути на іммерсію гравця. Баланс ігрової складності є ще однією важливою проблемою, оскільки генеровані рівні або ситуації можуть бути надто легкими або надто складними для гравця, порушуючи гармонію геймплею.

Ефективність обчислень важлива, оскільки складні алгоритми генерації можуть вимагати значних обчислювальних ресурсів, що може бути проблемою на обмежених платформах. Консистентність генерації є викликом, оскільки забезпечення послідовності та сумісності у динамічно генерованому контенті може бути складною задачею.

Відсутність контролю є ще однією проблемою, коли розробник втрачає контроль над створеним вмістом, що може привести до непередбачуваних результатів. Ці проблеми можуть виникати, але їх можна подолати за допомогою ретельного планування, налаштування та тестування процедурної генерації.

Усвідомлення цих проблем та вжиття відповідних заходів допоможуть розробникам створювати більш унікальні, збалансовані та ефективні ігрові світи, які забезпечують незабутні враження гравцям.

A* алгоритм застосовується у багатьох галузях геймдеву та має декілька переваг. Наприклад, його використовують у рольових іграх для генерації рівнів, де він знаходить найкоротший шлях від гравця до цілі, враховуючи перешкоди та складності терену. Також A* використовується для навігації штучного інтелекту у стратегічних іграх і відкритих світах, де він допомагає знаходити оптимальний шлях для гравця або ШІ. У puzzle-

іграх він використовується для перевірки розв'язуваності пазлів та знаходження шляху до розв'язку.

При використанні A* алгоритму та процедурної генерації в ігрових проектах варто враховувати ефективність та різноманітність генерації. Також потрібно дотримуватися балансу між випадковістю та контрольованістю, а також ретельно тестувати згенерований вміст. Генерація повинна забезпечувати задоволення та виклик для гравців, а можливості для редагування або впливу гравця на згенерований вміст можуть покращити взаємодію з гравцем. Підтримка продукту також може бути важливою.

A* алгоритм є ефективним інструментом пошуку оптимального шляху, який має переваги над іншими методами. Він уникає недоліків методів "повний перебір", "сліпий пошук" та "випадковий пошук", забезпечуючи швидкий та ефективний пошук оптимального шляху. A* алгоритм також може бути легко налаштований для врахування різних факторів, що дозволяє контролювати.

РОЗДІЛ 2
РОЗРОБКА ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ ПОЛЯ НА ОСНОВІ A* АЛГОРИТМУ У СЕРЕДОВИЩІ UNITY З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ C#

2.1 Основна ідея проекту

Головною метою є розробити алгоритм, який буде здатен процедурно генерувати ігрове поле таким чином, щоб воно було «проходимим», тобто гравець завжди міг дістатися від старту до фінішу. Алгоритм повинен створювати неповторні патерни ігрового поля, забезпечуючи унікальний досвід для користувача.

Алгоритм повинен бути ефективним та оптимізованим, не використовувати зайвих ресурсів, таким чином буде досягнена висока продуктивність на не самих сучасних обчислювальних машинах, що буде перевагою для перевага для охоплення ширшої аудиторії

Додатково, важливо відзначити, що проект був розроблений з урахуванням принципів розробки відео-ігор та користувацької зручності. Особлива увага була приділена взаємодії гравця з процедурно згенерованим полем, забезпечуючи стимулюючий та цікавий геймплей.

Крім того, було здійснено тестування та налагодження проекту з метою виявлення та виправлення помилок та недоліків. Це дозволило покращити стабільність та якість проекту перед його випуском.

Загальна структура проекту передбачала модульну архітектуру, що сприяла зручному розширенню та підтримці. Код був написаний з урахуванням кращих практик програмування та забезпечував читабельність, модульність та повторне використання.

Всі ці аспекти допомогли забезпечити успішну реалізацію проекту та досягнення його основних цілей, надаючи користувачам унікальний та захоплюючий досвід взаємодії з процедурно генерованим полем у контексті гри.

2.2 Основні вимоги до розробки

Зробивши детальний огляд на вже реалізовані подібні проекти та методику їх розробки та проектні рішення були сформовані таки вимоги до розробки:

Функціональні вимоги:

- Реалізація алгоритму A* для пошуку шляху на процедурно генерованому полі.
- Генерація випадкового поля з можливістю введення параметрів, таких як розмір поля, складність, розташування перешкод тощо.
- Рендерінг поля та його елементів у середовищі Unity, такі як текстури, освітлення, тощо.
- Реалізація контролерів гравця та взаємодії з процедурно згенерованим полем.

Нефункціональні вимоги:

- Сумісність з платформами, на яких планується запуск гри (ОС Windows).
- Ефективність та оптимізація для покращення продуктивності гри та забезпечення плавності ігрового процесу.
- Висока якість графіки та аудіо, що включає деталізацію полів, анімацію персонажів та спеціальні ефекти.
- Надійність та стабільність програмного забезпечення, включаючи обробку помилок та управління виключними ситуаціями.
- Зручний та інтуїтивно зрозумілий інтерфейс користувача, що включає меню, налаштування, інструкцію з гри тощо.
- Сумісність з рекомендованими версіями середовища Unity та інших необхідних залежностей.
- Залежно від конкретних вимог і обсягу проекту можуть бути додаткові вимоги, такі як режим гри в одиночному або

багатокористувацькому режимі, підтримка різних роздільних здатностей екрану, інтеграція з соціальними мережами тощо. Важливо чітко визначити вимоги перед початком розробки, щоб мати чіткий напрямок та функціональність, яку потрібно реалізувати.

2.3 Реалізація алгоритму A* для пошуку шляху на процедурно генерованому полі.

Алгоритм A* є одним з найпоширеніших алгоритмів пошуку шляху в графах та сітках. Він використовує евристичну оцінку для вибору наступного вузла, що має найбільшу потенційну вигоду, що дозволяє досягти мети. Алгоритм A* поєднує в собі низку розрахунків, що оцінюють якість кожного варіанту шляху, з метою знайти найкоротший шлях від початкового вузла до цільового вузла.

Після ретельного огляду різних алгоритмів пошуку шляху, проведеного на основі різноманітних джерел, згідно з даними, представленими в розділі [1], був обраний алгоритм A* як оптимальний варіант для реалізації пошуку шляху на процедурно генерованому полі у середовищі Unity.

Рішення про вибір алгоритму A* базується на його визнаних перевагах щодо ефективності та оптимальності. Відзначається, що алгоритм A* здатний досягати швидкого часу виконання та одночасно обробляти обмежену кількість розглядуваних варіантів, що призводить до зменшення вимог до оперативної пам'яті при його програмній реалізації. Це забезпечує позитивний досвід використання програмного забезпечення, особливо для майбутніх користувачів.

Прикладне застосування алгоритму A* в контексті розробки гри з процедурною генерацією поля дозволяє створити цікаві та непередбачувані шляхи для гравців, покращуючи загальний геймплей та сприяючи більш глибокій взаємодії з грою. Крім того, алгоритм A* демонструє гнучкість та адаптивність до різноманітних варіацій генерації поля, що дозволяє створювати унікальні та захоплюючі рівні для гравців.

Узагальнюючи, використання алгоритму A* для реалізації пошуку шляху на процедурно генерованому полі в середовищі Unity відповідає високим стандартам ефективності, точності та надзвичайного геймплейного досвіду. Цей алгоритм виявляється невід'ємним інструментом у створенні захоплюючих та передбачуваних ігрових світів, що гарантують користувачам задоволення від продукту.

2.4 Генерація випадкового поля з можливістю задання параметрів, таких як розмір поля, складність, розташування перешкод тощо.

Ця вимога покликана забезпечити можливість генерації випадкового поля з варіативними параметрами, що дозволяють їх задавати користувачем. Основною метою поля є створення унікальних ігрових просторів, де гравці можуть взаємодіяти та виконувати різноманітні завдання.

Поперше, це передбачає можливість встановлення розміру поля, що включає його ширину та висоту. Гнучкість у виборі розміру дозволяє створювати ігрові світи з різними масштабами та деталізацією.

Крім того, значимим параметром є складність генерованого поля. Це означає, що розробник повинен мати змогу налаштовувати рівень викликів та складності головоломок чи завдань, які гравці повинні розв'язувати на полі. Це дозволяє створювати ігрові сценарії, що задовольняють різні рівні вмінь та вподобань гравців.

Додатковою можливістю, яку передбачає ця вимога, є можливість визначати розташування перешкод на полі. Це означає, що розробник повинен мати інструменти для введення обмежень та перешкод, які ускладнюють проходження гравцем та створюють викликові ситуації.

Узагальнюючи, реалізація генерації випадкового поля з заданими параметрами дозволяє створювати унікальні, різноманітні та захоплюючі ігрові світи, які привертають гравців своєю непередбачуваністю та можливістю адаптуватися до різних геймплейних вимог та вподобань.

2.5 Обгрунтування вибору технології розробки та мови програмування

В результаті ретельного аналізу спектру доступних варіантів мов програмування та середовищ розробки (таких як C++, Java, Python, Unreal Engine, тощо), було обрано розробку програми за використанням мови програмування C# в середовищі розробки Unity, так як такий вибір є найбільш оптимальним та надасть розробнику низку переваг, які зараз будуть розглянуті.

Створення проекту з процедурною генерацією поля на основі A* алгоритму у середовищі Unity з використанням мови програмування C# має кілька переваг:

- Універсальність: Unity є однією з найпопулярніших інтегрованих середовищ розробки (IDE) для розробки відео ігор. Вона підтримує кросплатформну розробку ігор для різних платформ, таких як Windows, macOS, iOS, Android і багатьох інших. Це дозволяє створювати проекти, які можна публікувати на різних пристроях та операційних системах.
- Зручність програмування: Мова програмування C# використовується в Unity для розробки геймплею, логіки гри та функціональності. C# є потужною, легко зрозумілою та гнучкою мовою, що дозволяє швидко реалізувати складні алгоритми та маніпулювати об'єктами у грі під час розробки та ігрового процесу.
- Розширення та активна спільнота: Unity має велику кількість розширень, плагінів, ресурсів та активну спільноту розробників. Це означає, що ви можете знайти готові рішення, бібліотеки або навіть спілкуватися з іншими розробниками, які працюють над подібними проектами. Це спрощує розробку, забезпечує

доступ до новітніх технологій та полегшує вирішення проблем, з якими ви можете зіткнутися.

- Вбудована підтримка 2D та 3D графіки: Unity надає потужний інструментарій для роботи з 2D та 3D графікою. Ви можете легко створювати реалістичні ігрові світи, об'єкти, анімацію та ефекти. Це дозволяє створювати вражаючі візуальні ефекти та привабливі графічні об'єкти, які залучають гравців до вашої гри. Unity надає широкі можливості для налаштування освітлення, текстур, матеріалів, частинок та багато іншого, що дозволяє досягти високої якості графіки.
- Можливості кросплатформної розробки: Завдяки Unity можна розробляти ігри для різних платформ, включаючи ПК, консолі, мобільні пристрої та віртуальну реальність. Це дозволяє досягти широкої аудиторії ігрових гравців та розширити потенційну базу користувачів вашої гри.
- Широкий спектр інструментів: Unity надає набір інструментів для розробки ігор, включаючи вбудований редактор сцен, редактор анімації, систему частинок, фізичний двигун та багато іншого. Ці інструменти спрощують розробку, тестування та налагодження ігрового вмісту, що дозволяє розробникам більше часу приділяти самому творчому процесу.
- Спільна робота та командна розробка: Unity підтримує спільну роботу над проектами, що дозволяє командам розробників працювати над одним проектом одночасно. Це полегшує співпрацю, обмін ресурсами та керування версіями, що зроблює процес розробки більш ефективним та продуктивним.

Ці переваги роблять Unity ідеальним інструментом для створення проектів, які використовують процедурну генерацію та A* алгоритм. Ви можете поєднувати силу гнучкого програмування C# з потужним інструментарієм Unity длястворення складних ігрових світів, де кожна графа

поля або рівня може бути створена автоматично за допомогою процедурної генерації. Використання A* алгоритму дозволяє ефективно вирішувати завдання пошуку оптимального шляху для гравця або ворогів у цих згенерованих світах.

Завдяки інтегрованій підтримці 2D та 3D графіки в Unity, ви можете створювати реалістичні об'єкти та візуальні ефекти, які доповнюють ваші ігрові світи, роблять їх привабливими та цікавими для гравців.

Крім того, Unity має широкі можливості кросплатформної розробки, що дозволяє вам розгортати вашу гру на різних платформах, таких як ПК, консолі та мобільні пристрої. Це дає можливість залучити широку аудиторію та розширити потенційну базу користувачів для проекту.

Загалом, використання Unity та мови програмування C# для створення проектів з процедурною генерацією та A* алгоритмом дозволяє розробникам втілювати свої творчі ідеї, створювати захоплюючі ігрові світи та надавати гравцям унікальний досвід.

2.6 Проблема використання СУБД

У рамках даного проекту, розробка власного програмного забезпечення для гри з процедурною генерацією поля на основі алгоритму A* у середовищі Unity, система управління базою даних не є необхідною з кількох обґрунтованих причин.

По-перше, враховуючи характер гри з процедурною генерацією поля, її принциповою особливістю є відсутність постійної необхідності зберігання та доступу до значних обсягів даних. Оскільки поле генерується випадковим чином під час кожного запуску гри або на підставі заданих параметрів, не виникає необхідності постійно зберігати інформацію про згенероване поле у базі даних. Вся необхідна інформація може бути збережена тимчасово в оперативній пам'яті під час сеансу гри.

По-друге, впровадження системи управління базою даних може призвести до складності та значних ресурсних витрат, особливо якщо в

контексті проекту відсутня потреба у складних операціях зберігання, оновлення та витягу даних. Розробка та підтримка бази даних можуть зайняти значний час, зусилля та ресурси, які можуть бути краще використані для розвитку інших аспектів гри.

Нарешті, відсутність системи управління базою даних спрощує процес розробки та розгортання гри. Замість витрачання зусиль на складну інтеграцію з базою даних, розробники можуть сконцентруватися на розробці ключового функціоналу гри та оптимізації алгоритму A* для ефективного пошуку шляху.

Зважаючи на наведені фактори, в даному проекті з реалізацією процедурної генерації поля на основі алгоритму A* у середовищі Unity, система управління базою даних не є необхідною, оскільки вона може бути зайвою та не вносити суттєвого внеску в контексті функціональності та ефективності гри.

2.7 Види та етапи робіт

В процесі розробки програмного забезпечення для досягнення успішного результату необхідно виконати ряд видів робіт. Кожен вид робіт має свою важливу роль і сприяє досягненню поставленої мети. У наступному списку представлено логічну послідовність видів робіт, яка може служити орієнтиром під час розробки програмного продукту. З кожним кроком в процесі робіт розширюється розуміння вимог, проектується архітектура, розроблюються функціональність і компоненти, проводяться тестування та оптимізація, і, нарешті, завершується релізом та супроводом проекту. Послідовний виконання цих етапів допомагає забезпечити якість, ефективність та успішний результат розробки програмного забезпечення.

1. Збір вимог і аналіз.
2. Створення плану робіт, які будуть проводитися, та я необхідними для створення програмного забезпечення, яке буде відповідати поставленому завданню.

3. Проектування інтерфейсу користувача.

4. Створення діаграми класів, які необхідні для функціонування програмного забезпечення.

5. Розробка тестових сценаріїв.

6. Реалізація класів, відповідно до створеної діаграми та можливе доповнення її за виробничою необхідністю.

7. Інтеграція зовнішніх компонентів.

8. Оптимізація коду (перший етап).

9. Створення діаграми компонентів програми та діаграми розгортання.

10. Оптимізація коду (другий етап).

11. Етап тестування створеного програмного забезпечення.

12. Виправлення можливих помилок та фінальний етап оптимізації та рефакторингу.

13. Міграція та розгортання.

14. Документування.

15. Конфігурація та управління версіями.

16. Реліз та супровід проекту.

2.8 Загальний опис та структура розробки

Загальний опис та структура розробки проекту, який включає процедурну генерацію поля на основі A* алгоритму у середовищі Unity, можуть бути наступними:

- Огляд та аналіз вимог: Початок розробки проекту передбачає вивчення та аналіз вимог, які включають функціональні та технічні вимоги до гри. Це допомагає зрозуміти, яким чином процедурна генерація та A* алгоритм повинні бути впроваджені в проект.

- Проектування архітектури: Наступним кроком є проектування архітектури проекту, яка включає в себе структуру класів, компонентів та взаємодію між ними. Тут важливо визначити, як будуть взаємодіяти компоненти, включаючи генерацію поля, реалізацію A* алгоритму та взаємодію з графічним інтерфейсом користувача.
- Реалізація процедурної генерації поля: Один з ключових етапів розробки - реалізація алгоритму процедурної генерації поля. Тут використовуються різні методи, такі як генерація лабіринту, випадкове розташування об'єктів, генерація терену тощо. Головна мета полягає в створенні цікавого та унікального геймплею для гравців.

- Реалізація A* алгоритму: Далі виконується реалізація A* алгоритму для пошуку оптимального шляху в згенерованому полі. Цей алгоритм дозволяє гравцеві або ігровим об'єктам знаходити найкоротший шлях до певної цілі, уникаючи перешкод та обмежень.
- Інтеграція графічного інтерфейсу користувача (GUI): Після успішної реалізації процедурної генерації поля та A* алгоритму, необхідно інтегрувати їх у графічний інтерфейс користувача. Це включає в себе створення відповідних кнопок, меню, текстових полів, анімацій та інших елементів, які дозволяють користувачу взаємодіяти з грою. Графічний інтерфейс може відображати інформацію про стан гри, шляхи, ефекти та інші важливі аспекти геймплею.
- Тестування та налагодження: Після завершення реалізації основного функціоналу проекту, важливо провести тестування та налагодження, щоб переконатися, що процедурна генерація поля та A* алгоритм працюють належним чином. Тестування допомагає виявити й виправити помилки, оптимізувати алгоритми та поліпшити геймплей.

- Поліпшення та довершення проекту: Після успішного тестування можна приступати до поліпшення та довершення проекту. Це може включати вдосконалення алгоритмів генерації, оптимізацію швидкодії, вдосконалення візуального вигляду, додавання нових функцій та можливостей, а також вирішення будь-яких зауважень, які виникли під час тестування.
- Реліз та підтримка: Після завершення всіх етапів розробки, проект можна випустити в режимі реального часу. Важливо забезпечити підтримку проекту, вирішувати виявлені проблеми та надавати оновлення та покращення з часом.

Також належить враховувати наступні аспекти:

- Збір зворотного зв'язку від користувачів: Слід активно прислуховуватися до зворотного зв'язку від гравців та розуміти їхні потреби. Це дозволить виявити потенційні проблеми проекту та внести необхідні зміни для поліпшення користувацького досвіду.
- Постійна оптимізація: Розробка гри з використанням процедурної генерації та A* алгоритму може вимагати значних обчислювальних ресурсів. Оптимізація проекту, така як вдосконалення алгоритмів, зменшення навантаження на процесор та пам'ять, допоможе забезпечити плавну роботу гри на різних пристроях та задовольнити потреби користувачів.
- Підтримка різних платформ: Важливо мати на увазі, що ринок відеоігор має потребу в адаптації гри для різних платформ, таких як ПК, консолі або мобільні пристрої. В майбутньому слід забезпечити сумісність проекту з цими платформами та відповідність їхнім вимогам.
- Слідкування за трендами та інноваціями: Індустрія відеоігор постійно розвивається, і нові технології та тренди постійно з'являються.

Важливо бути в курсі останніх інновацій та використовувати їх для поліпшення свого проекту.

2.9 Опис діаграми класів

У процесі розробки програмного забезпечення, був використан підхід об'єктно-орієнтованого програмування, щоб забезпечити ефективність, модульність та розширюваність системи. Одним з ключових елементів цієї архітектури є діаграма класів, яка ілюструє структуру та зв'язки між класами в програмному забезпеченні.

На діаграмі класів, ви знайдете наступні класи: MainMenu, PlayFieldGeneration, Tile, PathFinder та Node. Кожен з цих класів виконує важливу роль у функціонуванні нашої програми та сприяє досягненню її головної мети.

Клас MainMenu відповідає за інтерфейс головного меню програми, де користувач може вибрати режим гри, налаштування та інші опції. Цей клас надає зручний спосіб взаємодії з програмою та керування її основними функціями. [додаток В.1]

PlayFieldGeneration є класом, який відповідає за процедурну генерацію ігрового поля. Він включає алгоритми та логіку, яка дозволяє створювати унікальні та різноманітні ігрові простори, забезпечуючи цікавий геймплей для користувача. [додаток А]

Клас Tile представляє окремий елемент ігрового поля, такий як плитка або тайл. Він зберігає інформацію про стан та властивості цих елементів, а також надає методи для їх обробки та взаємодії з іншими класами. [додаток В.4]

PathFinder є ключовим класом, відповідальним за пошук шляху в ігровому полі. Він використовує алгоритм A*, щоб знайти найкоротший шлях від початкової до кінцевої точки на полі. Цей клас грає важливу роль у реалізації головної механіки гри. [додаток Б]

Нарешті, клас Node використовується для представлення вузлів або точок на ігровому полі, які використовуються алгоритмом пошуку шляху. Він містить інформацію про координати, стан та інші характеристики кожного вузла.

Ця діаграма класів дає глибше розуміння структури програмного забезпечення та зв'язків між його компонентами. Вона служить основою для подальшого розроблення, розширення та підтримки цього проекту, допомагаючи забезпечити якість, ефективність та надійність створеного програмного забезпечення (рис. 2.1).

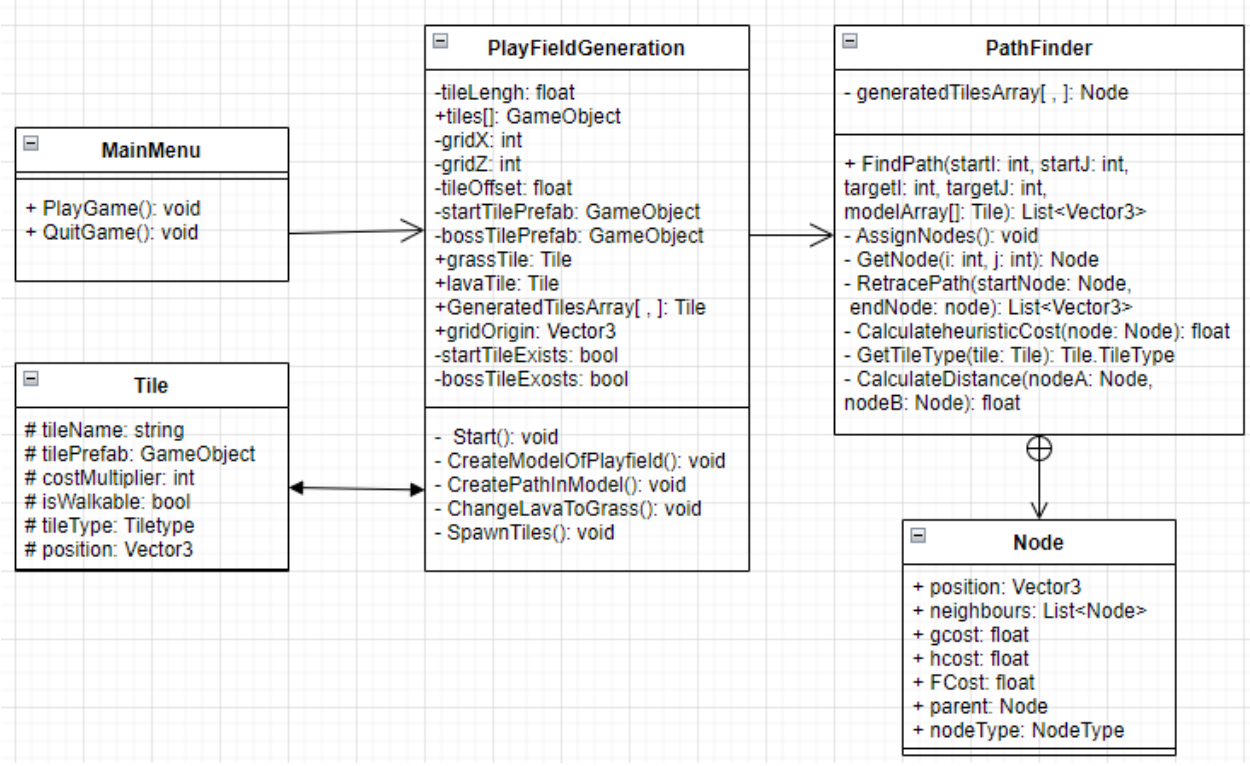

Рис. 2.1. Діаграма класів створеної програми

ВИСНОВКИ ЗА РОЗДІЛОМ 2

В цьому розділі були розглянуті проектні рішення, прийняті у процесі власної розробки гри з процедурною генерацією поля на основі A* алгоритму у середовищі Unity.

Одним з головних проектних рішень було використання середовища Unity для реалізації гри. Unity є потужним інструментом, який надає широкі можливості для створення ігор, включаючи вбудовані функції для генерації випадкових рівнів та роботи з графічними ресурсами.

Для реалізації процедурної генерації поля були використані вбудовані функції Unity, що дозволяють створювати випадкові рівні з різноманітними елементами. Це дало можливість гравцям насолоджуватись новими і унікальними рівнями кожного разу, коли вони починають гру.

Одним з ключових рішень було використання A* алгоритму для пошуку найкоротшого шляху гравця на генерованому полі. A* алгоритм є ефективним алгоритмом пошуку шляху, який використовує комбінацію оцінки вартості і евристичної оцінки для знаходження найоптимальнішого шляху. Це дозволило гравцям швидко й ефективно переміщатись по рівнях та знаходити шлях до цілі.

Також було прийнято рішення надати гравцю можливість взаємодіяти з оточенням та досліджувати нові рівні, що генеруються процедурно. Це створює більш іммерсивний геймплей та дозволяє гравцям відкривати нові області та відкривати нові можливості в грі.

У цьому розділі було представлено огляд проектних рішень, які були прийняті у процесі власної розробки гри з процедурною генерацією поля на основі A* алгоритму у середовищі Unity. Ці рішення дозволили створити успішний проект, який надав гравцям можливість насолоджуватись грою зі створеними випадковими рівнями та забезпечував оптимальний пошук шляху для гравця.

РОЗДІЛ 3
РЕАЛІЗАЦІЯ ПРОЕКТУ РОЗРОБКИ ГРИ З ПРОЦЕДУРНОЮ ГЕНЕРАЦІЄЮ ПОЛЯ НА ОСНОВІ A* АЛГОРИТМУ У СЕРЕДОВИЩІ UNITY З ВИКОРИСТАННЯМ МОВИ ПРОГРАМУВАННЯ C#

3.1 Огляд готового проекту

В результаті проведених робіт, готовий проект являє собою готовий білд програми, яка має інтерфейс користувача, засоби для зручного огляду створеного ігрового світу та саму згенеровану ігрову мапу. Розміри ігрового поля задаються програмно та можуть бути зміненими за бажанням розробника, це реалізовано за використанням змінних, зачення яких задається розробником, тобто змінивши 2 відповідні змінні розміру, програма не втратить здатності коректно генерувати поле.

Отриманий готовий проект представляє собою повноцінну програму з інтерфейсом користувача, яка надає зручні інструменти для огляду та взаємодії зі створеним ігровим світом. Основною особливістю проекту є можливість процедурної генерації ігрового поля.

Головними компонентами готового проекту є ігровий інтерфейс, система візуалізації та засоби для огляду згенерованої ігрової мапи. Інтерфейс користувача дозволяє зручно управляти програмою та вибирати потрібні опції та налаштування. Система візуалізації забезпечує відтворення ігрового світу, включаючи графічне відображення тайлів та ігрових об'єктів. Засоби для огляду ігрової мапи дозволяють користувачу ретельно розглянути створений світ та вивчити його деталі.

Один з ключових аспектів проекту - гнучкість розмірів ігрового поля. Завдяки реалізованій можливості програмного задання розмірів поля, розробник може встановити бажані значення. Це досягається за допомогою змінних, значення яких можуть бути налаштовані розробником. Таким чином, змінивши дві відповідні змінні розміру, програма все ще здатна

коректно генерувати ігрове поле, що дозволяє адаптувати гру під різні умови та вимоги.

Загальний результат роботи - це функціональний та естетичний проект, який виконує поставлені завдання та надає задоволення користувачам. Готовий білд програми є практичним інструментом, який може бути використаний для демонстрації, тестування та насолоди грою з процедурною генерацією поля.

3.2 Функціонал програми

3.2.1 Ігровий інтерфейс

Інтерфейс користувача програми надає зручну та інтуїтивно зрозумілу платформу для взаємодії з ігровим світом. Він містить меню з різними опціями та налаштуваннями, що дозволяють користувачу керувати грою, розпочинати нову гру, зберігати або завантажувати гру та налаштовувати параметри гри.

Було створено ігрове меню, яке дозволяю гравцю почати нову гру, налаштувати ігрові процеси (звук, яскравість, тощо) або зручно та коректно завершити роботу програми. Це проілюстровано на рисунку (3.1)

Загалом, інтерфейс користувача є ключовим компонентом програми, який забезпечує зручну та доступну взаємодію з ігровим світом, дозволяючи гравцям насолоджуватись іммерсивним ігровим досвідом. Ознайомитись з програмно-кодовою частиною інтерфейсу користувача можна в додатку [В.1-В.3].

Рис. 3.1 Ігрове меню, яке представляє вибір користувачу з налаштувань (кнопка Options), самої гри (кнопка Play), де генерується ігрове поле чи виходу з програми (кнопка Quit).

3.2.2 Система візуалізації

Система візуалізації відповідає за графічне відображення ігрового світу, тайлів та об'єктів. Вона забезпечує живописну та реалістичну візуальну подачу гри, де кожен елемент має своє власне зображення та анімацію. Наприклад, при переміщенні персонажа по ігровому полю, система візуалізації оновлює його позицію та анімацію, щоб створити враження руху.

3.2.3 Засоби огляду ігрової мапи

Засоби огляду, які входять до складу програми, надають користувачеві можливість детально досліджувати створений ігровий світ та розглядати його деталі. Зокрема, цей компонент дозволяє збільшувати або зменшувати масштаб ігрової мапи, переміщатися по ній та оглядати окремі області.

За допомогою засобів огляду, користувач може змінювати масштаб мапи, що дозволяє збільшити його для більш детального розгляду конкретної локації або зменшити його, щоб охопити більшу територію. Це дає

можливість краще побачити деталі об'єктів, шляхи, ландшафт та інші елементи, які утворюють ігровий світ.

Крім того, користувач може вільно переміщатися по мапі, використовуючи інтерактивні елементи, такі як панорамування та перетягування. Це дозволяє досліджувати різні області світу гри, зосереджуючись на тих аспектах, які викликають особливий інтерес.

Загалом, засоби огляду ігрового світу надають користувачу можливість контролювати своє взаємодію з ігровим оточенням, дозволяючи йому вивчати його деталі та насолоджуватись візуальним досвідом, який цей проект пропонує.[додаток В.2, В.3]

На рисунках (3.2 – 3.5) проілюстровано здатність керування камерою та переміщення її для огляду ігрового поля, гравець може наближувати, віддаляти, обертати камеру навколо осі на переміщувати камеру за векторами (вперед, назад, вліво, вправо).

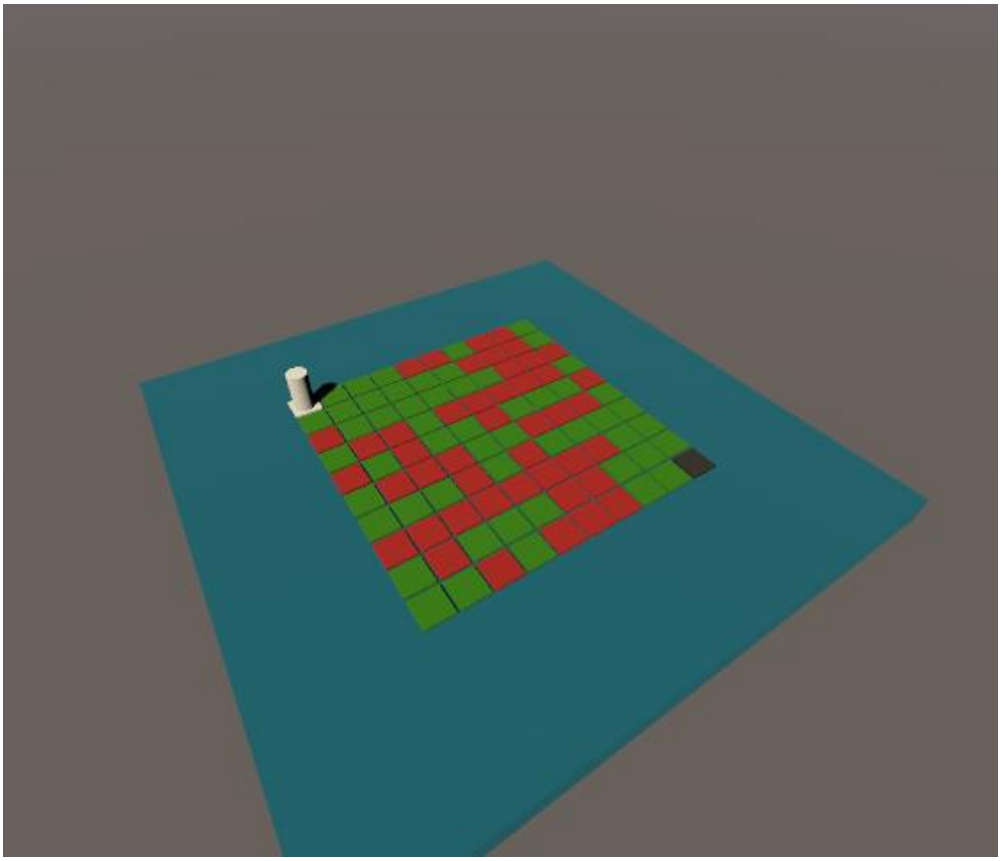

Рис 3.2 Огляд згенерованого ігрового поля з однієї точки

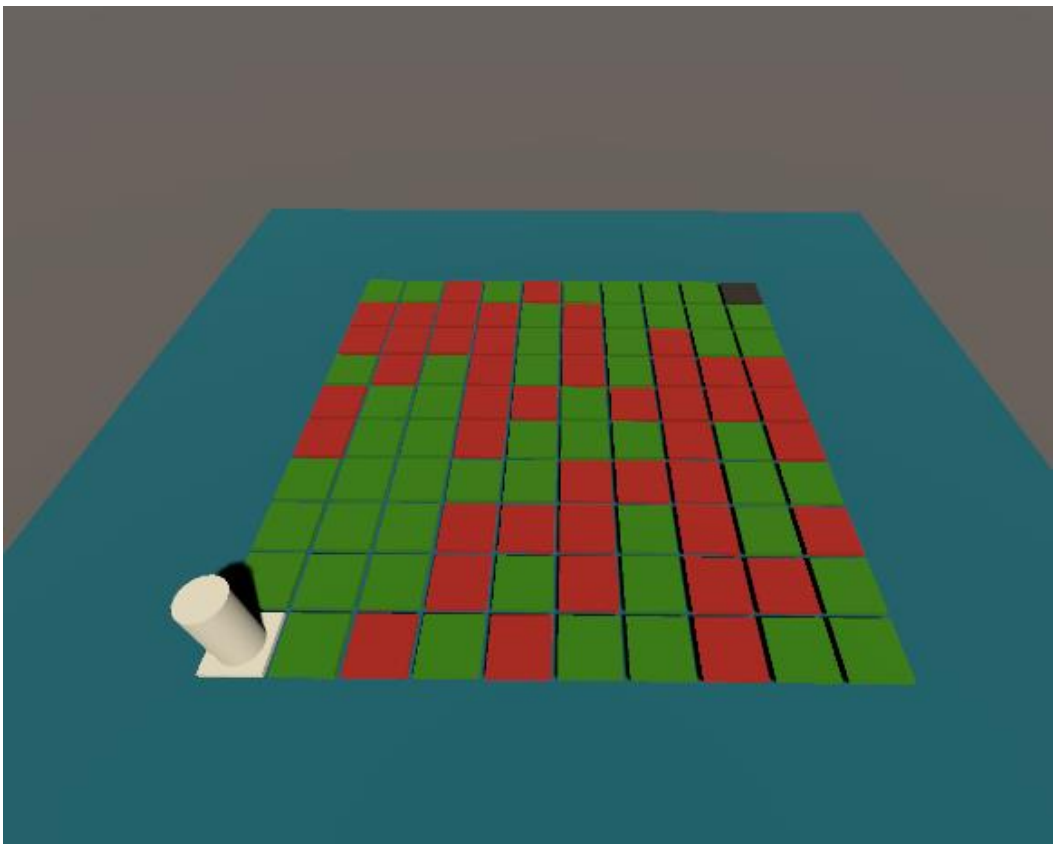

Рис 3.3 Огляд згенерованого ігрового поля з другої точки

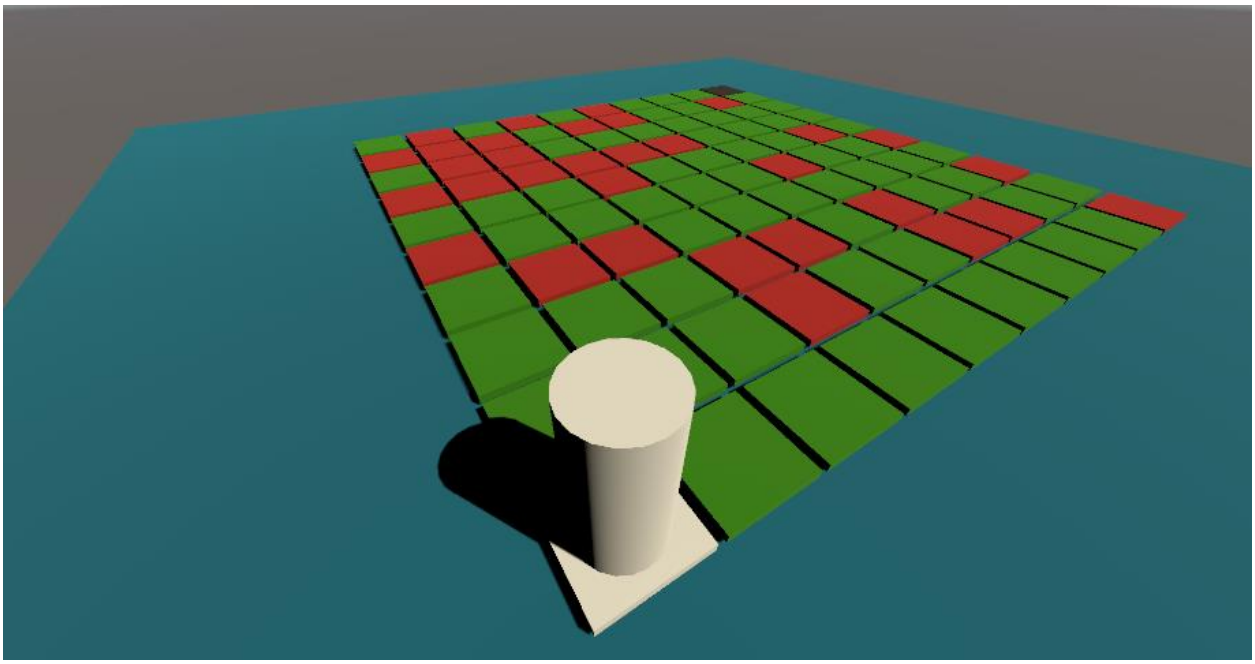

Рис 3.4 Огляд згенерованого ігрового поля з третьої точки

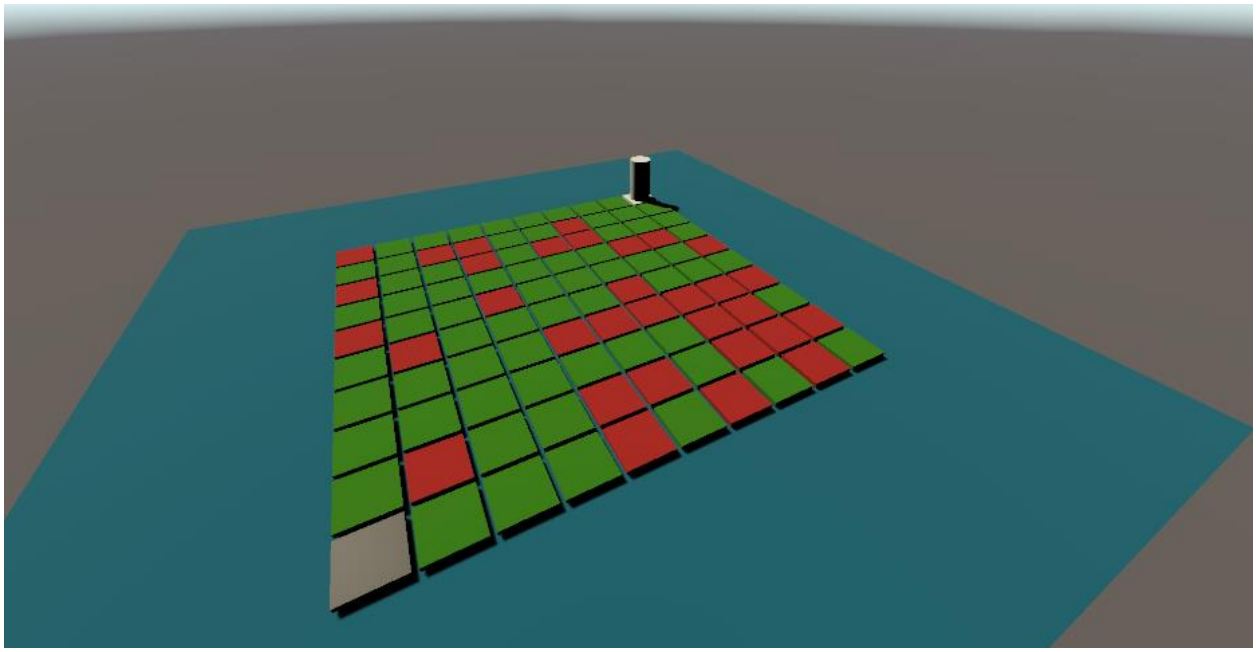

Рис. 3.5 Огляд згенерованого ігрового поля з четвертої точки

3.2.4 Процедурна генерація ігрового поля

Ця функція дозволяє програмі автоматично створювати ігрове поле без необхідності вручну розміщувати тайли. Вона використовує алгоритми та правила для генерації унікальних ігрових полів, що забезпечують різноманітність та цікавість гри кожного разу, коли гравець розпочинає нову сесію.

3.3 Реалізовані класи в програмному коді

В ході розробки програмного забезпечення були реалізовані класи, які раніше описувалися в діаграмі класів та їх відповідні методи.

3.3.1 PlayFieldGeneration [додаток А]

Цей програмний код представляє клас PlayFieldGeneration, який відповідає за генерацію ігрового поля. Він містить різні змінні та методи для створення та редагування поля.

У цьому коді використовуються різні імпортовані бібліотеки, такі як System.Collections, System.Diagnostics, System.IO, System.Threading, UnityEditor та UnityEngine.

Клас має такі змінні:

tileLength - довжина одного елемента поля;

tiles - масив об'єктів GameObject, що містить префаби (макети) для різних типів плиток на полі;

gridX та gridZ - розміри сітки поля;

tileOffset - зсув між плитками;

startTilePrefab та bossTilePrefab - префаби початкової та кінцевої плиток;

grassTile та lavaTile - екземпляри класу Tile для типів плиток;

GeneratedTilesArray - двовимірний масив, який містить створені плитки поля;

gridOrigin - вектор, що визначає початкову позицію поля;

startTileExists та bossTileExists - флаги, які вказують, чи вже створені початкова та кінцева плитки.

У цьому класі також присутні методи:

Start(): Цей метод викликається при запуску програми і викликає інші методи для генерації та редагування поля. Він викликає метод GenerateGrid() для створення сітки поля, CreateModelOfPlayfield() для створення моделі поля, ShowGeneratedTilesArray() для виведення інформації про плитки на полі, і CreatePathInModel() для створення шляху на моделі поля.

GenerateGrid(): Цей метод відповідає за генерацію сітки поля та створення початкової та кінцевої плиток. Він використовує змінні gridX і gridZ для визначення розмірів сітки, ітерується по кожному рядку і стовпцю сітки і створює плитки з використанням префабів з масиву tiles. Початкова плитка створюється за допомогою startTilePrefab, а кінцева плитка - bossTilePrefab. Плитки додаються до масиву GeneratedTilesArray.

CreateModelOfPlayfield(): Цей метод створює модель поля, яка містить інформацію про розташування плиток і їх типи. Він створює новий екземпляр класу PlayfieldModel і заповнює його, обходячи масив GeneratedTilesArray і отримуючи типи плиток з об'єктів плиток. Модель поля зберігається у змінній playfieldModel.

ShowGeneratedTilesArray(): Цей метод виводить інформацію про всі плитки на полі. Він обходить масив GeneratedTilesArray і виводить координати кожної плитки та її тип.

ShowListV3(): Цей метод виводить інформацію про список векторів. Він отримує список векторів як параметр і виводить кожен вектор у консоль.

CreatePathInModel(): Цей метод створює шлях на моделі поля, використовуючи алгоритм пошуку шляху. Він використовує алгоритм пошуку шляху, наприклад, алгоритм A* або DFS (пошук в глибину), щоб знайти шлях між початковою та кінцевою плитками на моделі поля. Шлях зберігається у моделі поля для подальшого використання.

ChangeLavaToGrass(): Цей метод виконує зміну типу плиток з лави на траву. Він обходить масив GeneratedTilesArray, перевіряє тип кожної плитки і замінює тип з лави на траву, якщо це потрібно.

Ці методи спільно працюють для генерації і редагування ігрового поля.

3.3.2 PathFinder [додаток Б]

Цей програмний код представляє клас PathFinder, який відповідає за пошук шляху в грі. Він використовується для знаходження оптимального шляху між двома позиціями на ігровому полі, використовуючи алгоритм пошуку шляху.

Клас містить внутрішній клас Node, який представляє вузол або вершину в графі шляху. Кожен вузол має позицію у просторі (position), список сусідніх вузлів (neighbors), значення вартості шляху від початкового вузла до поточного (gCost), оцінку вартості шляху від поточного вузла до кінцевого вузла (hCost), сумарну оцінку вартості шляху (FCost), батьківський вузол (parent) та тип вузла (nodeType), який вказує, чи це трав'яна плитка чи лавова плитка.

Клас PathFinder має такі методи:

FindPath(int startI, int startJ, int targetI, int targetJ, Tile[,] modelArray): Цей метод приймає початкові та кінцеві координати плиток на ігровому полі

та двовимірний масив modelArray, який містить інформацію про типи плиток. Він використовує алгоритм пошуку шляху (наприклад, алгоритм A*), щоб знайти найкоротший шлях між цими позиціями на полі. Метод повертає список векторів, які представляють шлях від початкової до кінцевої позиції.

AssignNodes(Tile[,] tiles): Цей метод приймає двовимірний масив tiles, який містить інформацію про плитки на полі. Він створює вузли (Node) для кожної плитки на полі і присвоює їм позиції та типи, використовуючи дані з tiles. Вузли зберігаються у статичному двовимірному масиві generatedNodesArray.

GetNode(int i, int j): Цей метод приймає індекси i та j і повертає вузол (Node) з generatedNodesArray за вказаними індексами.

RetracePath(Node startNode, Node endNode): Цей метод приймає початковий вузол (startNode) і кінцевий вузол (endNode) і повертає список векторів, які представляють шлях від початкового до кінцевого вузла.

CalculateHeuristicCost(Node node): Цей метод приймає вузол (node) і обчислює оцінку вартості шляху (hCost) для даного вузла на основі типу плитки (nodeType). Він використовує різні вартості для трав'яних плиток та лавових плиток.

GetTileType(Tile tile): Цей метод приймає об'єкт Tile і повертає тип плитки (tileType) з нього.

CalculateDistance(Node nodeA, Node nodeB): Цей метод приймає два вузли (nodeA і nodeB) і обчислює відстань між їх позиціями.

Цей клас використовується для знаходження шляху на ігровому полі та допомагає ігровому движку визначати, як пересуватись гравцю або іншим об'єктам у грі.

3.3.3 CameraMovement [Додаток В.2]

Цей програмний код представляє клас CameraMovement, який відповідає за рух камери в грі. Клас використовується для контролю позиції камери відносно вхідних даних гравця та параметрів швидкості.

Клас має такі поля та методи:

[SerializeField] private float _speed = 0.1f;: Це поле швидкості камери, яке встановлюється через інспектор Unity. Значення _speed визначає швидкість руху камери.

LateUpdate(): Цей метод викликається один раз на кожен кадр після всіх інших оновлень. У цьому методі відбувається обробка вхідних даних гравця та зміна позиції камери на основі цих даних. Зчитуються значення осей Horizontal та Vertical від вхідних пристроїв гравця (наприклад, клавіатури або джойстика) за допомогою Input.GetAxis(). Значення цих осей використовуються для створення вектора direction з компонентами xDirection, 0 та zDirection. Потім позиція камери (transform.position) змінюється шляхом додавання вектора direction, помноженого на значення _speed.

Цей клас використовується для контролю руху камери в грі, де гравець може переміщувати камеру вперед, назад, вліво та вправо за допомогою відповідних вхідних пристроїв. Швидкість руху камери визначається значенням _speed, яке можна налаштувати у редакторі Unity.

3.3.4 CameraRotation [Додаток В.3]

Цей програмний код представляє клас CameraRotation, який відповідає за обертання та наближення камери в грі. Клас використовується для контролю позиції та орієнтації камери на основі вхідних даних гравця та параметрів чутливості.

Клас має такі поля та методи:

[SerializeField] private float _mouseSensitivity = 8f;: Це поле визначає чутливість руху камери при руху миші.

[SerializeField] private float _scrollSensitivity = 8f;: Це поле визначає чутливість зумування камери при прокрутці миші.

[SerializeField] private float _smoothTime = 0.3f;: Це поле визначає час згладжування руху камери.

[SerializeField] private Transform _target;: Це поле визначає ціль, до якої спрямована камера.

[SerializeField] private float _distance = 5.0f;: Це поле визначає відстань між камерою та ціллю.

LateUpdate(): Цей метод викликається один раз на кожен кадр після всіх інших оновлень. У цьому методі відбувається обробка вхідних даних гравця та зміна орієнтації камери на основі цих даних. Якщо користувач прокручує колесо миші, викликається метод ZoomCamera(). Якщо користувач утримує натиснутою праву кнопку миші, викликається метод MoveCamera().

ZoomCamera(): Цей метод відповідає за зумування камери на основі прокрутки колеса миші. Зчитується значення прокрутки колеса миші за допомогою Input.GetAxis("Mouse ScrollWheel"). Значення цієї прокрутки помножується на _scrollSensitivity, а потім використовується для зміни значення _distance. Значення _distance обмежується в межах від 1 до 20.

MoveCamera(): Цей метод відповідає за рух камери на основі руху миші. Зчитуються значення руху миші по вісі X та Y за допомогою Input.GetAxis("Mouse X") та Input.GetAxis("Mouse Y"). Значення цього руху додаються до _rotationY та _rotationX відповідно. Значення _rotationX обмежується в межах від 0 до 90 градусів. Потім створюється вектор nextRotation з компонентами _rotationX, _rotationY та 0. Вектор _currentRotation згладжується до nextRotation за допомогою Vector3.SmoothDamp(), використовуючи _smoothTime та _smoothVelocity. Орієнтація камери (transform.localEulerAngles) встановлюється на _currentRotation. Позиція камери (transform.position) встановлюється на позицію цілі (_target.position), змінену у напрямку вперед на відстань _distance.

Цей клас використовується для контролю орієнтації та наближення камери в грі, де гравець може обертати камеру навколо цілі за допомогою

руху миші та збільшувати або зменшувати відстань між камерою та ціллю за допомогою прокрутки колеса миші.

3.4 Результат роботи

Під час дослідження була розроблена програма, яка вирішує проблеми, пов'язані з генерацією ігрового поля. При створенні ігрового поля програма використовує червоні та зелені клітини для символізації різних елементів поля. Червоні клітини представляють собою лаву, що позначає непрохідні області, які гравець не може використовувати для переміщення свого персонажу. Зелені клітини символізують траву, що вказує на доступні зони для переміщення. Однак, важливо, щоб при генерації ігрового поля завжди був наявний шлях від початку, де знаходиться прототип персонажу, до самого кінця ігрового поля. Для цього використовуються білі та чорні клітини, які позначають початок та кінець поля відповідно.

На рисунках (3.6 – 3.13) наведені ілюстрації, які демонструють роботу розробленої програми і процес генерації ігрового поля. При кожній новій генерації завжди існує зелений шлях від чорного до білого, незалежно від розміру поля. При тестуванні програма навіть змогла згенерувати поле розміром 1000x1000 (1млн. клітин).

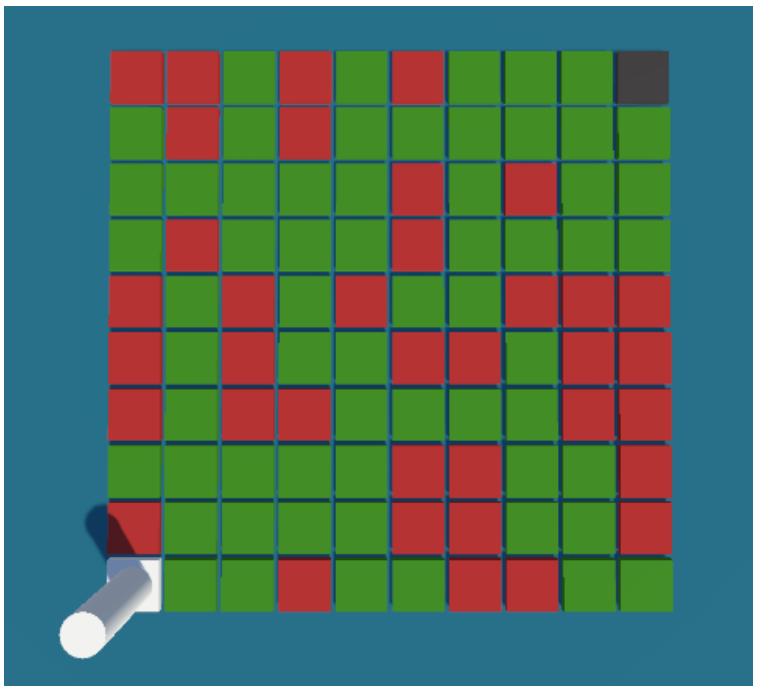

Рис. 3.6 Перша спроба генерації, бачимо шлях, який прямує до центра поля, а потім оминає червони клітини згори.

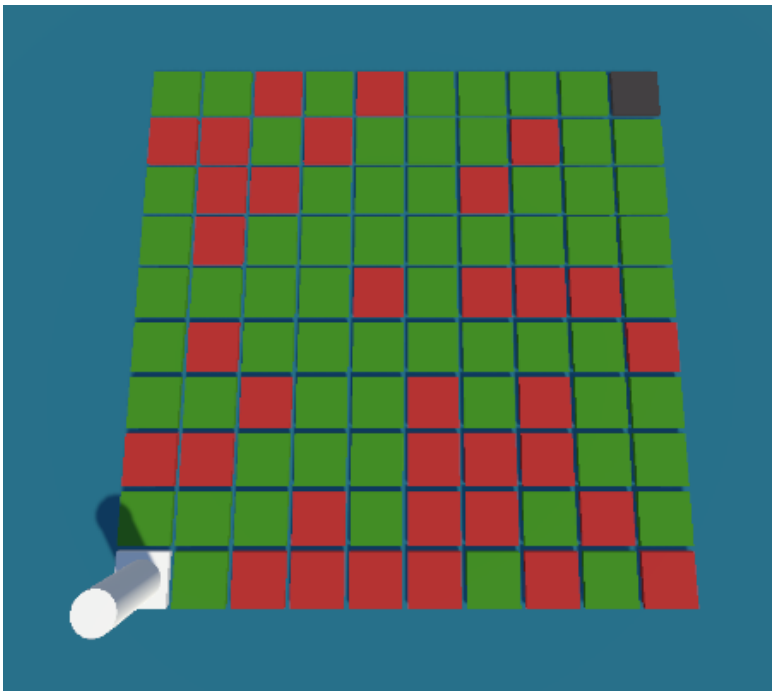

Рис. 3.7 Друга спроба генерації, шлях пролягає через центр поля, але через випадковість генерації існує альтернатива через верх карти, що позитивно складається при формуванні ігрового досвіду у гравця

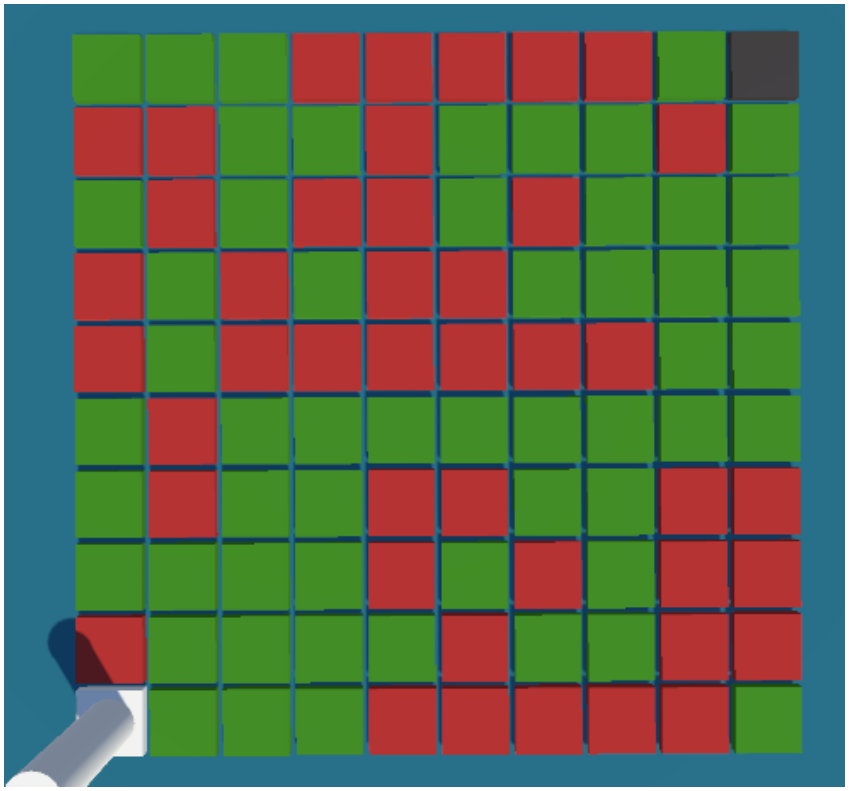

Рис. 3.8 Третя спроба генерації, шлях прокладено через центр поля та через його лівий край, зверху превалюють червоні клітини.

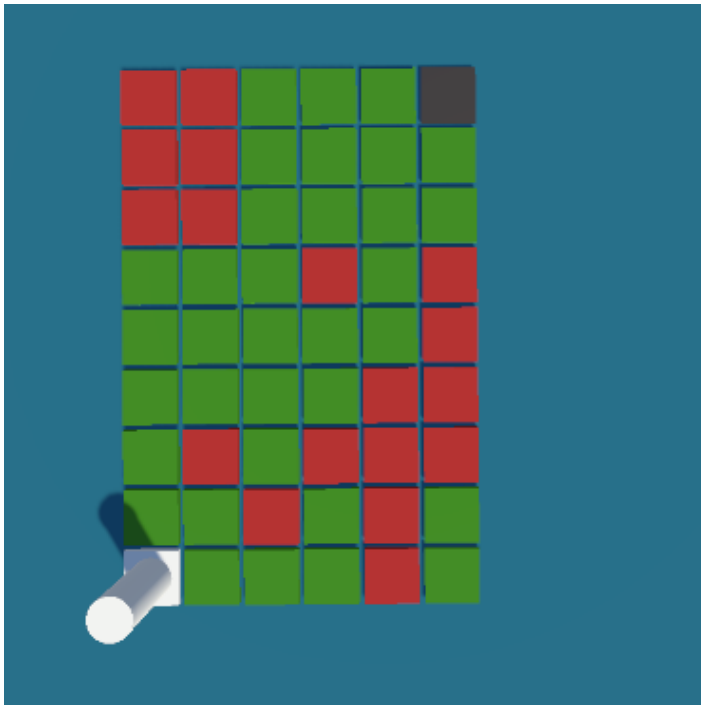

Рис. 3.9 Задання розміру поля 6x9, наглядно бачимо зелений шлях

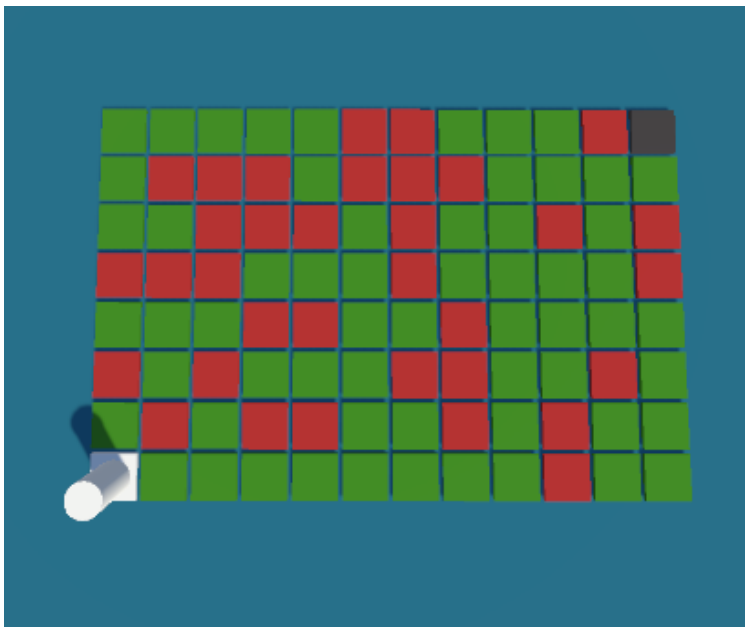

Рис. 3.10 Розмір поля 12x8, шлях побудований через його нижню частину

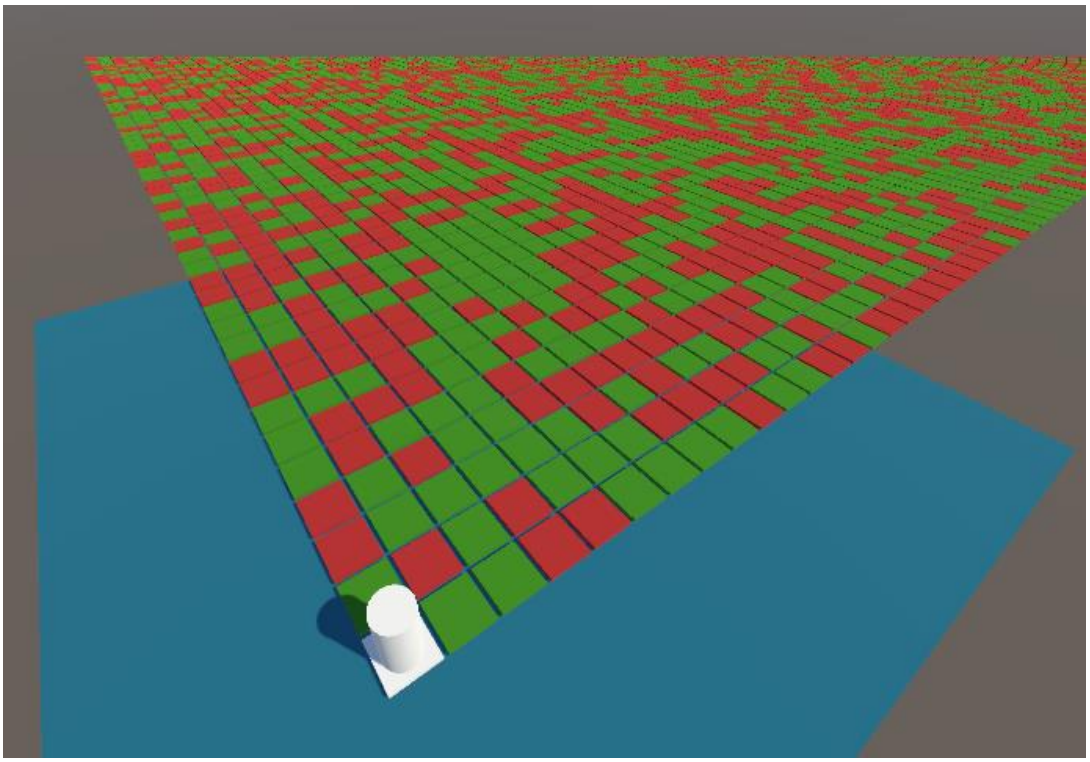

Рис. 3.11 Розмір поля 100x100, через особливості графічного відображення моделей кінець поля лежить за полем бачення гравця

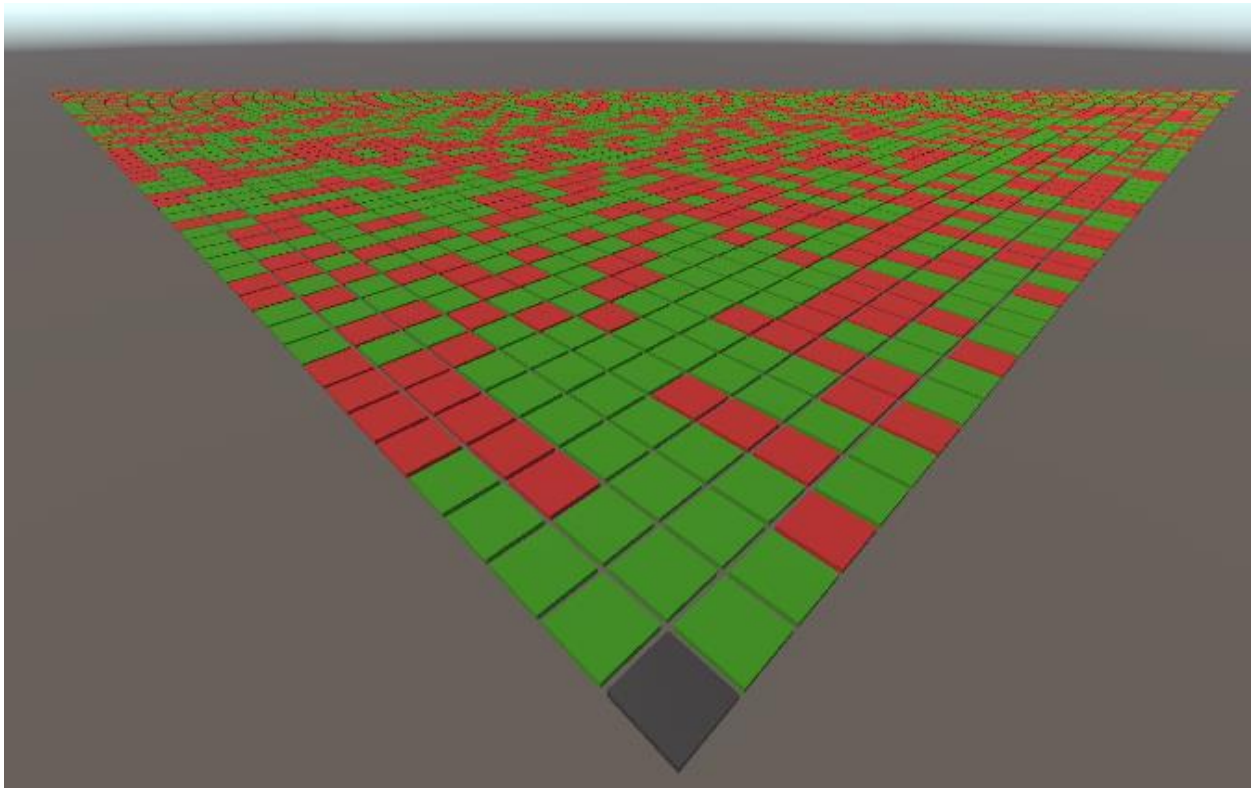

Рис. 3.12 Кінець ігрового поля розміром 100x100

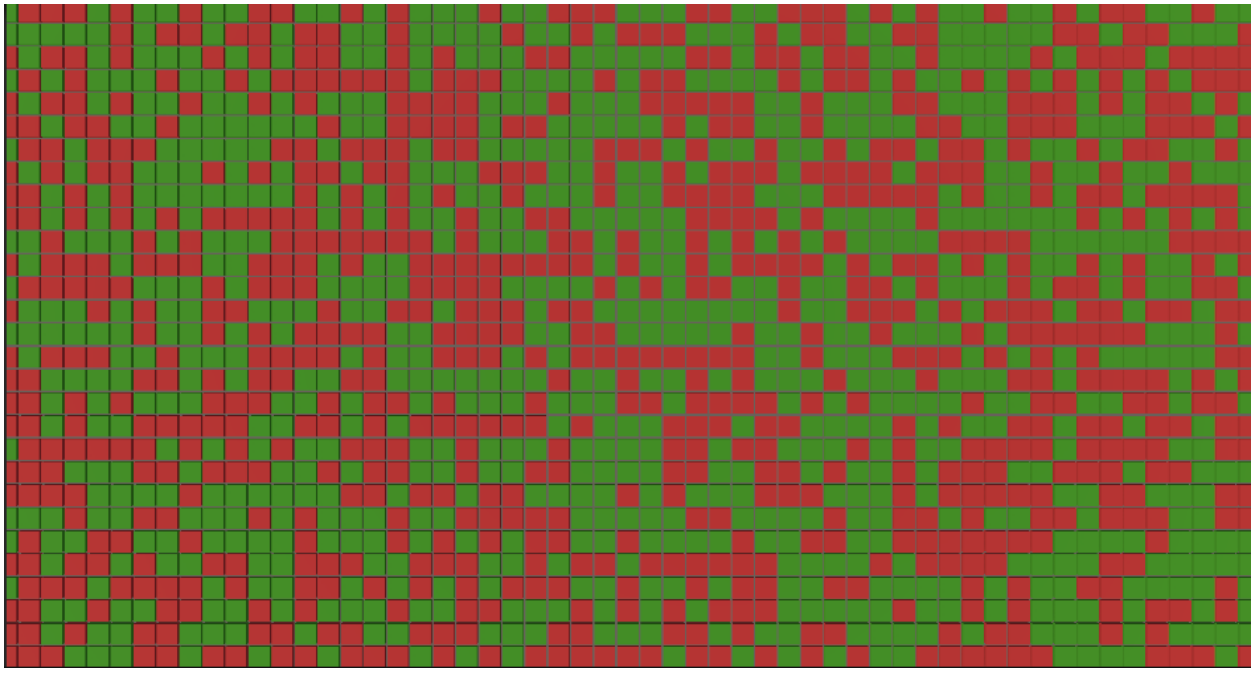

Рис 3.13 Частина з середини ігрового поля, розміром 100x100

ВИСНОВКИ ЗА РОЗДІЛОМ 3

У даному проекті було створено гру з процедурною генерацією поля, що використовує A* алгоритм для пошуку шляху гравця. Такі ігри з повторно генерованими рівнями є дуже популярними серед гравців, тому ця тема є актуальною. Головною метою проекту було дослідження та реалізація процесу генерації випадкових рівнів та оптимального пошуку шляхів для гравця.

У проекті було створено гру в середовищі Unity, де гравець має можливість пересуватися по полю, яке генерується процедурно. Для досягнення цього був використаний A* алгоритм, який дозволяє знаходити найкоротший шлях до цілі на генерованому полі. Гравець може взаємодіяти з оточенням та досліджувати нові рівні, які генеруються кожного разу за допомогою процедурної генерації.

Основною метою проекту було створення ігрового середовища з процедурною генерацією поля та ефективним пошуком шляху гравця за допомогою A* алгоритму. Проект ставив за мету надати користувачу можливість грати у гру з генерованими рівнями, які завжди будуть новими та цікавими. Використання A* алгоритму дозволило забезпечити оптимальний пошук шляху для гравця на кожному згенерованому полі.

Розробка проекту відбувалася у середовищі Unity, яке є популярним інструментом для створення ігор. Для реалізації процедурної генерації поля використовувалися вбудовані функції Unity, що дозволяли створювати випадкові рівні з різноманітними елементами. A* алгоритм був реалізований за допомогою програмного коду, який враховував структуру гри та розміщення об'єктів на полі.

Проект був успішно завершений, а його результати відповідали вимогам. Розмір поля можна задати в інтерфейсі розробника в середовищі Unity. Доречно використовувати невеликі розміри, проте доведено що програма може згенерувати поле розміром 1000x1000, хоча це й займе недоцільно багато часу для проходження користувачем-гравцем.

Реалізація проекту включала створення ігрового середовища у Unity з можливістю процедурної генерації поля. Гравець мав можливість взаємодіяти з оточенням та досліджувати нові рівні, які генерувалися кожного разу з використанням процедурної генерації. A* алгоритм забезпечував ефективний пошук найкоротшого шляху гравця до цілі на генерованому полі, незалежно від розміру ігрового поля.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. A* Algorithm Concepts and Implementation [Електрон. ресурс]. – Спосіб доступу: URL: https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm
2. A* Search Algorithm [Електрон. ресурс]. – Спосіб доступу: URL: https://www.geeksforgeeks.org/a-search-algorithm/
3. A* and Dijkstra search algorithm [Електрон. ресурс]. – Спосіб доступу: URL: https://isaaccomputerscience.org/concepts/dsa_search_a_star?examBoard=all&stage=all
4. Breadth First Search or BFS for a Graph [Електрон. ресурс]. – Спосіб доступу: URL: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
5. C# documentation [Електрон. ресурс]. – Спосіб доступу: URL: https://learn.microsoft.com/en-us/dotnet/csharp/
6. Depth First Search or DFS for a Graph [Електрон. ресурс]. – Спосіб доступу: URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
7. How to find Shortest Paths from Source to all Vertices using Dijkstra’s Algorithm [Електрон. ресурс]. – Спосіб доступу: URL: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
8. Ian Millington Artificial Intelligence for Games (Штучний інтелект для ігор). - CRC Press, Taylor & Francis Group, 2019 - 1010 с.
9. Jeremy Gibson Bond Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C# (Вступ до дизайну ігор, прототипування та розробки: Від концепції до гри з використанням Unity та C#). - Pearson Technology Group Canada, 2022 - 1296 с.
10. Joe Hocking Unity in Action: Multiplatform Game Development in C# (Unity у дії: Розробка ігор для різних платформ на мові C#). - Manning Publications Company, 2015 – 326 с.
11. Lee's Algorithm Explained with Examples [Електрон. ресурс]. – Спосіб доступу: URL: https://www.freecodecamp.org/news/lees-algorithm-explained-with-examples/
12. Noor Shaker, Julian Togelius та Mark J. Nelson Procedural Content Generation for Games: A Textbook and an Overview of Current Research (Процедурна генерація вмісту для ігор: підручник і огляд поточних досліджень) -. Springer, 2016 – 237 с.
13. Paris Buttfield-Addison, Jon Manning та Tim Nugent Unity Game Development Cookbook (Кулінарна книга розробки ігор у Unity). - "O'Reilly Media, Inc.", 13 мар. 2019 – 406 с.
14. Ryan Watkins Procedural Generation in Game Design (Процедурне генерування в дизайні ігор) . - CRC Press, 2017 – 336 с.

15. Ryan Watkins Procedural Generation in Unity Game Development (Процедурне генерування в розробці ігор на Unity). - Packt Publishing Ltd, 2016 – 260 с.
16. Tanya X. Short, Tarn Adams Procedural Generation in Game Design (Процедурне генерування в дизайні ігор) . - CRC Press, 2017 – 336 с.
17. Tanya X. Short, Tarn Adams Procedural Storytelling in Game Design (Процедурне розповідання у дизайні ігор). - CRC Press, 2019 - 408 с.
18. Unity Learn [Електрон. ресурс]. – Спосіб доступу: URL: learn.unity.com

Додаток А

PlayFieldGeneration.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Threading;
using UnityEditor;
using UnityEngine;

public class PlayFieldGeneration : MonoBehaviour
{
    private float tileLength = 1f;

    public GameObject[] tiles;
    [SerializeField] private int gridX;
    [SerializeField] private int gridZ;
    [SerializeField] private float tileOffset = 0.1f;

    [SerializeField] private GameObject startTilePrefab;
    [SerializeField] private GameObject bossTilePrefab;

    public Tile grassTile;
    public Tile lavaTile;



    public Tile[,] GeneratedTilesArray;

    public Vector3 gridOrigin = Vector3.zero;

    bool startTileExists = false;
    bool bossTileExists = false;
```

```
    void Start()
    {
        tileOffset += tileLength;

        UnityEngine.Debug.Log("tiles prefab are:");
        UnityEngine.Debug.Log("0, grass" + tiles[0]);
        UnityEngine.Debug.Log("1, lava" + tiles[1]);

        CreateModelOfPlayfield();
        //ShowGeneratedTilesArray();
        //GenerateGrid();
        CreatePathInModel();
        ClearMemory();
    }

    void GenerateGrid()
    {
        GeneratedTilesArray = new Tile[gridX, gridZ];

        for (int i = 0; i < gridX; i++)
        {
            for (int j = 0; j < gridZ; j++)
            {
                if (!startTileExists)
                {
                    GameObject startTile = Instantiate(startTilePrefab, new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin,
Quaternion.identity);
                    startTileExists = true;
                    Tile startTileAsTile = new Tile { tilePrefab = startTile };
                    startTileAsTile.costMultiplier = 1;
                    GeneratedTilesArray.SetValue(startTileAsTile, i, j);
```

```
            }
            else if (!bossTileExists && i == gridX - 1 && j == gridZ - 1)
            {
                GameObject bossTile = Instantiate(bossTilePrefab, new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin,
Quaternion.identity);
                bossTileExists = true;
                Tile bossTileAsTile = new Tile { tilePrefab = bossTile };
                bossTileAsTile.costMultiplier = 1;
                GeneratedTilesArray.SetValue(bossTileAsTile, i, j);
            }
            else
            {
                Vector3 spawnPosition = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
                ChosingTile(spawnPosition, Quaternion.identity, i, j);
            }
        }
    }

}

private void CreateModelOfPlayfield()
{
    int[,] modelArray = new int[gridX, gridZ];

    GeneratedTilesArray = new Tile[gridX, gridZ];

    modelArray[0, 0] = Tile.TileType.Grass.GetHashCode();
    modelArray[gridX - 1, gridZ - 1] = Tile.TileType.Grass.GetHashCode();

    for (int i = 0; i < gridX; i++)
    {
        for (int j = 0; j < gridZ; j++)
        {
```

```
if (modelArray[i, j] != (int)Tile.TileType.Grass)
{
    int randomInt = UnityEngine.Random.Range((int)Tile.TileType.Grass, (int)Tile.TileType.Lava + 1);
    Tile.TileType randomTile = (Tile.TileType)randomInt;

    switch (randomTile)
    {
        case Tile.TileType.Grass:
            {
                modelArray[i, j] = Tile.TileType.Grass.GetHashCode();
                Tile tile = ScriptableObject.CreateInstance<Tile>();
                tile.tileName = "Grass";
                tile.costMultiplier = 1;
                tile.isGrass = true;
                tile.tilePrefab = tiles[0];
                tile.tileType = randomTile;
                tile.position = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
                //Debug.Log("Tile prefab: " + tile.tilePrefab);
                GeneratedTilesArray.SetValue(tile, i, j);
                break;
            }
        case Tile.TileType.Lava:
            {
                modelArray[i, j] = Tile.TileType.Lava.GetHashCode();
                Tile tile = ScriptableObject.CreateInstance<Tile>();
                tile.tileName = "Lava";
                tile.isGrass = false;
                tile.costMultiplier = 10;
                tile.tileType = randomTile;
                tile.position = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
                tile.tilePrefab = tiles[1];
                //Debug.Log("Tile prefab: " + tile.tilePrefab);
                GeneratedTilesArray.SetValue(tile, i, j);
```

```
                break;
            }
        default:
            break;
    }
}
else if (!startTileExists && i == 0 && j == 0)
{
    Tile tile = ScriptableObject.CreateInstance<Tile>();
    tile.tileName = "Start";
    tile.costMultiplier = 1;
    tile.isGrass = true;
    tile.tileType = Tile.TileType.Grass;
    tile.position = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
    GeneratedTilesArray.SetValue(tile, i, j);
    startTileExists = true;
}
else if (!bossTileExists && i == gridX - 1 && j == gridZ - 1)
{
    Tile tile = ScriptableObject.CreateInstance<Tile>();
    tile.tileName = "Boss (End)";
    tile.costMultiplier = 1;
    tile.isGrass = true;
    tile.tileType = Tile.TileType.Grass;
    tile.position = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
    GeneratedTilesArray.SetValue(tile, i, j);
    bossTileExists = true;
}


  }
}
```

```
}



private void ShowGeneratedTilesArray()
{
    for (int i = 0; i < gridX; i++)
    {
        for (int j = 0; j < gridZ; j++)
        {
            UnityEngine.Debug.Log(GeneratedTilesArray[i, j].tileName + " " + i + " " + j);
        }
    }
}

private void ShowListV3(List<Vector3> list)
{
    int i = 0;

    UnityEngine.Debug.Log("Path is: ");

    foreach (var item in list)
    {
        UnityEngine.Debug.Log("element "+ i + ": (" + list[i].x + "; " + list[i].y + "; " + list[i].z + ";) ");
        i++;
    }

    UnityEngine.Debug.Log("Path ended.");
}

void CreatePathInModel()
{
    Vector3 startTilePosition = new Vector3(0 * tileOffset , 0, 0 * tileOffset) + gridOrigin;
```

```
Vector3 bossTilePosition = new Vector3((gridX - 1) * tileOffset, 0, (gridZ - 1) * tileOffset) + gridOrigin;

if(startTilePosition != GeneratedTilesArray[0,0].position || bossTilePosition != GeneratedTilesArray[gridX-1, gridZ-1].position)

{

   UnityEngine.Debug.Log("CreatePathInModel() ERROR: vector positions didn't match");

   return;

}


//coordinates of elemenst in GeneratedTilesArray

int startCoordinatesI = 0;

int startCoordinatesJ = 0;

int targetCoordinatesI = gridX-1;

int targetCoordinatesj = gridZ-1;


List<Vector3> path = PathFinder.FindPath(startCoordinatesI, startCoordinatesJ, targetCoordinatesI, targetCoordinatesj, GeneratedTilesArray);

if (path != null)

{

   ShowListV3(path);

   ChangeLavaToGrass(path);

}


/*if (path != null && path.Count > 0)

{

   Debug.Log("Here is the path: ");


   for (int i = 0; i < path.Count; i++)

   {

      Debug.Log("X: " + path[i].x + " Z: " + path[i].z);


      if (path[i].costMultiplier == 10)

      {

         path[i].costMultiplier = 1;

      }
```

```
}

int pathCounter = 0;

for (int x = 0; x < gridX; x++)
{
    for (int z = 0; z < gridZ; z++)
    {
        // If the current tile is in the path
        if (pathCounter < path.Count && x == path[pathCounter].x && z == path[pathCounter].z)
        {
            // If the current tile is a lava tile
            if (GeneratedTilesArray[x, z].name == "LavaTile")
            {
                GeneratedTilesArray[x, z] = grassTile;
                GeneratedTilesArray[x, z].name = "GrassTile";
                GeneratedTilesArray[x, z].costMultiplier = path[pathCounter].costMultiplier;
                GeneratedTilesArray[x, z].isWalkable = path[pathCounter].isWalkable;
            }
            pathCounter++;
        }
        // If the current tile is not in the path and it's a grass tile, keep it as is
        else if (GeneratedTilesArray[x, z].name == "GrassTile")
        {
            GeneratedTilesArray[x, z].costMultiplier = 1;
            GeneratedTilesArray[x, z].isWalkable = true;
        }
        // If the current tile is not in the path and it's a lava tile, keep it as is
        else if (GeneratedTilesArray[x, z].name == "LavaTile")
        {
            GeneratedTilesArray[x, z].costMultiplier = 10;
            GeneratedTilesArray[x, z].isWalkable = false;
        }
```

```
            }
        }

        Debug.Log("End of path.");
    }*/

    else
    {
        UnityEngine.Debug.Log("CreatePathInModel() ERROR: Path is null or empty");
    }

   /* Debug.Log("New Array: ");
    ShowGeneratedTilesArray();*/
    SpawnTiles();
}

private void ChangeLavaToGrass(List<Vector3> path)
{
    int counter = 0;

    foreach (Vector3 position in path)
    {
        int x = (int)(position.x / tileOffset); //hardcoded moment 10*1.1(tileOffset) = 11 => int 11 for 10th element (won`t work for grid > 10); / tileOffset fixes it
        int z = (int)(position.z / tileOffset);

        if (GeneratedTilesArray[x, z].isGrass == false)
        {
            GeneratedTilesArray[x, z].isGrass = true;
            GeneratedTilesArray[x, z].tileName = "Grass";
            GeneratedTilesArray[x, z].costMultiplier = 1;
```

```
            GeneratedTilesArray[x, z].tilePrefab = tiles[0];
            counter++;
        }
    }
    UnityEngine.Debug.Log("Transformed " + counter + " tiles.");

}

void DebugLogArray()
{
    for (int x = 0; x < GeneratedTilesArray.GetLength(0); x++)
    {
        for (int z = 0; z < GeneratedTilesArray.GetLength(1); z++)
        {
            UnityEngine.Debug.Log("x: " + x + " z: " + z + " " + GeneratedTilesArray[x, z].name);
        }
    }
}

private void SpawnTiles()
{
    startTileExists = false;
    bossTileExists = false;

    for (int i = 0; i < GeneratedTilesArray.GetLength(0); i++)
    {
        for (int j = 0; j < GeneratedTilesArray.GetLength(1); j++)
        {
            if (!startTileExists)
            {
                GameObject startTile = Instantiate(startTilePrefab, new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin,
Quaternion.identity);
                startTileExists = true;
```

```
            Tile startTileAsTile = new Tile { tilePrefab = startTile };
            startTileAsTile.costMultiplier = 1;
            GeneratedTilesArray.SetValue(startTileAsTile, i, j);
        }
        else if (!bossTileExists && i == gridX - 1 && j == gridZ - 1)
        {
            GameObject bossTile = Instantiate(bossTilePrefab, new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin,
Quaternion.identity);
            bossTileExists = true;
            Tile bossTileAsTile = new Tile { tilePrefab = bossTile };
            bossTileAsTile.costMultiplier = 1;
            GeneratedTilesArray.SetValue(bossTileAsTile, i, j);
        }
        else
        {
            if (GeneratedTilesArray[i, j].tilePrefab == null)
            {
                UnityEngine.Debug.Log("Tile prefab not found for Lava tile. Assigning default lava prefab.");
                GeneratedTilesArray[i, j].tilePrefab = tiles[0];
            }
            Vector3 spawnPosition = new Vector3(i * tileOffset, 0, j * tileOffset) + gridOrigin;
            GameObject randomTile = Instantiate(GeneratedTilesArray[i, j].tilePrefab, spawnPosition, Quaternion.identity);
        }
      }
    }
  }

  void ChosingTile(Vector3 spawnPosition, Quaternion spawnRotation, int i, int j)
  {
    int tileIndex = UnityEngine.Random.Range(0, tiles.Length);
    GameObject randomTile = Instantiate(tiles[tileIndex], spawnPosition, spawnRotation);

    Tile tile = new Tile() { tilePrefab = randomTile };
```

```
        if (tileIndex == 0)
        {
            tile = grassTile;
            tile.tileName = "Grass";
            tile.costMultiplier = 1;

        }
        else
        {
            tile = lavaTile;
            tile.tileName = "Lava";
            tile.costMultiplier = 10;

        }
        GeneratedTilesArray.SetValue(tile, i, j);
    }

    void ClearMemory()
    {
        GeneratedTilesArray = null;
        tiles = null;
        GC.Collect();
    }

}
```

Додаток Б

PathFinder.cs

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

public class PathFinder
{
    private const int STRAIGHT_MOVE_COST = 10;
    private const int DIAGONAL_MOVE_COST = 14;
    private static Node[,] generatedNodesArray;

    private class Node
    {
        public Vector3 position;
        public List<Node> neighbors;
        public float gCost;
        public float hCost;
        public float FCost => gCost + hCost;
        public Node parent;
        public NodeType nodeType;

        public Node(Vector3 pos)
        {
            position = pos;
            neighbors = new List<Node>();
            gCost = 0f;
            hCost = 0f;
            parent = null;
        }
    }

    public enum NodeType
    {
        Grass,
        Lava
    }

    public static List<Vector3> FindPath(int startI,int startJ, int targetI, int targetJ, Tile[,] modelArray)
    {
        int gridX = modelArray.GetLength(0);
        int gridZ = modelArray.GetLength(1);
        generatedNodesArray = new Node[gridX, gridZ];

        AssignNodes(modelArray);
```

```
        Node startNode = GetNode(startI, startJ);
        Node targetNode = GetNode(targetI, targetJ);

        List<Node> openList = new List<Node>();
        HashSet<Node> closedSet = new HashSet<Node>();

        openList.Add(startNode);

        while (openList.Count > 0)
        {
            Node currentNode = openList[0];

            currentNode.hCost = CalculateHeuristicCost(currentNode); //error was here

            for (int i = 0; i < openList.Count; i++)
            {

                if (openList[i].FCost < currentNode.FCost || openList[i].FCost == currentNode.FCost && openList[i].hCost
< currentNode.hCost)
                {
                    currentNode = openList[i];
                }
            }

            openList.Remove(currentNode);
            closedSet.Add(currentNode);

            if (currentNode == targetNode)
            {
                return RetracePath(startNode, targetNode);
            }

            foreach (Node neighbour in currentNode.neighbors)
            {
                if (closedSet.Contains(neighbour))
                {
                    continue;
                }

                neighbour.gCost = currentNode.gCost + CalculateDistance(currentNode, neighbour);

                bool isDiagonal = false;
                if(System.Math.Abs(currentNode.position.x - neighbour.position.x) ==
System.Math.Abs(currentNode.position.z - neighbour.position.z))
                {
                    isDiagonal = true;
```

```
            }

            float hCost = CalculateHeuristicCost(neighbour);

            float newMovementCostToNeighbor = currentNode.gCost + Vector3.Distance(currentNode.position,
neighbour.position);

            if (newMovementCostToNeighbor < neighbour.gCost || !openList.Contains(neighbour))
            {
               neighbour.gCost = newMovementCostToNeighbor;
               neighbour.hCost = hCost;
               neighbour.parent = currentNode;

               if (!openList.Contains(neighbour))
               {
                  openList.Add(neighbour);
               }
            }
         }
      }

      // Path not found
      return null;
   }

   private static void AssignNodes(Tile[,] tiles)
   {
      int gridX = tiles.GetLength(0);
      int gridZ = tiles.GetLength(1);



      for (int i = 0; i < gridX; i++)
      {
         for (int j = 0; j < gridZ; j++)
         {
            Tile tile = tiles[i, j];

            Node node = new Node(tile.position); // Assign position from the Tile to the Node

            // Assign other properties to the node if needed
            node.nodeType = (NodeType)GetTileType(tile);

            generatedNodesArray[i, j] = node;
         }
      }
```

```
    // Assign neighbors to each node based on the adjacent tiles
    for (int i = 0; i < gridX; i++)
    {
        for (int j = 0; j < gridZ; j++)
        {
            Node currentNode = generatedNodesArray[i, j];

            // Add the neighboring nodes
            if (i > 0)
                currentNode.neighbors.Add(generatedNodesArray[i - 1, j]); // Left neighbor
            if (i < gridX - 1)
                currentNode.neighbors.Add(generatedNodesArray[i + 1, j]); // Right neighbor
            if (j > 0)
                currentNode.neighbors.Add(generatedNodesArray[i, j - 1]); // Bottom neighbor
            if (j < gridZ - 1)
                currentNode.neighbors.Add(generatedNodesArray[i, j + 1]); // Top neighbor
        }
    }
}
private static Node GetNode(int i, int j)
{
    return generatedNodesArray[i, j];
}

private static List<Vector3> RetracePath(Node startNode, Node endNode)
{
    List<Vector3> path = new List<Vector3>();
    Node currentNode = endNode;

    while (currentNode != startNode)
    {
        path.Add(currentNode.position);
        currentNode = currentNode.parent;
    }

    path.Reverse();
    return path;
}

private static float CalculateHeuristicCost(Node node)
{
    int moveCost = 10; // the value of the tile itself, does not depends upon path

    // Assign different heuristic costs based on the node type
    switch (node.nodeType)
```

```
        {
            case NodeType.Grass:
                return moveCost * 1;

            case NodeType.Lava:
                return moveCost * 3;

            default:
                return moveCost;

        }
    }

    public static Tile.TileType GetTileType(Tile tile)
    {
        return tile.tileType;
    }

    private static float CalculateDistance(Node nodeA, Node nodeB)
    {
        return Vector3.Distance(nodeA.position, nodeB.position);
    }


}
```

Додаток В

B. 1 CameraMovement.cs

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

public class CameraMovement : MonoBehaviour
{
    [SerializeField] private float _speed = 0.1f;

    // Update is called once per frame
    void LateUpdate()
    {
        float xDirection = Input.GetAxis("Horizontal");
        float zDirection = Input.GetAxis("Vertical");

        Vector3 direction = new Vector3(xDirection, 0, zDirection);

        transform.position += direction * _speed;
    }
}
```

B.2 CameraRotation.cs

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

public class CameraRotation : MonoBehaviour
{
    [SerializeField] private float _mouseSensitivity = 8f;
    [SerializeField] private float _scrollSensitivity = 8f;
    [SerializeField] private float _smoothTime = 0.3f;
    [SerializeField] private Transform _target;
    [SerializeField] private float _distance = 5.0f;

    float _rotationY;
    float _rotationX;

    private Vector3 _currentRotation;
    private Vector3 _smoothVelocity = Vector3.zero;


    void LateUpdate() //the same as update , but is called after all items have been processed in update
    {
```

```
        if (Input.GetAxis("Mouse ScrollWheel") != 0f)
        { ZoomCamera(); }

        if(Input.GetMouseButton(1))
        { MoveCamera(); }

    }

    void ZoomCamera()
    {
        float scrollAmount = Input.GetAxis("Mouse ScrollWheel") * _scrollSensitivity;

        scrollAmount *= (_distance * 0.3f);
        _distance += scrollAmount * -1f;

        _distance = Mathf.Clamp(_distance, 1f, 20f);
    }

    void MoveCamera()
    {
        float mouseX = Input.GetAxis("Mouse X") * _mouseSensitivity;
        float mouseY = Input.GetAxis("Mouse Y") * _mouseSensitivity;

        _rotationY += mouseX;
        _rotationX += mouseY;

        _rotationX = Mathf.Clamp(_rotationX, 0, 90f);

        Vector3 nextRotation = new Vector3(_rotationX, _rotationY, 0);
        _currentRotation = Vector3.SmoothDamp(_currentRotation, nextRotation, ref _smoothVelocity, _smoothTime);
        transform.localEulerAngles = _currentRotation;

        transform.position = _target.position - transform.forward * _distance;
    }
}
```

B.3 MainMenu.cs

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

public class MainMenu : MonoBehaviour
{
    public void PlayGame()
    {
        Debug.Log("Play Game");
```

```
        SceneManager.LoadScene("GameBoard");
    }
    public void QuitGame()
    {
        Debug.Log("Quit Game");
        Application.Quit();
    }
}
```

B.4 Tile.cs

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

[CreateAssetMenu(fileName = "New Tile", menuName = "Tile", order = 0)]
[DataContract]
public class Tile : ScriptableObject
{

    public string tileName { get; set; }

    public GameObject tilePrefab { get; set; }

    public int costMultiplier { get; set; }

    public bool isWalkable = true; //{ get; set; }

    public bool isGrass { get; set; }

    public TileType tileType { get; set; }

    public enum TileType : int
    {
        Grass = 1,
        Lava
    }

    public Vector3 position { get; set; }

}
```