МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему: **«Інформаційне та програмне забезпечення ігрової системи. Аналіз та передбачення поведінки гравців.»**

Завідувач
випускаючої кафедри _____ **Довбиш А.С.**

Керівник роботи _____ **Шелехов І.В.**

Студентка гр. ІН.м-92 _____ **Бацман А.Р.**

СУМИ 2020

ЗМІСТ

ВСТУП

Комп’ютерні ігри займають можливо найбільшу частину дозвілля як сучасної молоді, так і дорослих людей. Останнім часом ця сфера швидко розвивалася, поступово переходячи до віртуальної та доповненої реальності. Ігри стали не тільки розвагою, але і носіями культури. Перевагою ігор є їхня інтерактивність – людина перестає бути лише спостерігачем, вона має здатність змінювати події, впливати на рішення головних героїв.

Працювати в цьому напрямі цікаво, проте це досить складно, адже для розробки ігор необхідний великий обсяг знань у програмуванні, вміння розробляти і розуміти складні алгоритми та командна робота.

Метою даної роботи є дослідження програмних засобів для створення комп’ютерних ігор, їхньої актуальності для певних груп користувачів. Також виконується дослідження вже існуючих алгоритмів проходження лабіринту для пошуку найоптимальнішого. Ідея самої створюваної гри полягає у включенні концепції довіри в проходження алгоритмом лабіринту.

Довіра є невід’ємною частиною життя людей. В роботі буде досліджений вплив «брехливих» інструкцій людини на героя, який проходить лабіринт. Для цього буде виконано навчання системи що дозволить герою передбачувати можливі дії гравця та реагувати на його поведінку.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Огляд ігрових движків

Ігровий движок (англ. Game engine) - це центральний програмний компонент комп'ютерних та відеоігор, і інших інтерактивних додатків з графікою, оброблюваної в реальному часі.[1] Він забезпечує основні технології, спрощує розробку і часто дає грі можливість запускатися на декількох платформах, таких як ігрові консолі та настільні операційні системи, наприклад, GNU / Linux, Mac OS X і Microsoft Windows.

Словосполучення «ігровий движок» має на увазі цілий комплекс прикладних програм, що включає движок рендеринга («візуализатор») для 2D або 3D графіки, фізичний движок, або виявлення зіткнень, звук, скриптинг, анімацію, штучний інтелект, мережевий код, управління пам'яттю і граф сцени. Строго кажучи, всі частини коду, написані програмістами при розробці гри, є компонентами движка. Ігровий процес (геймплей) визначається функціями, реалізованими в цих програмах.[12]

Ігрові движки пропонують 4 основні види функцій:

1. Шаблони гри: шаблони повністю функціональних завершених ігор. Розробник лише змінює дрібні деталі. Наприклад, імпортує власну ілюстрацію або переміщає кілька повзунків, щоб змінити колір неба.

2. Перетягування: дуже просте у використанні - не потрібне знання програмування. Це дозволяє створювати події або властивості, вибираючи їх зі списку. Наприклад, можна додати властивість "твердість" на платформу, для того, щоб персонаж не провалювався через неї.

3. Візуальні сценарії: це дозволяє вибирати різні функції, які копіюють код без програмування. Вони більш потужні, ніж перетягування.

4. Кодування: деякі ігрові двигуни використовують власну мову сценаріїв, намагаючись зробити його максимально легким для початківців.

Інші покладаються на відомі мови, і освоєння їх дасть повний контроль над грою, незалежно від її складності.

Кожний ігровий движок має свої плюси і мінуси у кожній з категорій. Досить важко обрати найкращий з них. Найбільш відомими ігровими движками є Unity і Unreal Engine.

Unity - міжплатформене середовище розробки комп'ютерних ігор. Unity дозволяє створювати додатки, що працюють під більш ніж 20 різними операційними системами, що включають персональні комп'ютери, ігрові консолі, мобільні пристрої, інтернет-додатки та інші. Випуск Unity відбувся в 2005 році і з того часу йде постійний розвиток.

Редактор Unity має простий Drag & Drop інтерфейс, який легко налаштовувати, що складається з різних вікон, завдяки чому можна проводити налагодження гри прямо в редакторі. Движок підтримує дві мови програмувння: C #, JavaScript (модифікація). В Unreal Engine 4 використовується мова програмування C ++. [3]

На відміну від багатьох ігрових движків, у Unity є дві основні переваги: наявність візуальної середовища розробки та міжплатформена підтримка. Перший фактор включає не тільки інструментарій візуального моделювання, а й інтегроване середовище, що направлено на підвищення продуктивності розробників, зокрема, етапів створення прототипів і тестування.

У Unity Asset Store можна придбати розширення для редактора під назвою Playmaker, яке також дозволяє розробляти прототипи ігор без написання коду.[2]

Обидва движки, Unreal Engine 4 iUnity3d, мають свій магазин додатків: з них можна завантажити готові 3d моделі персонажів і оточення, текстури і навіть такі речі, як звуки і системи частинок. [3] Проте, Unity3d однозначно виходить на перше місце з точки зору кількості м в магазині. У ньому є все - від анімацій і генераторів графічного

інтерфейсу користувача (GUI) до розширень редактора для управління штучним інтелектом (ШІ). Там є все, що потрібно для створення гри.

Unreal Engine 4 і Unity3d є умовно безкоштовними. Повна версія Unreal Engine 4 буде безкоштовною, поки дохід від гри становить менше $ 3000 в квартал, якщо ж більше - вам доведеться ділитися 5% від своїх доходів. [15]

З Unity3d все складніше. Якщо доходи компанії становлять менше $ 100 000 на рік, можна використовувати безкоштовну версію, але вона неповна. [4] Хоч і безкоштовної версії цілком достатньо для написання повноцінної гри.

Для 2D-розробки в Unity є чудові інструменти - sprite creator, sprite editor і sprite packer. UE4 також підтримує спрайт в Paper 2d, але рішення з Unity могутніше, крім того, в останньому є окремий фізичний движок для 2d-об'єктів. [5]

1.2 Unity

Unity має велику кількість інструментів для розробки ігор спрямованих як на саму розробку так і на покращення та оптимізацію вже готового продукту:

- Редактор "все-в-одному": доступний у Windows, Mac і Linux, він включає в себе ряд зручних для виконавців інструментів для проектування захоплюючих ігрових світів, а також потужний набір інструментів для розробки логіки гри та високопродуктивного геймплея.
- 2D & 3D: Unity підтримує як 2D, так і 3D-розробку з функціями та функціональністю для конкретних потреб у різних жанрах.
- ШІ для пошуку шляху (*AI pathfinding tools*): Unity включає в себе навігаційну систему для створення неігрових персонажів (NPC), які можуть розумно переміщатися по ігровому світу. Система використовує

навігаційні сітки, які створюються автоматично з геометрії сцени, або навіть динамічні перешкоди, щоб змінити навігацію персонажів під час виконання.

- Ефективні робочі процеси (Unity Prefabs): попередньо налаштовані ігрові об'єкти, які надають ефективні та гнучкі робочі процеси, які дозволяють працювати впевнено, не турбуючись про тимчасові помилки.
- Інтерфейси користувача: Вбудована система інтерфейсу дозволяє швидко і інтуїтивно створювати інтерфейси користувача.
- Фізичні двигуни: Є можливість використання переваг Box2D, нової системи фізики DOTS і підтримкою NVIDIA PhysX для реалістичного і високопродуктивного ігрового процесу.
- Спеціальні інструменти: Редактор можна розширити за допомогою інструментів, необхідних для відповідності робочого процесу. Можна створювати та додавати спеціальні розширення або знайти те, що потрібно, у магазині додатків, який містить тисячі ресурсів, інструментів і розширень для прискорення ваших проектів.
- Підтримка більшої кількості платформ, ніж будь-який інший движок: за допомогою Unity можна охопити найбільш широку публіку.
- Інструменти вдосконаленого профілювання: це дає можливість постійно оптимізувати вміст гри у процесі розробки за допомогою функцій профілювання Unity. Доступна перевірка, чи вміст, наприклад, пов'язаний з процесором або графічним процесором. Це визначає ті області, які потребують поліпшення, щоб надати аудиторії безперебійний досвід роботи. [16]
- Нова, багатомодульна програма Unity дозволяє створювати ігри, які є малими, легкими та швидкими. Миттєвий доступ до контенту є

надзвичайно важливим для максимального охоплення аудиторії на мобільних пристроях.

- Інструменти для дизайнерів та художників. Unity є творчим центром для художників, дизайнерів, розробників та інших членів команди. Він включає в себе 2D та 3D засоби проектування сцени, оповідання історії та кінематографії, освітлення, аудіосистем, інструменти управління спрайтами, ефекти частинок та потужна система анімації персонажів.
- Онлайн ігри. Від динамічних одиночних ігор до багатокористувацьких ігор в реальному часі і поза ними, онлайн ігри є найбільш популярними і успішними. Unity має набір інструментів, служби двигунів і інфраструктуру, необхідні для створення цих видів ігор, і масштабування до будь-якого рівня успіху. [17]
- Ігровий сервер хостингу для багатокористувацьких ігор (Multiplay): Масштабовані і стійкі хостинг-рішення для ігрових серверів. Multiplay підтримує студії, такі як Respawn Entertainment і PUBG Corp., надаючи найкращий досвід для деяких з найбільших ігор на планеті.
- Командна співпраця. Unity Teams дають змогу творчим групам працювати більш ефективно разом із функціями, які дозволяють співпрацювати та спрощувати робочі процеси.
- Хмарна діагностика. Це набір інструментів, які підтримують хмару та допомагають ідентифікувати, збирати та встановлювати пріоритети даних про продуктивність та відгуки кінцевих користувачів.

1.3 Огляд існуючих алгоритмів для проходження лабіринтів

Існує багато алгоритмів для проходження лабіринтів. Деякі з них не можуть знайти вихід при певній структурі, деякі вимагають багато часу та

ресурсів для використання. Також далеко не в усі алгоритми можливо вписати концепцію довіри.

Найпопулярніші алгоритми:

1. Алгоритм «триматися за стіну» (*Wall follower*): Це простий алгоритм розв'язання лабіринту.[6] Він завжди працює швидко, і не використовує додаткової пам'яті. Кожен раз при досягненні розвилки завжди виконується поворот праворуч (або ліворуч). Схоже на людину яка вирішує лабіринт, поклавши руку на праву (або ліву) стіну і залишивши її там. Цей метод не обов'язково знайде найкоротше рішення, і він не працює взагалі, коли ціль знаходиться в замкнутому контурі. Через неповний розв’язок лабіринту цей алгоритм недоцільно використовувати.

2. Алгоритм застави (*Pledge algorithm*): Це модифікована версія Слідувача стіни, яка здатна стрибати між островами. Починається алгоритм з вибору напрямку і завжди виконується рух в цьому напрямку доки це можливо. Коли на шляху стає стіна починається слідування цієї стіни доки алгоритм знову не повернеться на обраний напрямок. Підраховується кількість витків, наприклад, лівий поворот: -1, поворот вправо: 1. Слідування стіни зупиняється коли загальна кількість зроблених обертів дорівнює 0. Підрахунок гарантує, що в кінцевому підсумку можливо досягти далекого боку поточного острова і перейти на наступний острів у вибраному напрямку. Це гарантований спосіб досягти виходу на зовнішній край будь-якого 2D лабіринту з будь-якої точки середини, однак неможливе зворотне, тобто знайти ціль в лабіринті.

3. Алгоритм заповнення тупиків (*Dead end filler*): Цей алгоритм завжди працює дуже швидко і не використовує додаткової пам'яті.[11] Переглядається лабіринт і заповнюється кожен глухий кут, поки не досягнуто перехід. Це включає заповнення проходів, які стають тупиками після кожного заповнення. В кінці залишаться тільки рішення.

Допрацьована версія алгоритму також вставляє стіни в петлі, що дозволяє після нового використання фільтру позбутися зациклень в лабіринті. Цей алгоритм незручний для використання з концепцією довіри.

4. Алгоритм Трема (*Trémaux's algorithm*): Цей метод розв'язання лабіринту призначений для використання людиною.[8] Він знайде рішення для всіх лабіринтів. Йдучи по проходу малюється лінія. Коли зустрічається раніше не відвідане перехрестя обирається новий прохід у випадковому порядку. При потраплянні в глухий кут виконується повернення. Якщо йдучи новим проходом зустрічається раніше відвідане перехрестя, воно вважається глухим кутом. (Цей останній крок є ключем, який перешкоджає об'їжджати кола або відсутні проходи). Всі проходи будуть порожніми, відзначеними один раз або двічі позначеними. Після знаходження виходу, проходи, позначені точно один раз, покажуть прямий шлях до початку. Якщо лабіринт не має жодного рішення, ви знову опинитеся на початку з усіма відмітками, позначеними двічі. В цей алгоритм досить легко вписати концепцію довіри, він працює для усіх видів лабіринтів. Тому для подальшої розробки буде використаний саме даний лабіринт.

5. Алгоритм найкоротшого шляху: Як видно з назви, цей алгоритм знаходить найкоротше рішення, вибираючи його, якщо їх існує декілька. Він є швидким для всіх типів лабіринтів, і вимагає зовсім небагато додаткової пам'яті, пропорційної розміру лабіринту. Цей алгоритм затоплює лабіринт «водою», так, щоб всі відстані від початку заповнюються одночасно, однак кожна «крапля» або піксель запам'ятовує піксель яким вони були заповнені. Після того, як в вихід вдарить "крапля", прослідковується шлях назад від неї до початку і він буде найкоротший. Цей алгоритм добре працює з будь-яким лабіринтом, тому що не вимагає, щоб лабіринт мав будь-які піксельні ширини, через які можна пройти.

Хоча цей метод швидкий і ефективний, він не може працювати з вибором людиною наступного шляху для нього.

6. Алгоритм вирішення зіткнень: Цей метод також називатиметься розв'язувачем "амеби".[7] Він є швидким для всіх типів лабіринтів, і вимагає принаймні одну копію лабіринту в пам'яті та пам'ять відповідну до розміру лабіринту. Цей метод затоплює Лабіринт «водою», так що всі відстані від початку заповнюються одночасно і всякий раз, коли дві «колони води» наближаються до переходу з обох кінців (вказуючи петлі) додається стіна до оригінального лабіринту, де вони стикаються. Після того, як всі частини Лабіринту були "затоплені", заповнюються всі нові тупики, які не можуть бути на найкоротшому шляху, і повторюється процес, поки не відбудеться більше зіткнень.

7. Випадкова миша (*Random mouse*): Для контрасту, існує неефективний метод розв'язання лабіринту, який полягає в русі випадковим чином, тобто виконується рух в одному напрямку через будь-які повороти, поки не буде досягнуто наступного перехрестя. Не виконується жодних поворотів на 180 градусів, якщо не потрібно. Це імітує людину, що випадково блукає по лабіринту без будь-якої пам'яті про те, де вона була.

1.4 Огляд існуючих алгоритмів для генерування лабіринтів

1. Алгоритм бінарного дерева (Binary Tree): Один з небагатьох алгоритмів, що має можливість генерувати ідеальний лабіринт, не зберігаючи жодного стану: це точний алгоритм генерації лабіринту без обмеження, розміру лабіринту, який ви можете створити. Він може побудувати весь лабіринт, дивлячись на кожну клітинку незалежно. Це найпростіший і найшвидший з можливих алгоритмів. Проте лабіринти, які він створює, мають дефекти (довгі коридори, що охоплюють дві

сторони) і помітний нахил (маршрути, як правило, проходять по діагоналі).

2. Алгоритм Крускала (Kruskal's Algorithm): Це рандомізована версія алгоритму Крускала створення мінімального дерева для зваженого графа. Крускал цікавий тим, що він не «вирощує» лабіринт, як дерево, а замість цього навмання вирізує сегменти проходу по всьому Лабіринту. В підсумку виходить ідеальний лабіринт. Алгоритм вимагає багато пам’яті, пропорційного розміру лабіринту. Отримані лабіринти, як правило, мають багато коротких відхилень.

3. Алгоритм Прима (Prim's Algorithm): Рандомізована версія алгоритму Прима - методу створення мінімального охоплюючого дерева з неорієнтованого зваженого графіка. Алгоритм Прима створює дерево, отримуючи сусідні клітини та знаходячи найкращу для переходу. Він вимагає багато пам’яті, пропорційно розміру лабіринту. Отриманий лабіринт має багато прямих відхилень.

4. Алгоритм рекурсивного ділення (Recursive Division): Найшвидший алгоритм без спрямованих схилів. Цей алгоритм є особливо захоплюючим завдяки своїй фрактальній природі: теоретично можно продовжувати процес необмежено довго на точніших рівнях деталізації. Як генератор стінок, процес починається з відкритого простору (усі комірки з’єднані) і додає стіни (від’єднує комірки) до отримання лабіринту.

5. Алгоритм рекурсивного пошуку з поверненням (Recursive Backtracking) - це рандомізована версія алгоритму обходу пошуку за глибиною. Цей підхід є одним з найпростіших способів генерування лабіринту. Необхідно достатньо пам’яті, щоб зберегти в пам’яті весь лабіринт, пропорційний розміру лабіринту, тому для великих лабіринтів цей метод може бути досить неефективним. Генеровані лабіринти мають низьку кількість перехресть та містять багато довгих коридорів, оскільки

алгоритм проходить, наскільки це можливо, уздовж кожної гілки перед поверненням до минулого перехрестя.

1.5 Машинне навчання

Машинне навчання - це метод аналізу даних, який автоматизує побудову аналітичної моделі. Це галузь штучного інтелекту, основана на ідеї, що системи можуть вчитися на основі даних, визначати закономірності та приймати рішення з мінімальним втручанням людини.[19]

Процес навчання починається із спостережень чи даних, таких як приклади, безпосередній досвід чи інструкція, щоб шукати закономірності в даних та приймати кращі рішення в майбутньому на основі прикладів, які ми надаємо. Основна мета - дозволити комп’ютерам навчатися автоматично без втручання та допомоги людини та відповідно коригувати дії.

Основні методи[18]:

- Контрольоване навчання — застосовується у випадках де відомий бажаний результат. Наприклад, фотографії з мітками, які визначають що показано на фото;
- Неконтрольоване навчання — бажаний результат невідомий. Система повинна сама знайти закономірності, виконати класифікацію;
- Напівконтрольоване навчання — присутні як позначені дані так і ні. Зазвичай більша частка даних не помічена;

- Навчання методом проб та помилок — система виконує одні й ті ж самі дії з невеликим відхиленням поки не достигне результату.

Машинне навчання займається побудовою та вивченням систем, які можуть вчитися на даних, а не слідувати лише чітко запрограмованим інструкціям, тоді як розпізнавання образів - це процес розпізнавання образів за допомогою алгоритму машинного навчання. Розпізнавання образів можна визначити як класифікацію даних на основі вже отриманих знань або статистичної інформації, вилученої із зразків та/або їх подання.[20]

1.6 Розпізнавання образів

Крістофер Бішоп у своїй фундаментальній роботі «Розпізнавання образів та машинне навчання» описує розпізнавання образів як: “автоматичне виявлення закономірностей даних за допомогою комп’ютерних алгоритмів та використання цих закономірностей для здійснення таких дій, як класифікація даних у різні категорії”[24]. Іншими словами, розпізнавання образів - це ідентифікація закономірностей у даних.

Самі дані можуть бути будь-якими:

- Текст
- Зображення
- Звук
- Почуття та ін.

Будь-яка послідовна інформація може бути оброблена алгоритмами розпізнавання образів, що робить послідовності зрозумілими та дає можливість їх практичного використання.

Існує три основні моделі розпізнавання образів:

- Статистична: щоб визначити, чому належить конкретний шматок (наприклад, це ліс чи ні). Ця модель використовує контрольоване машинне навчання;
- Синтаксична/структурна: для визначення більш складних взаємозв’язків між елементами (наприклад, частинами мови). Ця модель використовує напівавтоматичне машинне навчання;
- Пошук шаблонів: для зіставлення властивостей об’єкта із заздалегідь визначеним шаблоном. Одним із застосувань такої моделі є перевірка плагіату.

Сам процес виглядає так:

1. Дані збираються з джерел;
2. Дані очищаються від шуму;
3. Інформація перевіряється на наявність відповідних ознак або загальних елементів;
4. Ці елементи згодом групуються в конкретні сегменти;
5. Сегменти аналізуються для знаходження наборів даних;
6. Отримані статистичні дані впроваджуються.[22]

В цілому розпізнавання складається з навчання та розпізнавання[21].

Навчання здійснюється шляхом показу окремих об'єктів з вказанням їх приналежності тому або іншому образу. В результаті навчання розпізнаюча система повинна набути здатність однаково реагувати на всі об'єкти одного образу і по-різному – на всі об'єкти різних образів. За навчанням слідує процес розпізнавання нових об'єктів, який характеризує дії вже навчанної системи. Автоматизація цих процедур і складає проблему навчання розпізнаванню образів.

Методи розпізнавання образів і технічні системи, що реалізують ці методи, широко використовуються на практиці. Деякі з них:

- Технічна діагностика. На виробництві задача полягає в тому, щоб виявити, чи є деталь дефектною, чи ні. Якщо ж з'ясовується, що деталь має дефект, часто потрібно визначити тип цього дефекту.
- Медична діагностика. Найтиповіша ситуація полягає в тому, що ті чи інші захворювання діагностуються на основі аналізу кардіограм, рентгенівських знімків і т. п.
- Розпізнавання літер. Системи розпізнавання літер працюють разом зі сканерами – пристроями, які використовуються для введення до комп'ютера друкованих зображень і текстів.
- Розпізнавання мови. Сьогодні інтенсивно розвиваються технології, пов'язані, по перше, з голосовим керуванням комп'ютером, а по друге – з введенням текстів з голосу.
- Робототехніка. Застосування методів розпізнавання в робототехніці є абсолютно природним і необхідним, оскільки роботи повинні безпосередньо сприймати зовнішній світ і, відповідно, мати пристрої машинного зору.
- Охоронні системи. Застосування методів розпізнавання в охоронних системах пов'язано в першу чергу з проблемою ідентифікації.

1.7 Постановка задачі

Метою роботи є дослідження впливу правдивих та брехливих інструкцій на героя, який проходить лабіринт та передбачення дій гравця. Для цього необхідно:

1. Створити модель гри;

2. Обрати алгоритм створення лабіринту та реалізувати його у середовищі Unity;
3. Обрати алгоритм проходження лабіринту, який буде використано в подальшій роботі;
4. Проаналізувати взаємодію гравця з персонажем гри;
5. Спроектувати вплив брехливих та правдивих інструкцій на довіру героя;
6. Програмно реалізувати обраний алгоритм проходження лабіринту;
7. Інтегрувати в створений алгоритм довіру;
8. Проаналізувати отримані результати роботи алгоритму;
9. Інтегрувати ML-Agents у проект Unity;
10. Виконати навчання за допомогою ML-Agents;
11. Проаналізувати отримані результати.

2 ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ ГРИ

2.1 Загальні відомості

Метою гри є сповільнення проходження героєм лабіринту. На кожному перехресті гравець дає поради персонажу щодо подальшого напрямку. У відповідності до поточного рівня довіри герой буде слідувати порадам гравця або ні.[14]

Для опису можливостей об'єктів гри і гравця побудуємо діаграму використання. Вона описує функціональність і поведінку проектованої системи.

Акторами створюваної діаграми будуть:

• Герой - персонаж гри;

• Оточення – стіни лабіринту;

• Користувач (гравець) - людина.

Герой гри має можливість переміщатися по лабіринту. Він може довіряти гравцю та використовувати його поради для пошуку виходу. У тому випадку якщо герой помітить що гравець намагається заплутати його, він почне самостійно шукати вихід.

Спроектовану діаграму використання подано на рис. 2.1.

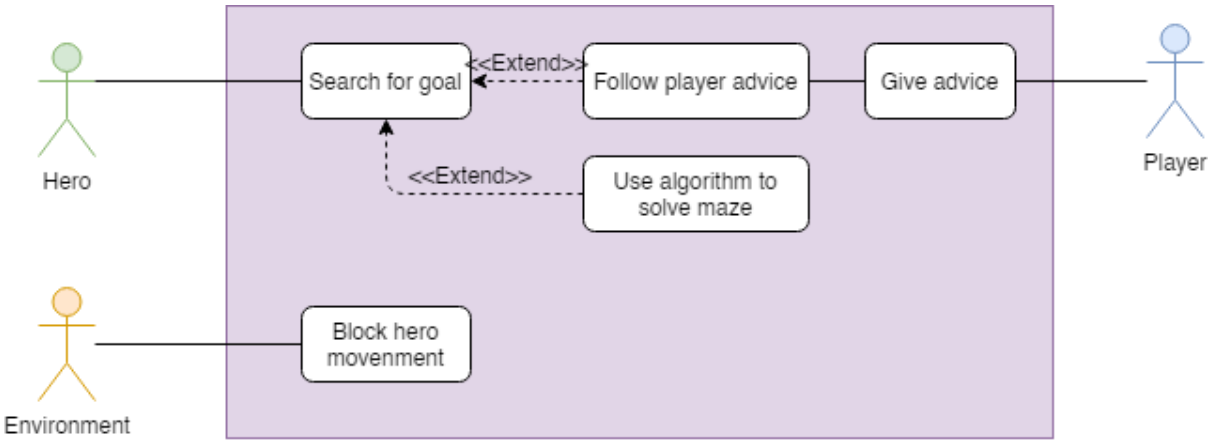

Рисунок 2.1 - Діаграма використання

На діаграмі описано дії героя - пошук виходу використовуючи поради гравця або ігноруючи їх та дії людини - взаємодія з персонажем.

2.2 Алгоритм рекурсивного пошуку з поверненням

В роботі буде використаний даний алгоритм для створення лабіринту. Алгоритм виконує наступні кроки:

1. Обирається початкова точка входу лабіринту;
2. Випадково обирається стіна в цій точці та створюється в ній прохід до сусідньої клітинки, але лише якщо алгоритм її ще не відвідував. Цей прохід стає новою поточною точкою;
3. Якщо всі сусідні проходи були відвідані, виконується повернення до останнього проходу, що має невідвідані сусідні клітинки і повторюється крок 2;
4. Алгоритм закінчується, коли процес відвідав усі клітинки і повернувся до початкової точки.

Цей алгоритм не є найефективнішим для генерування лабіринту, але в результаті він створює лабіринти з довшими шляхами, які є більш привабливими та складнішими для вирішення. Вони також є повністю

зв’язані, тобто можна пройти з будь-якої клітини до будь-якої іншої клітини.

Приклад створення лабіринту описано далі.

На початку виконання алгоритму обирається випадкова точка початку лабіринту, на рис 2 її зображено світло сірим кольором.

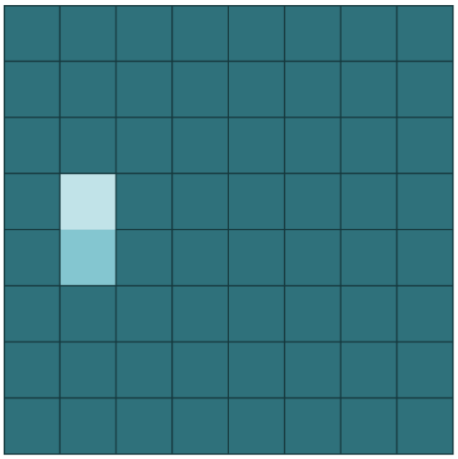

Рисунок 2.2 - Початок алгоритму

Далі виконується рандомне пробиття проходів поки алгоритм не дійде то проходу, з якого неможливе подальше переміщення. На рис 3 ця клітинка позначена білим кольором. Через специфіку дії алгоритму створюються довгі шляхи з невеликою кількістю перехресть.

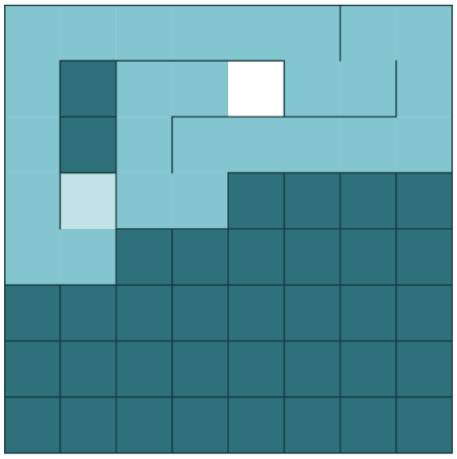

Рисунок 2.3 – Тупиковий прохід

Наступним кроком роботи алгоритму виконується повернення пройденим шляхом до проходу, з якого можливе подальше переміщення до ще невідвіданого проходу (рис 4). Далі повторюється минулий крок пробиття проходів.

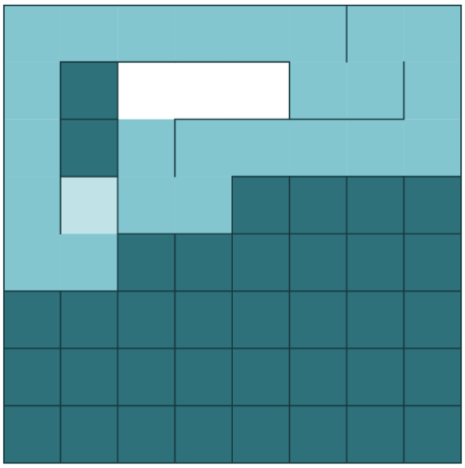

Рисунок 2.4 – Повернення

Подальше виконання алгоритму повторює ці дії доки всі клітинки не будуть відвідані. На рис 5 показано крок подальшої роботи алгоритму.

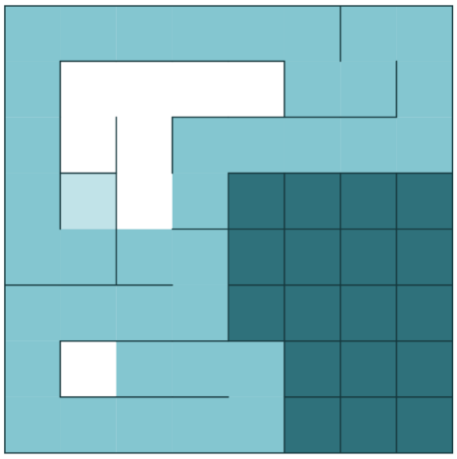

Рисунок 2.5 – Робота алгоритму

В результаті буде створений ідеальний алгоритм (рис 6). З кожної точки лабіринту можна пройти у будь-яку іншу точку.

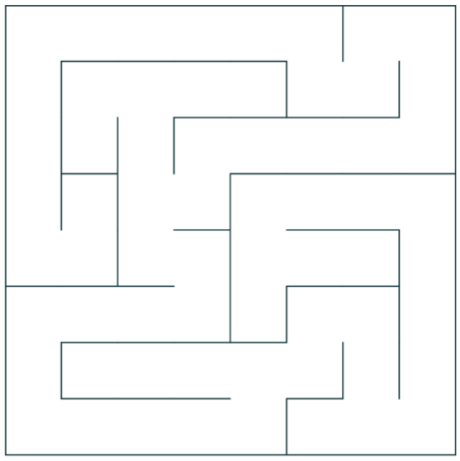

Рисунок 2.6 – Результат

Останнім кроком буде вибір входу та виходу з лабіринту. Початком можна залишити точку обрану на початку алгоритму. Проте для подальшого використання лабіринту в роботі буде створено вхід та вихід на краях отриманого лабіринту.

2.3 Алгоритм Трема

В роботі буде використаний даний алгоритм для проходження лабіринту. Його суть полягає в тому, що пройдені проходи маркуються - 1. При потраплянні на нове перехрестя обирається випадковий шлях. При потраплянні до глухого кута або до вже відвіданого проходу відбувається повернення до минулого перехрестя а пройдені проходи маркуються - 2. В кінці, якщо існує вихід з лабіринту, правильний шлях матиме маркування

1. Якщо ж виходу немає то алгоритм повернеться до початкової точки, з кожним проходом маркованим 2.[10]

В цей алгоритм досить легко вписати “довіру”, він працює для усіх видів лабіринтів. Достатньо замінити випадковий вибір шляху на перехресті певною функцією, яка взаємодіятиме з поточним рівнем довіри.

Приклад проходження лабіринту даним методом можна побачити на наступних схемах.

На початку алгоритму точка знаходиться на вході лабіринту (рис 7).

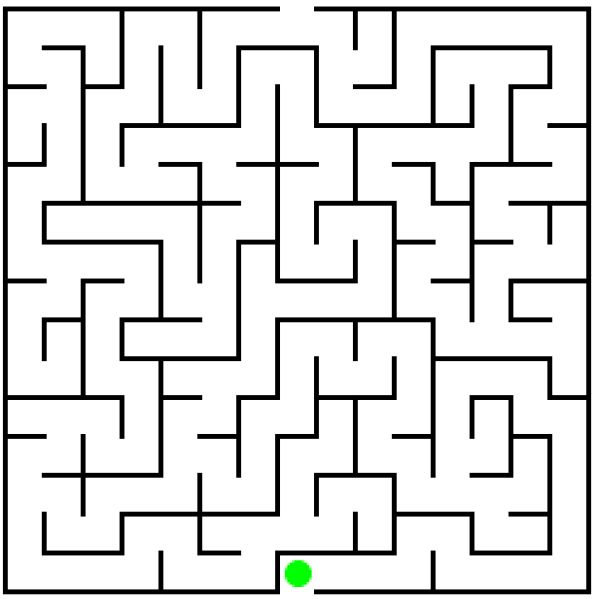

Рисунок 2.7 - Початок алгоритму

Далі виконується переміщення до першого перехрестя, зазвичай методом слідування лівої стіни (рис 8). В коді зручно помічати дані клітинки цифрою 1 – кількість разів алгоритм проходив даним проходом.

Рисунок 2.8 - Перше перехрестя

На даному етапі, якщо жоден з пов’язаних шляхів раніше не був відвіданий, наступний напрям обирається випадково. В розроблюваній системі саме на даний вибір впливає довіра. В прикладі алгоритм потрапив у глухий кут (рис 9).

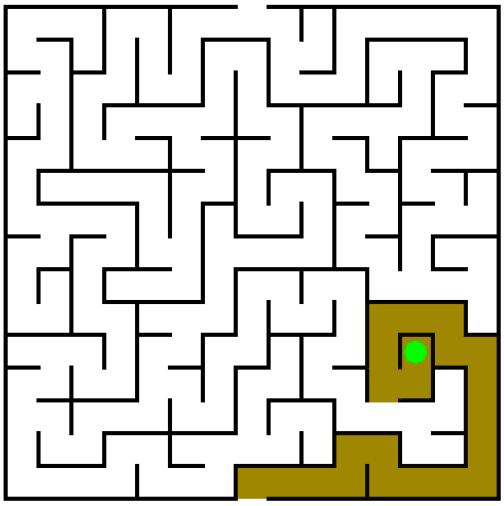

Рисунок 2.9 - Глухий кут

Далі алгоритм повертається назад, маркуючи пройдений шлях як неправильний (рис 10). В коді виконується маркування цифрою 2.

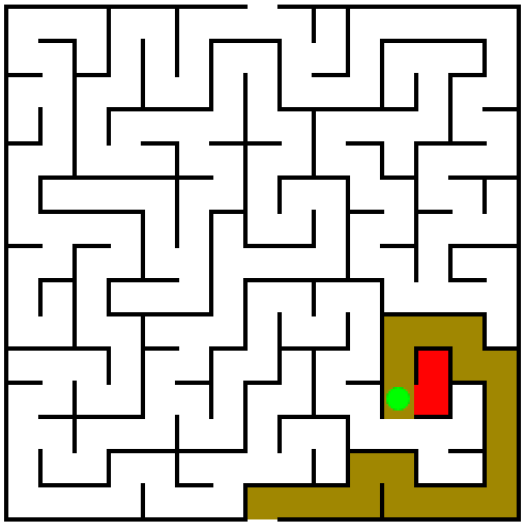

Рисунок 2.10 - Шлях назад

На рис 11-12 показано як алгоритм продовжує переміщення лабіринтом. Таким чином поступово заповнюється весь лабіринт. В прикладі алгоритм більш менш правильно обирає шлях до виходу.

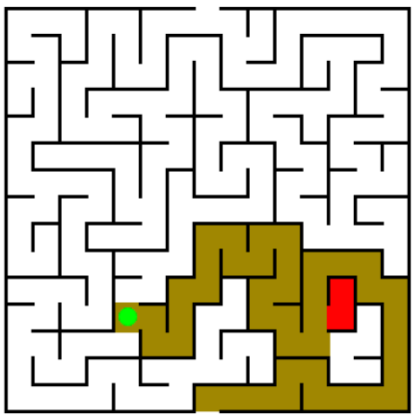

Рисунок 2.11 - Продовження шляху

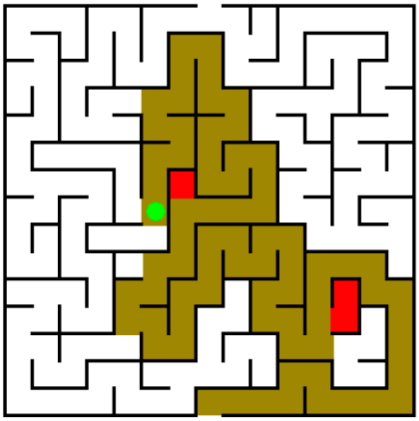

Рисунок 2.12 - Наступне перехрестя

В найгіршому сценарії алгоритм буде змушений пройти через кожен прохід в лабіринті. Далі алгоритм заповнює як неправильний шлях не тільки глухі кути, а і сам шлях до них. Ця дія буде виконуватися рекурсивно, незалежно від кількості пройдених неправильних перехресть. Приклад роботи цієї дії зображено на рис 13.

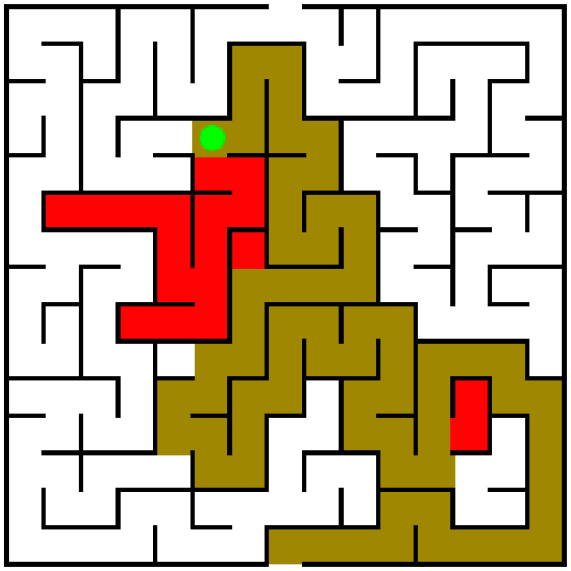

Рисунок 2.13 - Повернення на перехрестя

Останньою дією алгоритму буде маркування самого шляху від початку до кінця лабіринту. Правильними проходами будуть ті, які були пройдені лише 1 раз – в коді це будуть клітинки зі значенням 1. Повну роботу алгоритму зображено на рис 14.

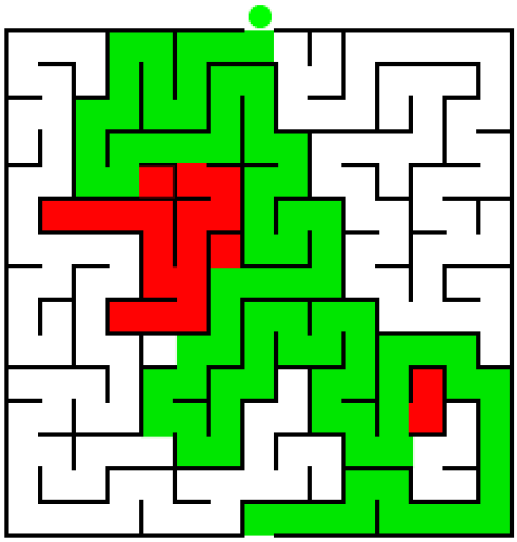

Рисунок 2.14 - Вихід

Через те, що цей алгоритм дуже схожий на те як людина проходить лабіринт, саме він буде використаний в подальшій розробці. Хоча більшість людей проходить лабіринт методом слідування стіни, в цей метод недоцільно включати інструкції людини алгоритму. Після кожної

втрати довіри людині цей метод починатиме свої дії спочатку, вважаючи поточну клітинку стартом алгоритму. Якщо ж дообробляти алгоритм, то він почне все більше і більше бути схожим на метод Трема.

2.4 Рефлексивні системи

Рефлексивне управління – це вплив на прийняті противником рішення через нав'язування йому таких вихідних даних, на підставі яких він діє бажаним для маніпулятора чином[9]. Найбільш помітні рефлексивні процеси саме у конфліктних ситуаціях. Прикладом використання даного управління є дезінформація під час війн, навмисний програш шулерами в іграх, здача позицій у шахах. В роботі використано саме даний метод для розробки взаємодії гравця з персонажем.

Рефлексія – це здатність людини або штучного інтелекту стати на чужу позицію або піднятися над позиціями для розуміння ситуації з усіх боків. Відповідно, метою персонажа гри є розуміння цілей та дій гравця. Герой знає, що людина намагатиметься сповільнити його просування в лабіринті і тому він має реагувати відповідно до цього.

Зобразимо символом T лабіринт. T_x та T_y – те, як лабіринт бачить гравець та герой відповідно. Отриману рефлексивну систему можна представити у вигляді суми:

$$Q_1=T+T_x \quad (1)$$

Система містить два компоненти: лабіринт та позицію гравця щодо цього лабіринту. Врахуємо також позицію персонажа щодо позиції гравця:

$$Q_2=T+T_x+T_y(T+T_x) \quad (2)$$

Сума у скобках відповідає тому, як герой сприймає лабіринт та позицію гравця щодо лабіринту. Для більш зручного розуміння можна розкрити скобку:

$Q_2=T+ T_x+ T_y+ T_{xy}$ (3)

В результаті отримано опис системи та взаємодій у ній. Дана рефлексивна система складається з самого лабіринту, його зображень в розумінні героя та гравця та позиції героя щодо розуміння гравцем лабіринту. Саме T_{xy} і є довірою героя в системі.

2.5 Машинне навчання у Unity 3d

Unity випустила першу версію Unity Machine Learning Agents Toolkit (ML-Agents) у 2017 році[25].

Мета цього середовища ML - дозволити розробникам ігор та дослідникам ШІ використовувати Unity як платформу для навчання, а також вбудовувати інтелектуальні агенти за допомогою останніх досягнень ML та AI.

Набір інструментів машинного навчання Unity або просто ML-Agents - це проект Unity з відкритим кодом, який дозволяє іграм та симуляціям служити середовищем для навчання інтелектуальних агентів. ML-Agents включає набір для розробки програмного забезпечення C# (SDK) для створення сцени та визначення агентів у ній, а також найсучаснішу бібліотеку ML для підготовки агентів для 2D, 3D та VR / AR середовищ.

Агенти Unity ML-включають ряд інтуїтивних функцій. Деякі з них[26]:

- ML-Агенти забезпечують підтримку декількох конфігурацій середовища, включаючи сценарії навчання;
- Механізм самостійної гри для підготовки агентів у змагальних сценаріях;

- Навчання використовуючи рандомізацію середовища;
- Можливість тренування використовуючи кілька одночасних екземплярів середовища Unity;
- Управління середовищем Unity через Python.

Навчальне середовище має три головні види об’єктів:

1. Агент - кожен Агент може мати унікальний набір станів і спостережень, робити унікальні дії в середовищі та отримувати унікальні винагороди за події в середовищі. Дії агента визначає мозок, з яким він пов’язаний;
2. Мозок - кожен Мозок визначає конкретний стан та простір дій і відповідає за дії, які буде виконувати кожен з його пов'язаних агентів. Підпримується встановлення Мозку у одному із чотирьох режимів:
   - Зовнішній - рішення про дії приймаються за допомогою бібліотеки TensorFlow шляхом спілкування через відкритий сокет з зовнішнім Python API;
   - Внутрішній - рішення про дії приймаються за допомогою навченої моделі, вбудованої в проект через бібліотеку TensorFlowSharp;
   - Гравець - рішення щодо дій приймаються за допомогою вводу гравця;
   - Евристичний - рішення про дії приймаються за допомогою кодованої поведінки(алгоритм).

3. Академія - об'єкт Академії в межах сцени також містить у собі як дітей всі Мізки у середовищі. Кожне середовище містить єдину Академію, яка визначає сферу середовища з точки зору:

    - Конфігурація движка - швидкість і якість рендерингу ігрового механізму як в режимы навчання, так і в результаты навчання;
    - Frameskip - скільки кроків двигуна потрібно пропустити між кожним агентом, який приймає нове рішення;
    - Глобальна тривалість епізоду - скільки триватиме епізод. Після досягнення всі агенти готові.

Окрім гнучких сценаріїв навчання, що стали можливими завдяки системі Академія / Мозок / Агент, набір інструментів Unity ML-Agents також включає інші функції, які покращують гнучкість та зрозумілість навчального процесу.

Моніторинг прийняття рішень агентом - Оскільки спілкування в наборі інструментів Unity ML-Agents є двостороннім, існує клас моніторингу агентів в Unity, який може відображати такі аспекти навченого агента, як лінія поведінки та вихідні дані в самому середовищі Unity.

Навчання за навчальною програмою - це процес поступового збільшення складності завдання для забезпечення більш ефективного навчання. Набір інструментів Unity ML-Agents підтримує встановлення власних параметрів середовища кожного разу, коли середовище оновлюється. Це дозволяє динамічно коригувати елементи середовища, пов’язані зі складністю, залежно від прогресу навчання.

Навчання за допомогою імітації - частіше буває більш інтуїтивно зрозумілим просто демонструвати поведінку, яку ми хочемо виконувати, а

не намагатися змусити агента навчитися методом спроб і помилок. Застосовуючи імітаційне навчання, гравець може продемонструвати, як агент повинен поводитися в навколишньому середовищі, а потім використати ці демонстрації для підготовки агента, або як перший крок у процесі навчання з підкріплення.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ

3.1 Створення лабіринту

За основу для створення лабіринту було обрано алгоритм рекурсивного пошуку з поверненням. Для зручності та привабливості лабіринту було додано наступні зміни до даного алгоритму:

- Замість стандартних клітинок з 4 стінами буде використано два типи клітинок: підлога та стіна. Це надає змогу більш зручно використовувати текстури при роботі з лабіринтом та зробить лабіринт більш легким для розуміння;
- Початковою точкою лабіринту за замовчуванням буде клітинка з позицією (1,1), проте можна обрати стартом іншу позицію вручну або за допомогою відповідного методу. Кінцева точка обиратиметься за допомогою відстані від початкової.

Алгоритм обирає випадкову точку на відстані 2 клітинок від поточної, якщо ця точка ще не була відвідана і лежить у межах лабіринту, пробивається прохід до даної клітинки. Далі рекурсивно виконується цей крок доки усі клітинки не будуть відвідані. Це виконується за допомогою функції MazeDigger(x, y), яка приймає на вхід поточну клітинку. Код функції знаходиться в додатку.

Для графічного відображення було створено спеціальний prefab(збірка спрайтів) у якому зберігаються текстури лабіринту. Текстури, використані у проекті зображено на рис 3.1. Кожна текстура має розмір 128x128.

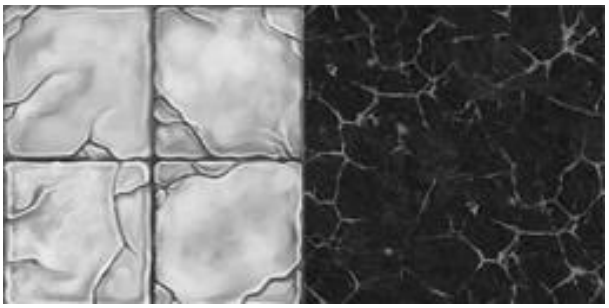

Рисунок 3.1 - Текстури

Для відображення лабіринту також було створено спеціальні ігрові об’єкти для зберігання текстур та самого лабіринту. До об’єкта лабіринту прикріплено скрипт MazeGenerator.cs (знаходиться в додатку), у якому виконується заповнення екрану текстурами у відповідності до раніше згенерованого лабіринту. На рис 3.2 знаходиться панель з відповідними опціями для зміни висоти та ширини лабіринту, текстур та prefab. Значення Maze Seed відповідає за випадкове генерування лабіринту і дозволяє запам’ятовувати згенеровані лабіринти.

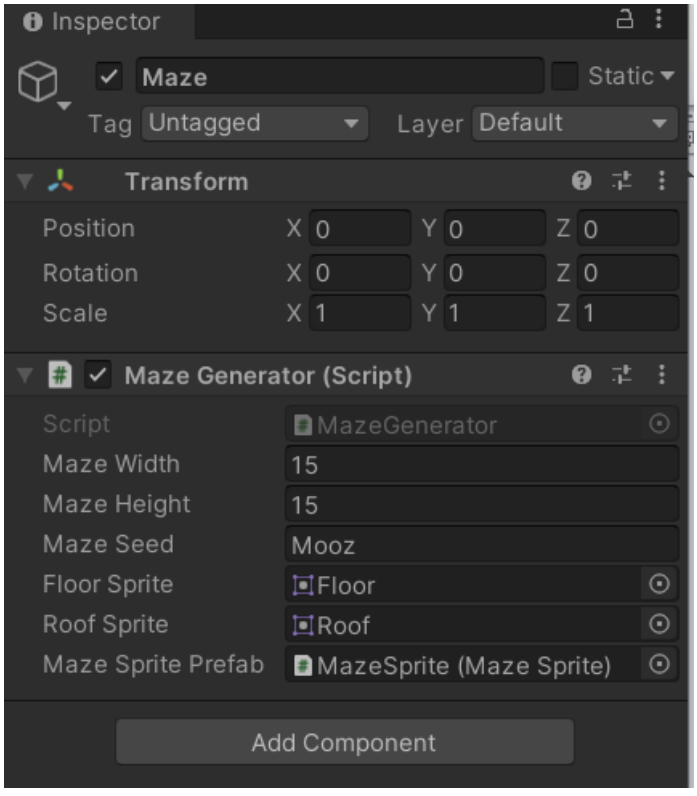

Рисунок 3.2 – Панель генерування лабіринту

Результат роботи алгоритму створення лабіринту зображено на рис 3.3.

Рисунок 3.3 – Отриманий лабіринт

В результаті було згенеровано ідеальний лабіринт з досить довгими переходами та достатньою кількістю розвилок для подальшої роботи.

3.2 Герой лабіринту

Наступним кроком створюється алгоритм проходження лабіринту. Спочатку необхідно створити відповідний об’єкт, який буде проходити лабіринт, прикріпити до нього текстуру та скрипт. Для відображення героя використано простий білий ромб.

За основу було обрано алгоритм Трема для проходження лабіринту у який було додано концепцію довіри. Переміщення персонажа ділиться на декілька видів руху:

- Пряме переміщення між двома перехрестями. Цей рух відбуватиметься після вибору шляху персонажем і триватиме доти, доки кожний прохід, через який переміщується герой, матиме тільки 2 точки входу – через одну персонаж потрапив у даний прохід а через іншу

він перейде в наступний. В даному випадку переміщення буде відбуватися в одну сторону до наступного перехрестя. Не важливе положення входів-виходів кожного проходу.

- Глухий кут. Цей рух відбувається коли герой потрапляє в прохід з лише однією точкою виходу. В даному випадку герой повертає назад і прямує прямим переміщенням до останнього перехрестя, якщо усі інші шляхи крім одного на тому перехресті також ведуть до тупика, то персонаж продовжує рух назад до наступного перехрестя.
- Перехрестя. В даному випадку прохід має мати 3 або 4 точки виходу. На даному етапі гравець дає пораду щодо подальшого шляху героя. Якщо обраний шлях є відомим персонажу глухим кутом то рівень довіри падає а герой обирає інший шлях (відмінний від того, яким він прийшов). Якщо герою не відомо що знаходиться обраним шляхом то далі вступає в силу рівень довіри героя.

Було перевірено правильність роботи даного алгоритму та гри в цілому при різних згенерованих лабіринтах. В результаті роботи, алгоритм завжди правильно проходив лабіринт.

Для розуміння процесу роботи алгоритму було створено карту, на якій схематично відображено прогрес алгоритму. Приклад проходження алгоритмом лабіринту зображено на рис 3.4.

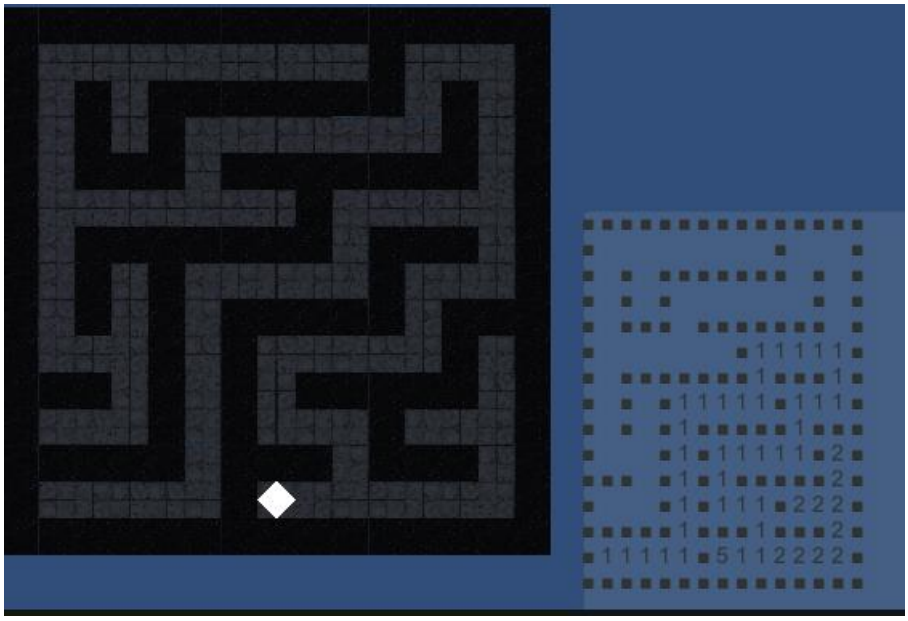

Рисунок 3.4 – Проходження лабіринту

3.3 Концепція довіри

Ключовою ідеєю гри є імплементація довіри в алгоритм. Так як ціль гравця якнайдовше водити персонажа лабіринтом, герой має самостійно приймати рішення – довіряти вказівкам гравця або ні.

Впливати на рівень довіри персонажа будуть кожні дії гравця. Так, якщо користувач заведе героя в глухий кут, наступного разу шанс того, що персонаж довіриться гравцю буде менший. Так само будуть впливати і позитивні дії людини. Наприклад, якщо герой не повірив гравцю і потрапив у глухий кут, то він зрозуміє що гравець не хотів заплутати його.

На кожному кроці алгоритму виконується випадкова генерація числа від 0 до 100. Для того, щоб алгоритм послухав пораду гравця та виконав переміщення обраним шляхом необхідно щоб отримане число входило до інтервалу довіри. Початковий інтервал довіри зображено на рис. 3.5.

Початковий інтервал може змінюватися в залежності від заданих параметрів. Чим менший інтервал тим менше алгоритм віритиме інструкціям гравця.

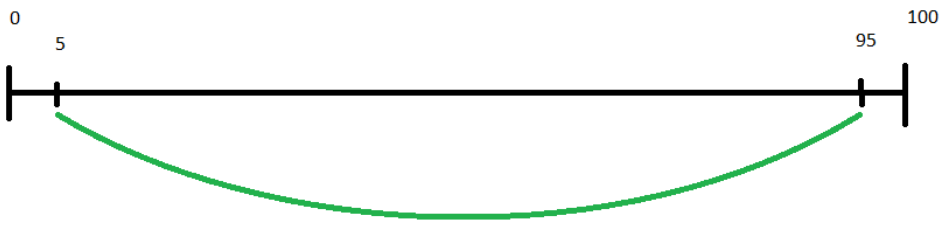

Рисунок 3.5 – Початковий стан довіри

Нижче приведено дії які негативно впливають на довіру героя.

Якщо гравець завів персонажа до глухого куту або до місця, яке вже було відвідане, то рівень довіри впаде. Наскільки багато залежить від того відомий правильний шлях людині чи ні. Якщо ні то існує шанс що гравець сам не знає куди необхідно піти щоб вийти з лабіринту. Так, з більшим розміром самого лабіринту людині буде складніше самій знаходити правильний шлях, навіть маючи повну карту лабіринту.

Набагато більше довіри буде втрачено коли гравець покаже на вже відомий герою тупик. Це лише підтвердить його недовіру, адже персонаж пам’ятає всі шляхи якими він ходив. Для того, щоб людина випадково не втратила довіру необхідно їй також бачити усі пройдені героєм проходи. Ця функція може бути використана людиною для того, щоб показуючи на правильний шлях, вести героя у інший бік. Проте слід звернути увагу, що після певного часу блукань персонаж почне повністю самостійно приймати рішення куди йому йти, незважаючи на довіру гравцю.

Якщо гравець вестиме героя тим же шляхом яким він вже проходив то рівень довіри буде поступово падати. Персонаж може слухатися людини доки не дійде до нового перехрестя а далі слідувати своїм думкам. Також він може відразу намагатися вийти на нові шляхи.

Зміну інтервалу довіри у відповідності до дій гравця зображено на рис 3.6. Різні дії по різному впливають на зменшення інтервалу.

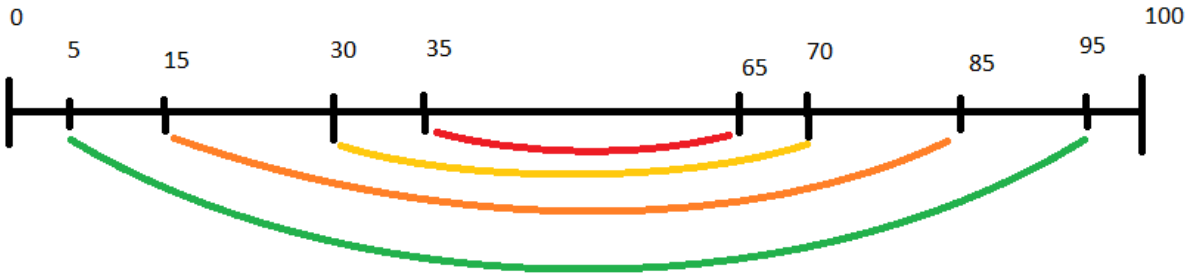

Рисунок 3.6 – Вплив негативних дій на довіру

Втратити довіру досить легко, існує багато способів для цього. Відновити її буде набагато складніше. Так само як і в реальному житті, втратити довіру набагато легше ніж отримати її.

Дії які позитивно впливають на довіру героя:

- Якщо герой не довірився гравцю та потрапив у глухий кут, то рівень довіри зросте. В даному випадку персонаж знатиме що людина не хотіла щоб він заплутав, хоча б на цій ділянці. Проте якщо в іншій стороні також був глухий кут, про який знає герой, то довіри людина не отримає а лише втратить її ще більше.

Як можна побачити, способів втратити довіру набагато більше ніж можливостей отримати її. Це можна виправити пасивним бонусом до довіри, який буде отриманий на кожному перехресті. Проте це не дуже

схоже на реакцію людини в подібній ситуації. Людина навпаки мала би менше довіри до того, хто вестиме її. Особливо якщо вона знатиме що ціль ведучого якомога довше плутати її в лабіринті. Тому не буде використано даний пасивний бонус в грі.

Взаємодія з рівнем довіри, його підвищення та пониження відбувається у функції trustUser.

3.4 Короткий опис програмної реалізації

В ході розробки було створено структуру об’єктів гри, яку зображено на рис 3.7. Кожен об’єкт пов’язаний з відповідним скриптом або текстурою.

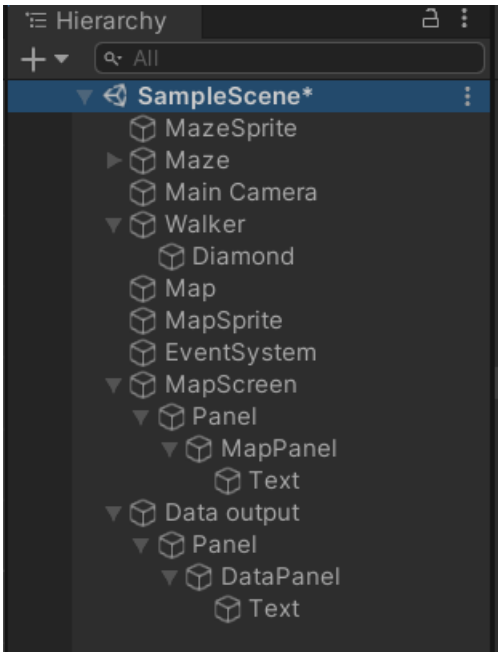

Рисунок 3.7 – Структура об’єктів

У грі використано 5 класів. Діаграму класів проекту подано на рис 3.8.

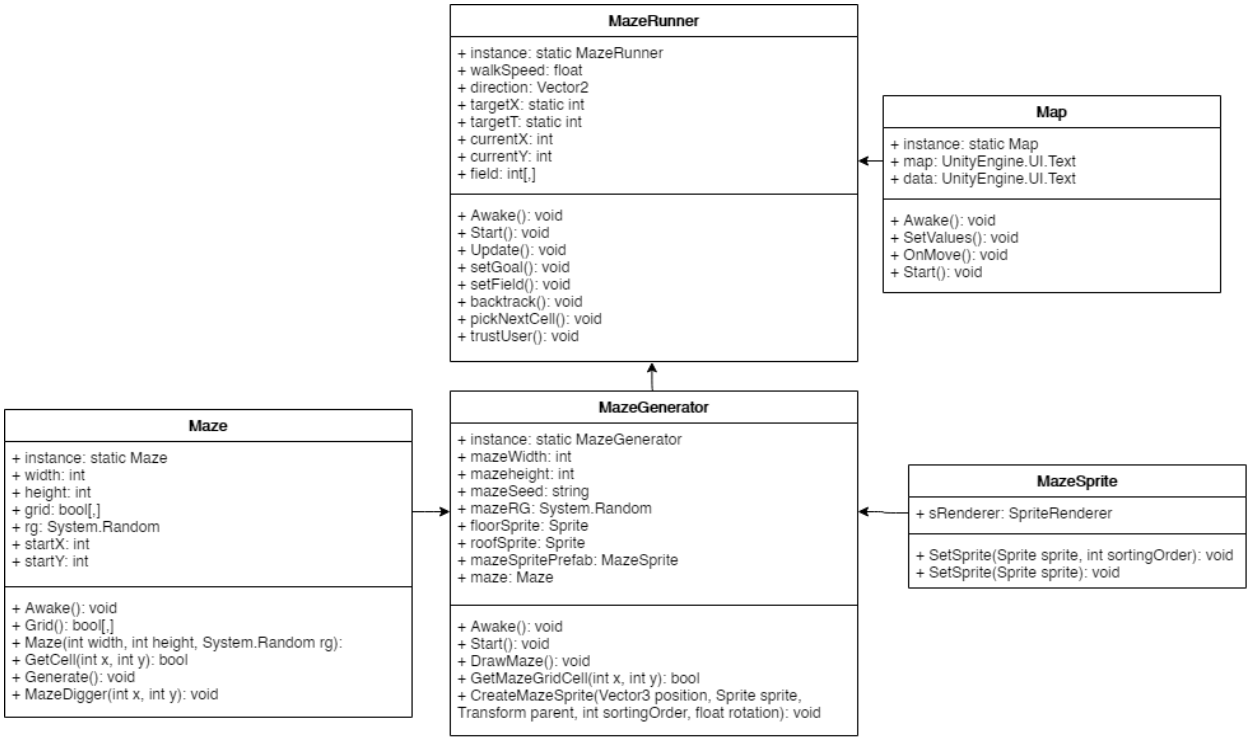

Рисунок 3.8 – Діаграма класів

Загальні данні щодо кожного класу описано в табл 3.1. Код приведено в додатку.

Таблиця 3.1 – Класи проекту

Назва	**Вхідні данні**	**Опис**
MazeSprite	—	Використовується для ініціалізації рендера текстур
Maze	—	Генерує сітку з описом лабіринту, викликається з MazeGenerator
MazeGenerator	MazeWidth, MazeHeight, MazeSeed, FloorSprite, Roof Sprite,	Виконує генерацію випадкового лабіринту з відповідними значеннями висоти та ширини на основі змінної MazeSeed. Вхідні спрайти використовуються при генерації візуального

	MazeSpritePrefab	зображення лабіринту.
MazeRunner	WalkSpeed, GoalX, GoalY	Клас алгоритму проходження лабіринту. Виконується переміщення героя, його взаємодія з гравцем. Змінні GoalX та GoalY використовуються для задання кастомної точки цілі лабіринту. За замовчуванням ця точка обирається автоматично відповідною функцією.
Map	Map, Data	Використовується для відображення схеми прогресу алгоритму та виводу загальним даних. Вхідними змінними являються відповідні текстові поля.

3.5 Приклад роботи гри

Для перевірки правильності роботи ігрової системи була створена нова сессія. На рис 3.9 подано початкове становище середовища розробки. Для тестування роботи системи було обрано розмір лабіринту 15x15, швидкість переміщення персонажу – 7. Стартову позицію залишено без змін (1,1).

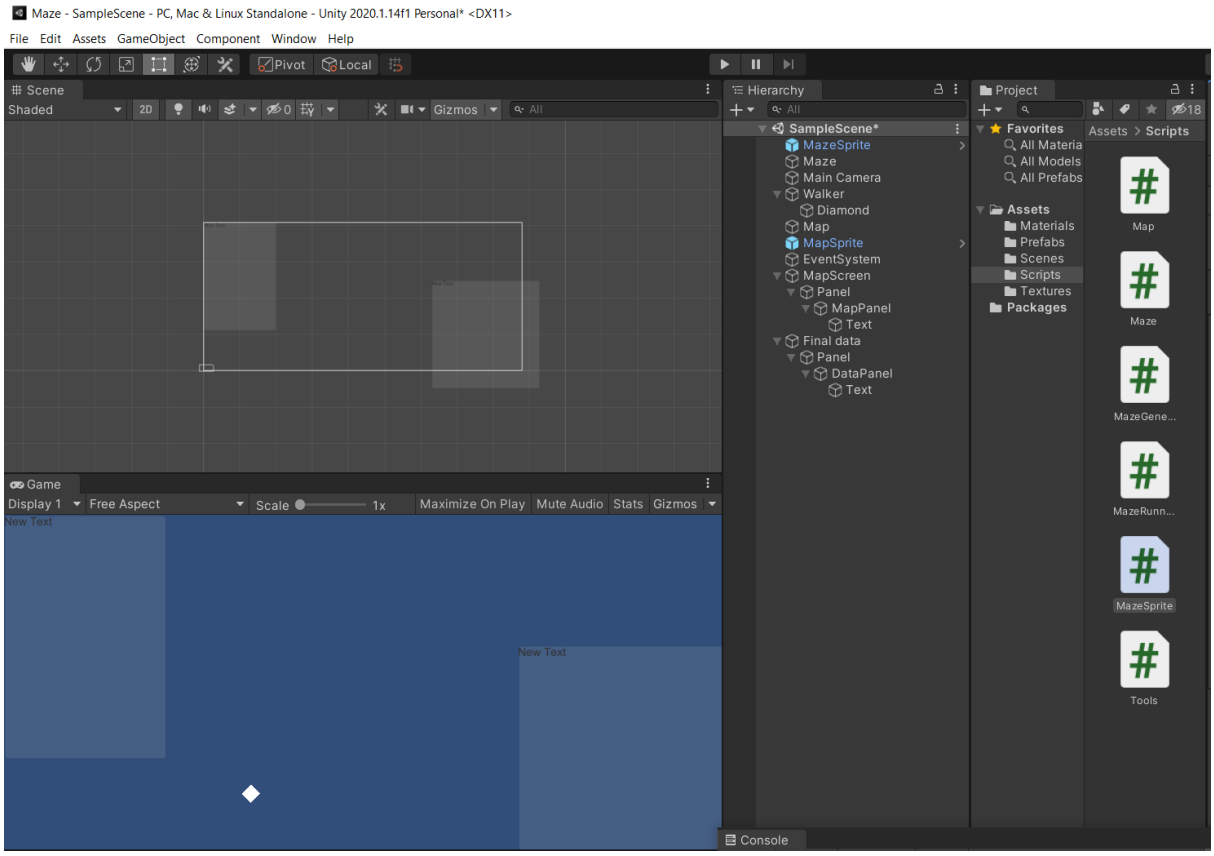

Рисунок 3.9 – Середовище розробки

Першим кроком після старту проекту є генерація та відображення лабіринту. В результаті отримано лабіринт, зображений на рис 3.10.

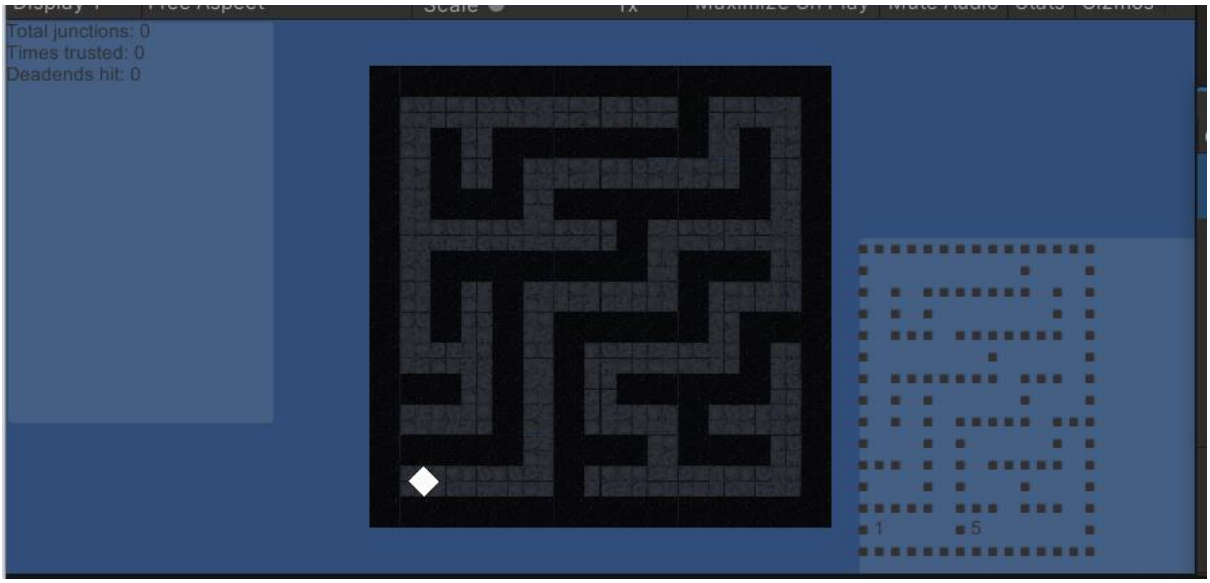

Рисунок 3.10 – Генерування лабіринту

Так як з поточної точки можливе переміщення лише в одному напрямку, алгоритм автоматично виконує рух до першого перехрестя. Зупинку алгоритму на перехресті зображено на рис 3.11.

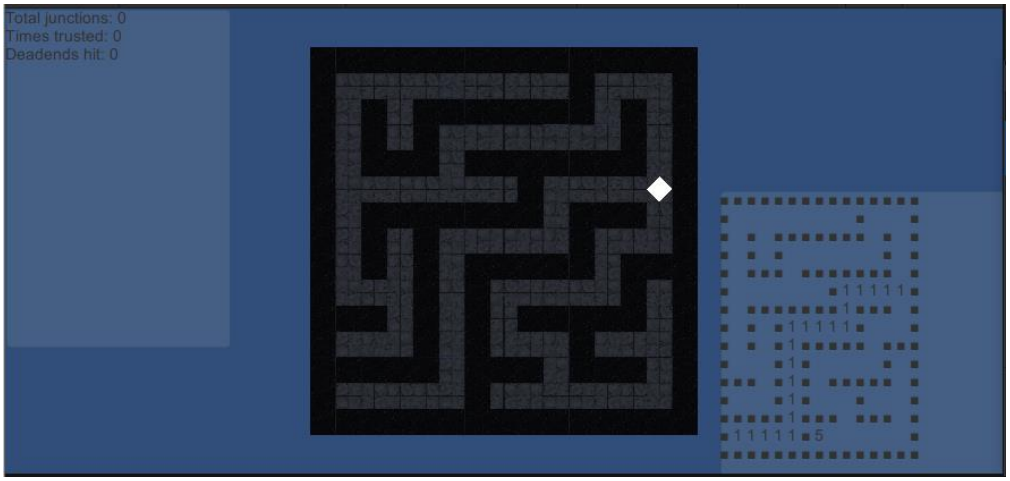

Рисунок 3.11 – Перше перехрестя

Для подальшого заплутування алгоритму була натиснута клавіша W для вибору проходу який не веде до цілі, проте герой вирішив не довіряти інструкції і пішов донизу. Це зображено на рис 3.12. Системне повідомлення зліва внизу (“Not your way”) викликається саме у разі недовіри героя.

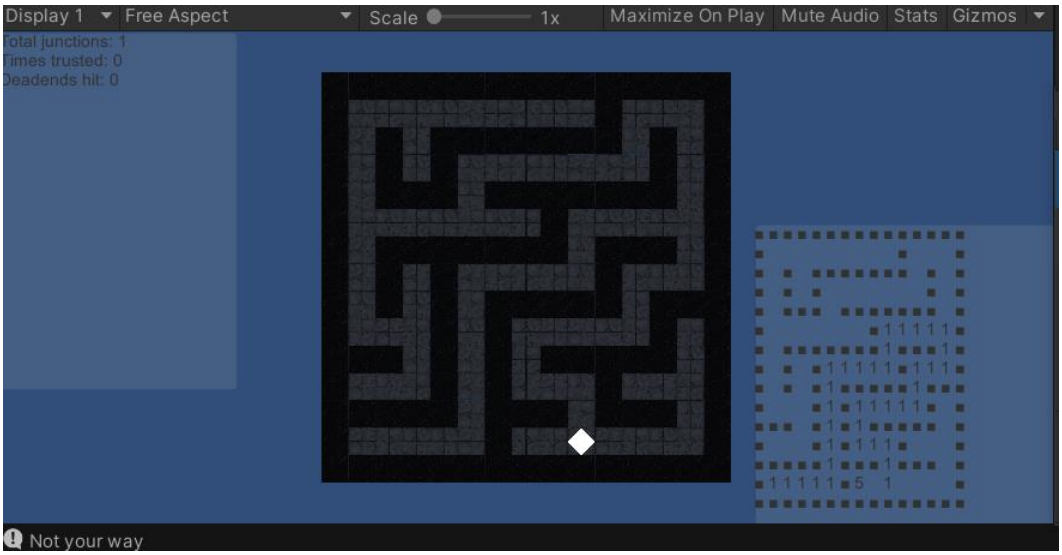

Рисунок 3.12 – Приклад недовіри

Наступним кроком була виконана наступна спроба заплутати алгоритм і цього разу герой вирішив довіритися. Результат показано на рис 3.13.

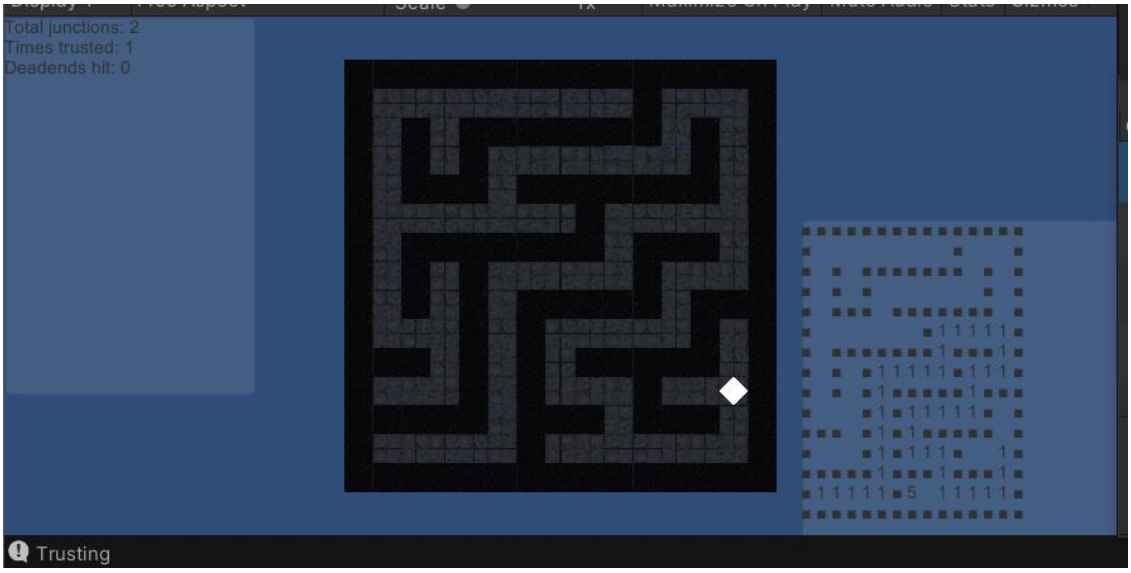

Рисунок 3.13 – Приклад довіри

Після останнього перехрестя алгоритм пройшов єдиним можливим шляхом до цілі. В результаті алгоритм пройшов лабіринт за 3 кроки з 7 максимально можливих. Отриманий результат підтверджує ефективність створеного алгоритму. Кінцевий стан системи зображено на рис 3.14.

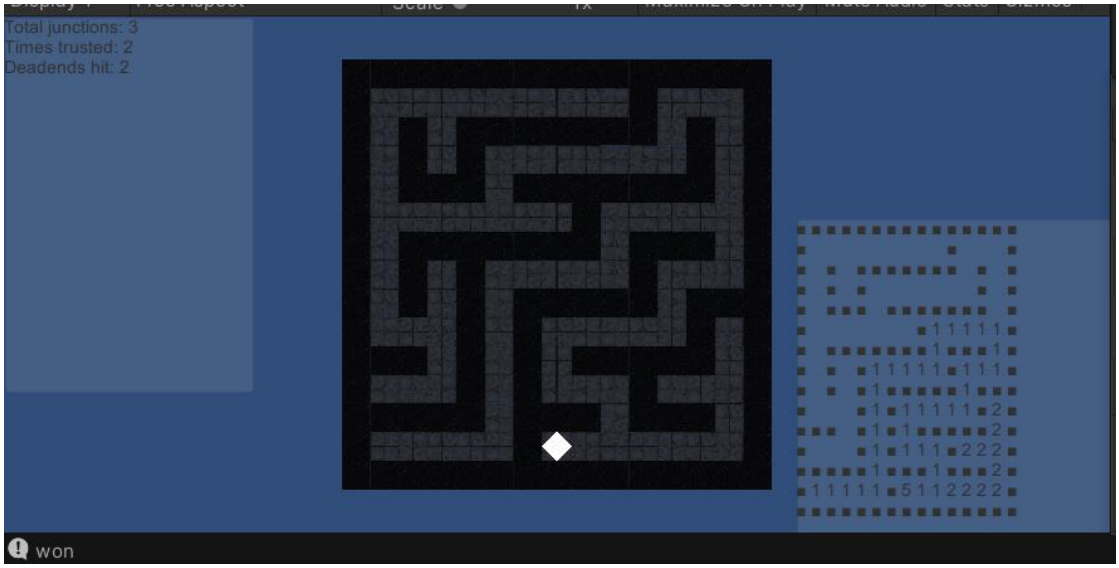

Рисунок 3.14 – Кінцевий стан

3.6 ML – agents

ML-agents у Unity дають змогу легко виконувати навчання системи у будь-якому середовищі. Загалом вони працюють за методом чорної коробки: агенту надаються деякі значення, отримані із взаємодії з середовищем, він їх обробляє і повертає результат. Цей отриманий результат далі використовується в алгоритмі і в залежності від впливу переданих даних на прогрес алгоритму агент отримує винагороду. В залежності від отриманої винагороди агент зрозуміє правильно він відреагував чи ні та змінить свою поведінку відповідно.

У випадку з лабіринтом достатніми вхідними даними будуть: поточне положення у лабіринті та напрям, який дає гравець. В результаті роботи агента буде отримано рішення довіряти гравцю чи ні. Отримане значення буде використано в алгоритмі і у відповідності с результатом подальшої його роботи відповідна нагорода буде надана агенту.

Значення нагороди обиратиметься за тим же принципом, яким змінюється інтервал довіри. Так, при потраплянні у глухий кут слідуючи інструкції гравця агент втратить певну кількість балів так як це рішення було неправильним. Єдиною відмінністю є те, що агент отримає позитивні бали тільки при знаходженні цілі так як його мета – завершення лабіринту з максимальною кількістю балів.

Для роботи агента створено спеціальний клас RunnerAgent. Для взаємодії з агентом використовуються наступні методи:

- CollectObservations – метод передає відповідні спостереження над навколишнім середовищем. В даному випадку це поточне положення та напрям гравця;

- OnActionReceived – метод приймає значення з агента для подальшого використання в алгоритму, також в даному методі виконується надання винагороди;
- MoveAgent – даний метод викликається з OnActionReceived. У ньому виконується подальше переміщення в лабіринті у відповідності до отриманого напряму. При потраплянні до розвилки знову викликається метод CollectObservations і цикл повторюється поки алгоритм не дійде до цілі.

Також важливо правильно виставити настройки параметрів класу (рис 3.15). Тут Space Size у Observation відповідає за кількість вхідних параметрів (передається дві позиції, кожна позиція має значення x та y – у результаті 4 значення), а Space Size у Action відповідає за кількість отримуваних значень (довіряти або ні – 1 значення).

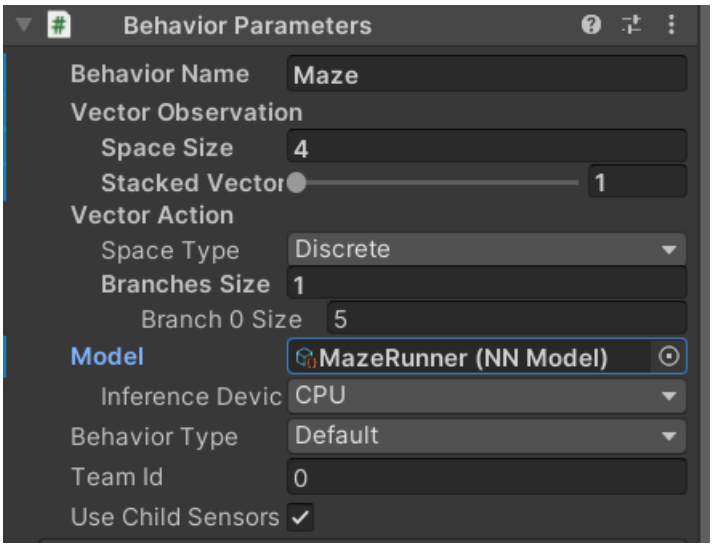

Рисунок 3.15 – Параметри агента

3.7 Аналіз отриманих результатів

Після закінчення навчання будо проведено аналіз роботи отриманої моделі та виконано порівняння її результатів з результатами роботи базового алгоритму.

Для початку було проаналізовано роботу алгоритму при проходженні однакового лабіринту при різних стартових значеннях рівня довіри. Для кожного рівня було зроблено 10 тестів. Результати показано на табл 3.2.

Таблиця 3.2 – Відношення кроків до рівня довіри

Початковий рівень довіри	**Мінімальна к-сть. кроків**	**Середня к-сть. кроків**	**Максимальна к-сть. кроків**
5-95	7	16	25
20-80	5	11	23
35-65	12	19	21
45-55	8	17	25
агент	5	10	19

В результаті аналізу початковий інтервал 20-80 має найменшу середню та мінімальну кількість кроків. На даному рівні довіри найскладніше контролювати переміщення героя. Проте все одно присутній рівень випадковості у роботі даного алгоритму що легко можна побачити зі значень максимальної кількості кроків.

Навчання можна назвати успішним, хоча отриманий результат не набагато кращий за роботу алгоритму. Можливим покращенням

результату може бути подовжене навчання з більшою кількістю супротивників.

Для аналізу зміни довіри з прогресом виконання алгоритму було побудовано графік (рис. 3.16), при створенні графіка було використано середні значення на кожному кроці виконання алгоритму.

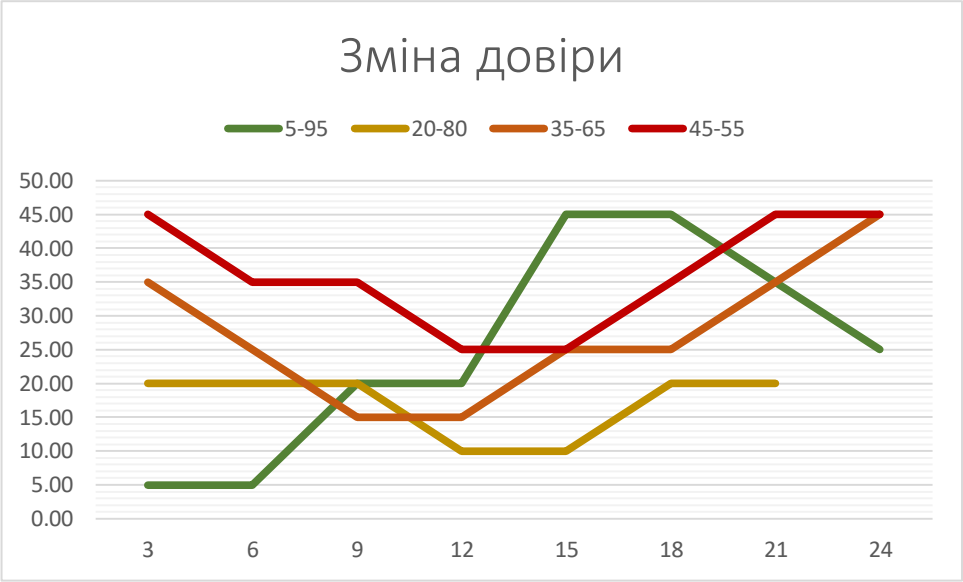

Рисунок 3.16 – Графік зміни довіри з часом

На графіку зображено зміну мінімального значення довіри, чим нижче розташована точка тим більший шанс того, що алгоритм повірить гравцю. З графіка видно, що при низькому початковому рівні довіри (45 та 35) алгоритм не слідкував правильним порадам гравця і постійно потрапляв до глухих кутів (спадаючий тренд), поступово збільшуючи довіру гравцю. Після відновлення довіри гравець мав змогу використати свій шанс для успішного призупинення героя.

Протилежна тенденція виникає при високій початковій довірі (5). І знову інтервал 20-80 є унікальним – тренд майже не змінюється протягом виконання алгоритму, що означає загалом правильне передбачення дій гравця.

4 ВИСНОВКИ

В процесі виконання роботи були проаналізовані найбільш популярні ігрові движки, описані їх сильні і слабкі сторони, було обрано движок Unity 3d як середовище подальшої розробки. Особлива увага була приділена алгоритмам проходження лабіринту та вибору оптимального — алгоритм має ефективно проходити лабіринт і у нього можна додати концепт довіри.

Було виконано аналіз алгоритмів створення лабіринту та обрано алгоритм для подальшого використання у роботі. Також був виконаний огляд машинного навчання та проаналізований набір інструментів для нього у Unity .

Після проведення аналізу існуючих технологій було реалізовано обрані алгоритми у середовищі Unity. Розробка алгоритму самостійної поведінки базувалася на алгоритмі Трема, що використовується для проходження двомірних лабіринтів. Алгоритм поведінки за умов взаємодії з іншим гравцем використовував концепцію довіри в рефлексивних системах та конкуруючих структурах.

Останнім кроком розробки було підключено ML-agents до проекту, за допомогою яких було успішно виконано навчання системи. Працездатність отриманої системи була перевірена та порівняна з результатами роботи створеного раніше алгоритму.

Хоча навчання і було успішним, система ефективніше передбачує поведінку гравця ніж алгоритм, результати можуть бути кращими при більшій кількості взаємодій з людиною в процесі навчання. Для цього можна зробити гру більш красивою та випустити на ринок, де набагато більше людей матимуть доступ до неї.

5 СПИСОК ЛІТЕРАТУРИ

1. Что такое игровой движок - Режим доступу: https://unotices.com/page-answer.php?id=5666
2. Сравнение Unity и Unreal Engine - Режим доступу: https://dtf.ru/gamedev/7227-orel-ili-reshka-sravnenie-unity-i-unreal-engine
3. Unity3D или Unreal Engine 4 - Режим доступу: https://stfalcon.com/ru/blog/post/unity3d-vs-unreal-engine-4
4. Unity Store - Режим доступа: https://store.unity.com/
5. UnrealEngine FAQ - Режим доступу: https://www.unrealengine.com/en-US/faq
6. Maze solving algorithms - Режим доступу: http://www.astrolog.org/labyrnth/algrithm.htm#solve
7. Tjiharjadi S. Optimization Maze Robot Using A* and Flood Fill Algorithm /S. Tjiharjadi, M. Chandra Wijaya - International Journal of Mechanical Engineering and Robotics Research, 2017 - Vol.6, No.5
8. Buck J. Basil and Fabian, a wizard and his man/ J. Basil - Algorithms blog, 2014 - interlude - [Режим доступу] - http://blog.jamisbuck.org/
9. Лефевр В.А. Конфликтующие структуры. Издание второе, переработанное и дополненное. — «Советское радио», 1973. // Электронная публикация: Центр гуманитарных технологий. — 17.10.2016.
10. Филимонов А. Б. Задача прохождения лабиринта интеллектуальными агентами. / А. Б. Филимонов, Н. Б. Филимонов, В. Ю. Тихонов - DOI: 10.17587/mau.17.750-761, 2016
11. Maze Algorithms - Режим доступу https://www.jamisbuck.org/mazes/
12. Katherine Isbister, Games Move Us: Emotion by Design - MIT Press, 2016 р. – 326 ст.

13. Karen Pryor, Don't Shoot the Dog: The New Art of Teaching and Training – Bantam, 1999 – 224 p.
14. Robert Axelrod, The Evolution of Cooperation - Basic Books, 2006 – 164 p.
15. Unreal Engine documentation - Режим доступу: https://docs.unrealengine.com/en-us/
16. Unreal Engine 4 vs. Unity: Which Game Engine Is Best for You? - Режим доступу: https://www.pluralsight.com/blog/film-games/unreal-engine-4-vs-unity-game-engine-best
17. The Top 10 Video Game Engines - Режим доступу: https://www.gamedesigning.org/career/video-game-engines/
18. Machine Learning - Машинне навчання - Режим доступу: https://www.it.ua/knowledge-base/technology-innovation/machine-learning
19. Machine Learning. What it is and why it matters - Режим доступу: https://www.sas.com/en_us/insights/analytics/machine-learning.html
20. Machine Learning and Pattern Recognition - Режим доступу: https://dzone.com/articles/machine-learning-and-pattern-recognition#:~:text=Pattern%20Recognition%20is%20an%20engineering,patterns%20and%20regularities%20in%20data
21. Розпізнавання образів - Режим доступу: https://wiki.tntu.edu.ua/%D0%A0%D0%BE%D0%B7%D0%BF%D1%96%D0%B7%D0%BD%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D1%96%D0%B2
22. What is pattern recognition and why it matters? - Режим доступу: https://theappsolutions.com/blog/development/pattern-recognition-guide/
23. What Is Pattern Recognition in Machine Learning - Режим доступу: https://huspi.com/blog-open/pattern-recognition-in-machine-learning

24. Bishop Christopher Pattern Recognition and Machine Learning/ C. Bishop; Springer-Verlag New York – 2006, p. 738

25. Everything You Need To Know About Machine Learning In Unity 3D - Режим доступу: https://analyticsindiamag.com/everything-you-need-to-know-about-machine-learning-in-unity-3d/

26. Unity ML-Agents Toolkit - Режим доступу: https://github.com/Unity-Technologies/ml-agents

27. Maze generations: Algorithms and Visualizations - Режим доступу: https://medium.com/analytics-vidhya/maze-generations-algorithms-and-visualizations-9f5e88a3ae37

28. Maze Generation: Algorithm Recap - Режим доступу: https://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap

29. Бацман, А. Р. : Інформаційне та програмне забезпечення ігрової системи. Модуль штучного інтелекту [Текст]: робота на здобуття кваліфікаційного ступеня бакалавра; спец.: 122 - комп`ютерні науки (інформатика) / А. Р. Бацман; наук. керівник І.В. Шелехов. - Суми: СумДУ, 2019. - 39 с.

5 ДОДАТОК

1. Класс Maze.cs основні функції:

```
public Maze(int width, int height, System.Random rg)
    {
        this.width = width;
        this.height = height;

        this.startX = 1;
        this.startY = 1;

        this.rg = rg;
    }

    public bool GetCell(int x, int y)
    {
        if (x >= width || x < 0 || y >= height || y <= 0)
        {
            return false;
        }

        return grid[x, y];
    }

public void Generate()
    {
        grid = new bool[width, height];

        startX = 1;
        startY = 1;

        grid[startX, startY] = true;

        MazeDigger(startX, startY);
    }

void MazeDigger(int x, int y)
    {
        int[] directions = new int[] { 1, 2, 3, 4 };

        Tools.Shuffle(directions, rg);

        for (int i = 0; i < directions.Length; i++)
        {
            if (directions[i] == 1)
            {
                if (y - 2 <= 0)
                    continue;

                if (grid[x, y - 2] == false)
                {
                    grid[x, y - 2] = true;
                    grid[x, y - 1] = true;

                    MazeDigger(x, y - 2);
                }
            }
```

```
            if (directions[i] == 2)
            {
                if (x - 2 <= 0)
                    continue;

                if (grid[x - 2, y] == false)
                {
                    grid[x - 2, y] = true;
                    grid[x - 1, y] = true;

                    MazeDigger(x - 2, y);
                }
            }

            if (directions[i] == 3)
            {
                if (x + 2 >= width - 1)
                    continue;

                if (grid[x + 2, y] == false)
                {
                    grid[x + 2, y] = true;
                    grid[x + 1, y] = true;

                    MazeDigger(x + 2, y);
                }
            }

            if (directions[i] == 4)
            {
                if (y + 2 >= height - 1)
                    continue;

                if (grid[x, y + 2] == false)
                {
                    grid[x, y + 2] = true;
                    grid[x, y + 1] = true;

                    MazeDigger(x, y + 2);
                }
            }
           }
```

2. Класс MazeGenerator.cs основні функції:

```
void Start()
    {
        mazeRG = new System.Random(mazeSeed.GetHashCode());

        if (mazeWidth % 2 == 0)
            mazeWidth++;

        if (mazeHeight % 2 == 0)
        {
            mazeHeight++;
        }

        maze = new Maze(mazeWidth, mazeHeight, mazeRG);
        maze.Generate();

        DrawMaze();
```

```
    }

    void DrawMaze()
    {
        for (int x = 0; x < mazeWidth; x++)
        {
            for (int y = 0; y < mazeHeight; y++)
            {
                Vector3 position = new Vector3(x, y);

                if (maze.Grid[x, y] == true)
                {
                    CreateMazeSprite(position, floorSprite, transform, 0, ma-
zeRG.Next(0, 3) * 90);
                }
                else
                {
                    CreateMazeSprite(position, roofSprite, transform, 0, 0);
                }
            }
        }
    }

    public bool GetMazeGridCell(int x, int y)
    {
        return maze.GetCell(x, y);
    }

    void CreateMazeSprite(Vector3 position, Sprite sprite, Transform parent, int sort-
ingOrder, float rotation)
    {
        MazeSprite mazeSprite = Instantiate(mazeSpritePrefab, position, Quaterni-
on.identity) as MazeSprite;
        mazeSprite.SetSprite(sprite, sortingOrder);
        mazeSprite.transform.SetParent(parent);
        mazeSprite.transform.Rotate(0, 0, rotation); }
```

3. Класс SpriteRenderer.cs основні функції:

```
void Awake()
    {
        sRenderer = GetComponent<SpriteRenderer>();
    }

public void SetSprite(Sprite sprite, int sortingOrder)
    {
        sRenderer.sprite = sprite;
        sRenderer.sortingOrder = sortingOrder;
    }

public void SetSprite(Sprite sprite)
    {
        SetSprite(sprite, 0);
           }
```

4. Класс Map.cs основні функції:

```
void SetValues()
    {
        string text = "";
        string dat = "";

        for (int x = MazeGenerator.instance.mazeWidth - 1; x >= 0; x--)
```

```
    {
        for (int y = 0; y < MazeGenerator.instance.mazeHeight; y++)
        {
            if(MazeRunner.instance.goalX == y && MazeRunner.instance.goalY == x)
            {
                text += "5 ";
            }
            else if (MazeRunner.instance.field[y, x] == 0)
            {
                text += "■ ";
            }
            else if (MazeRunner.instance.field[y, x] == 3)
            {
                text += "1 ";
            }
            else if (MazeRunner.instance.field[y, x] == 2)
            {
                text += "2 ";
            }
            else if(MazeRunner.instance.field[y, x] == 5)
            {
                text += "5 ";
            }
            else
            {
                text += "   ";
            }
        }
        text += "\n";
    }
    map.text = text;

    dat += "Total junctions: " + MazeRunner.instance.junctions;
    dat += "\nTimes trusted: " + MazeRunner.instance.trustedTimes;
    dat += "\nDeadends hit: " + MazeRunner.instance.deadends;

    data.text = dat;
}
```

5. Класс MazeRunner.cs змінні:

```
public static MazeRunner instance;
public float walkSpeed;
Vector2 direction = Vector2.zero;

static int targetX = 1;
static int targetY = 1;

int currentX = 1;
int currentY = 1;

public int goalX;
public int goalY;

public int[,] field;
static int hitEnd = 0;

bool trust = true;
bool trusted = false;
static int trustMin = 20;
static int trustMax = 80;
```

```
    bool wait = false;

    public int deadends = 0;
    public int trustedTimes = 0;
    public int junctions = 0;
```

6. Класс MazeRunner.cs основні функції:

```
void Start()
    {
        field = new int[MazeGenerator.instance.mazeWidth, MazeGenera-
tor.instance.mazeHeight];
        setField();
        //setGoal();
    }

    void Update()
    {
        bool targetReached = transform.position.x == targetX && transform.position.y
== targetY;

        currentX = Mathf.FloorToInt(transform.position.x);
        currentY = Mathf.FloorToInt(transform.position.y);

        if (targetReached)
        {
            pickNextCell();
            Map.instance.OnMove();
        }

        transform.position = Vector3.MoveTowards(transform.position, new Vec-
tor3(targetX, targetY), walkSpeed * Time.deltaTime);
    }

    void setGoal()
    {
        int radius = 2;

        int endX = MazeGenerator.instance.mazeWidth - 1;
        int endY = MazeGenerator.instance.mazeHeight - 1;

        for (int x = endX - radius; x <= endX; x++)
        {
            for (int y = endY - radius; y <= endY; y++)
            {
                if (field[x, y] == 1)
                {
                    goalX = x;
                    goalY = y;
                }
            }
        }

        Debug.Log(goalX + " " + goalY);
    }

    void setField()
    {
        for (int i = 0; i < MazeGenerator.instance.mazeWidth; i++)
            for (int j = 0; j < MazeGenerator.instance.mazeHeight; j++)
                field[i, j] = Convert.ToByte(MazeGenerator.instance.GetMazeGridCell(i,
j));
```

```
        }
void backtrack()
    {

        field[currentX, currentY] = 2;

        if (field[currentX + 1, currentY] == 3)
        {
            targetX = currentX + 1;
            targetY = currentY;
            return;
        }
        if (field[currentX - 1, currentY] == 3)
        {
            targetX = currentX - 1;
            targetY = currentY;
            return;
        }
        if (field[currentX, currentY + 1] == 3)
        {
            targetX = currentX;
            targetY = currentY + 1;
            return;
        }
        if (field[currentX, currentY - 1] == 3)
        {
            targetX = currentX;
            targetY = currentY - 1;
            return;
        }

        Debug.Log("No exit in maze");
        wait = true;
    }

    void pickNextCell()
    {
        if (field[currentX, currentY] == 1) field[currentX, currentY] = 3;
        if (currentX == goalX && currentY == goalY)
        {
            Debug.Log("won");
            return;
        }

        List<int> arrTargetX = new List<int>(), arrTargetY = new List<int>();

        if (MazeGenerator.instance.GetMazeGridCell(currentX + 1, currentY) &&
field[currentX + 1, currentY] == 1)
        {
            arrTargetX.Add(currentX + 1);
            arrTargetY.Add(currentY);
        }
        if (MazeGenerator.instance.GetMazeGridCell(currentX - 1, currentY) &&
field[currentX - 1, currentY] == 1)
        {
            arrTargetX.Add(currentX - 1);
            arrTargetY.Add(currentY);
        }
        if (MazeGenerator.instance.GetMazeGridCell(currentX, currentY + 1) &&
field[currentX, currentY + 1] == 1)
        {
```

```
            arrTargetX.Add(currentX);
            arrTargetY.Add(currentY + 1);
        }
        if (MazeGenerator.instance.GetMazeGridCell(currentX, currentY - 1) &&
field[currentX, currentY - 1] == 1)
        {
            arrTargetX.Add(currentX);
            arrTargetY.Add(currentY - 1);
        }

        if (wait)
        {
            trustUser();
        }

        if (!trust)
        {
            junctions++;

            if (trustMin > 40)
            {
                Debug.Log("Random");
                int pos = UnityEngine.Random.Range(0, arrTargetX.Count);
                targetX = arrTargetX[pos];
                targetY = arrTargetY[pos];

                trust = true;
                return;
            }

            Debug.Log("Not your way");
            if (targetX == arrTargetX[0] && targetY == arrTargetY[0])
            {
                targetX = arrTargetX[1];
                targetY = arrTargetY[1];

                trust = true;
                return;
            }

            targetX = arrTargetX[0];
            targetY = arrTargetY[0];

            trust = true;
            return;
        }
        if (arrTargetX.Count == 1)
        {
            hitEnd = 0;
            targetX = arrTargetX[0];
            targetY = arrTargetY[0];
            return;
        }

        if (arrTargetX.Count == 0)
        {
            if (hitEnd == 0)
            {
                if (trusted)
                {
                    trustMin += 20;
                    trustMax -= 20;
```

```
                trusted = false;
            }
            else if (!trusted)
            {
                trustMin -= 10;
                trustMax += 10;
            }
            hitEnd = 1;
            deadends++;
        }

        backtrack();
        return;
    }

    wait = true;
}

void trustUser()
{
    direction.x = Input.GetAxisRaw("Horizontal");
    direction.y = Input.GetAxisRaw("Vertical");

    //check if user point to already walked path
    if (direction.x > 0)
    {
        if (field[currentX + 1, currentY] == 2)
        {
            Debug.Log("Been there");

            trustMin += 10;
            trustMax -= 10;
            trusted = false;
            trust = false;
            wait = false;

            return;
        }
    }
    else if (direction.x < 0)
    {
        if (field[currentX - 1, currentY] == 2)
        {
            Debug.Log("Been there");

            trustMin += 10;
            trustMax -= 10;
            trusted = false;
            trust = false;
            wait = false;

            return;
        }
    }
    else if (direction.y > 0)
    {
        if (field[currentX, currentY + 1] == 2)
        {
            Debug.Log("Been there");

            trustMin += 10;
            trustMax -= 10;
```

```
            trusted = false;
            trust = false;
            wait = false;

            return;
        }
    }
    else if (direction.y < 0)
    {
        if (field[currentX, currentY - 1] == 2)
        {
            Debug.Log("Been there");

            trustMin += 10;
            trustMax -= 10;
            trusted = false;
            trust = false;
            wait = false;

            return;
        }
    }

    if(direction.x != 0 || direction.y != 0)
    {
        if (direction.x > 0)
        {

            if (MazeGenerator.instance.GetMazeGridCell(currentX + 1, currentY))
            {
                targetX = currentX + 1;
                targetY = currentY;
            }
        }
        else if (direction.x < 0)
        {

            if (MazeGenerator.instance.GetMazeGridCell(currentX - 1, currentY))
            {
                targetX = currentX - 1;
                targetY = currentY;
            }
        }
        else if (direction.y > 0)
        {

            if (MazeGenerator.instance.GetMazeGridCell(currentX, currentY + 1))
            {
                targetX = currentX;
                targetY = currentY + 1;
            }
        }
        else if (direction.y < 0)
        {

            if (MazeGenerator.instance.GetMazeGridCell(currentX, currentY - 1))
            {
                targetX = currentX;
                targetY = currentY - 1;
            }
        }
```

```
            float trustLvl = UnityEngine.Random.Range(0, 100);
            if (trustLvl < trustMin || trustLvl > trustMax)
            {
                Debug.Log("U lie");
                trusted = false;
                trust = false;
                wait = false;

                return;
            }

            Debug.Log("Trusting");
            transform.position = Vector3.MoveTowards(transform.position, new Vec-
tor3(targetX, targetY), walkSpeed * Time.deltaTime);

            junctions++;
            trust = true;
            trusted = true;
            trustedTimes++;
            wait = false;
        }

       }
```