Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота магістра

на тему: «Комп'ютерна реалізація аналізу пошуку шляху в програмованих об'єктах ігри»

Виконав: студента групи К22-1М
Спеціальність 122 «Комп'ютерні науки»
Фесенко А.Р.
(прізвище та ініціали)

Керівник к.т.н., доц. Чупілко Т.А.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів
(місто роботи)
Доцент кафедри кібербезпеки та
(посада)
інформаційних технологій
(посада)
к.т.н., доц. Савченко Ю.В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2024

АНОТАЦІЯ

Фесенко А.Р. Комп’ютерна реалізація аналізу пошуку шляху в програмованих об'єктах гри.

Дипломна робота (проект) на здобуття освітнього ступеня магістр за спеціальністю 122 «Комп’ютерні наук». – Університет митної справи та фінансів, Дніпро, 2024.

Об’єктом дослідження є процес пошуку найкоротшого шляху в комп’ютерних іграх.

Предмет дослідження це методи та алгоритми пошуку найкоротшого шляху в комп’ютерних іграх та їх програмна реалізація.

Метою роботи є оптимізації алгоритму пошуку найкоротшого шляху.

Ця робота присвячена розробці комп’ютерного аналізу пошуку шляху в програмованих об’єктах гри. Дослідження присвячено алгоритму A* в середовищі Unity з урахуванням gCost, hCost і fCost для ефективного визначення найкоротших шляхів. Особлива увага приділена динамічному оновленню сусідів кожного вузла, що забезпечує гнучкість алгоритму у відповіді на зміни у середовищі. Дослідження підтверджує ефективність алгоритму в задачах пошуку шляху, що є критичним у різних областях, таких як ігровий дизайн, симуляції та робототехніка. Інтеграція алгоритму в Unity відкриває нові можливості для розробників та підкреслює важливість дослідження для подальшого розвитку цих галузей.

Ключові слова: A*, Unity, IDA*, SMA*, пошук найкоротшого шляху, графи.

ABSTRACT

Fesenko A. R., Computer Implementation of Pathfinding Analysis in Programmed Game Objects.

Diploma thesis (project) for the degree of Master's Degree in specialty 122 "Computer Science." - University of Customs and Finance, Dnipro, 2024.

The object of the research is the process of finding the shortest path in computer games.

The subject of the research is the methods and algorithms of finding the shortest path in computer games and their software implementation.

The aim of the work is to optimize the shortest pathfinding algorithm.

This work is dedicated to the development of computer analysis of pathfinding in programmed game objects. The research is devoted to the A* algorithm in the Unity environment, considering gCost, hCost, and fCost for efficient determination of shortest paths. Special attention is paid to the dynamic updating of node neighbors, providing flexibility of the algorithm in response to changes in the environment. The research confirms the effectiveness of the algorithm in pathfinding tasks, which is critical in various areas such as game design, simulations, and robotics. Integrating the algorithm into Unity opens up new possibilities for developers and emphasizes the importance of research for the further development of these fields.

Keywords: A*, Unity, IDA*, SMA*, pathfinding, graphs.

ЗМІСТ

ВСТУП

Пошук шляху – це фундаментальна задача в комп'ютерних науках, що знаходить широке застосування в штучному інтелекті, робототехніці, комп'ютерних іграх, картографії та інших сферах. В контексті розробки комп’ютерних ігор задача пошуку шляху полягає у визначенні оптимального маршруту для персонажа з початкової точки до кінцевої, враховуючи особливості віртуального середовища та динаміку ігрового процесу.

Актуальність теми пошуку шляху в програмованих об'єктах ігри обумовлена зростанням складності та динамічності ігрових середовищ, підвищенням вимог до реалістичності поведінки агентів, а також розширенням сфери застосування алгоритмів пошуку шляху в інших сферах.

Сучасні ігри ставлять перед персонажами складні задачі з навігації в віртуальних світах, де їм необхідно швидко та ефективно знаходити шлях до цілей, обминаючи перешкоди та реагуючи на зміни в середовищі. Гравці очікують, що персонажі в іграх будуть вести себе максимально реалістично, що робить алгоритми пошуку шляху незамінними інструментами для розробників.

В умовах обмежених ресурсів мобільних пристроїв та інших платформ виникає потреба в оптимізації алгоритмів пошуку шляху для забезпечення плавної роботи ігрового процесу.

Пошук шляху використовується не лише в іграх, але й в робототехніці, навігації, логістиці та інших сферах. Розвиток штучного інтелекту, машинного навчання та інших передових технологій відкриває нові можливості для покращення алгоритмів пошуку шляху.

Новизна дослідження полягає в імплементації алгоритму A* в Unity, де було враховано gCost, hCost і fCost для ефективної роботи алгоритму. Особлива увага приділена динамічному оновленню сусідів для кожного вузла, що дозволяє гнучке налаштовування відповідно до змін у середовищі.

Алгоритм демонструє високу ефективність у визначенні найкоротших шляхів, що має значення для задач, де критично важлива мінімізація довжини маршруту. Така гнучкість та налаштовуваність робить алгоритм придатним для широкого спектру застосувань, включаючи ігрові проекти, симуляції та робототехніку. Легкість інтеграції в існуючі проекти Unity є ще одним значущим внеском дослідження, що спрощує використання та модифікацію алгоритму для різних потреб.

Загалом, дослідження підтверджує значущість алгоритму A* в комп'ютерних науках, демонструючи його практичну цінність в задачах шляхоутворення. Реалізація алгоритму в Unity відкриває нові можливості для розробників у сферах ігрового дизайну, симуляцій та робототехніки, підкреслюючи важливість дослідження для подальшого розвитку цих напрямків.

Метою дослідження є оптимізації алгоритму пошуку найкоротшого шляху.

Методи дослідження в роботі використовувалися методи та технології розробки ігрових додатків, ігровий рушій Unity, мова програмування C#.

Завдання дослідження:

1. Провести аналіз методів пошуку найкоротшого шляху для визначення їх переваг і недоліків.
2. Визначити, який метод є найбільш ефективним для різних типів рішень.
3. Розглянути можливі модифікації методу та реалізувати на практиці та визначити, який метод є найбільш ефективним для різних типів рішень.
4. Розробити програмне забезпечення, засноване на модифікованому методі пошуку найкоротшого шляху.

Об'єкт дослідження: Процес пошуку найкоротшого шляху.

Предмет дослідження: Методи та алгоритми пошуку найкоротшого шляху в комп’ютерних іграх та їх програмна реалізація.

Практичне значення отриманих результатів: Оптимізація процесу пошуку шляху в комп’ютерних іграх, зменшення часу на прийняття рішень.

Структура роботи:

Розділ 1 Дослідження та оцінка методів прийняття рішень. У даному розділі буде виконано пошук та аналіз методів пошуку найкоротшого шляху, їх опис та виявлення недоліків.

Розділ 2 Алгоритм пошуку найкоротшого шляху в іграх на основі методу A*. В даному розділі буде проаналізовано та обрано кращій метод та запропонована його оптимізація.

Розділ 3 Проектування та розробка алгоритму пошуку шляху в програмованих об'єктах гри. В даному розділі буде спроектовано та розроблено програму для вирішення важливих задачі пошуку найкортшого шляху в ігровому світі.

Робота складається зі вступу, 3-х розділів, висновків, списку використаних джерел, 1 додатку. Обсяг роботи 85 сторінок, 12 рисунків та 1 формули.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналіз публікацій щодо алгоритмів пошуку найкоротшого шляху в іграх

Пошук оптимального маршруту (англ. Pathfinding) – це складний процес визначення найбільш вигідного шляху між двома місцями. На перший погляд може здаватися, що це доволі проста задача, але вона ускладнюється, коли врахувати такі фактори, як наявність перешкод, складність території або час необхідний для пересування. Відтак, з'ясовується, що визначення найбільш вигідного шляху є більш складним завданням, ніж просто прокладення лінії між двома точками [1].

Багато алгоритмів пошуку шляху існують для вирішення одного й того самого завдання: визначення найкоротшого маршруту між двома точками на карті. Проте, методи, якими вони досягають цієї мети, можуть значно відрізнятися. З плином часу деякі алгоритми оптимізувались та стали кращими за своїх попередників з точки зору швидкості та ефективності.

Однак, не завжди найвища ефективність є пріоритетом. В контексті відеоігор, особливо важливим стає реалістичне моделювання поведінки персонажів, керованих комп'ютером (NPC). Розробники іноді віддають перевагу поведінці NPC, яка наближена до реалістичної замість того, щоб просто дотримуватись найкоротшого маршруту. Це означає, що вони можуть обирати алгоритми пошуку шляху, які не обов'язково є найефективнішими в плані використання ресурсів, але забезпечують більш природну поведінку персонажів [8].

Таким чином, у сфері розробки ігор, підбір алгоритму пошуку шляху може базуватися не лише на технічних характеристиках, а й на прагненні до створення більш занурюваного та реалістичного ігрового досвіду. Розглянемо

декілька найпопулярніших алгоритмів та особливості їхнього застосування в контексті відеоігор.

На ринку присутня велика кількість алгоритмів для визначення шляху. Багато відомих ігор розробляють унікальні версії цих алгоритмів, модифікуючи та адаптуючи існуючі рішення під свої специфічні потреби. Ось перелік декількох широко використовуваних алгоритмів пошуку шляху:

1) Пошук у ширину (BFS) рівномірно досліджує всі напрямки. Це надзвичайно корисний алгоритм, який використовується не тільки для знаходження шляхів, а й для генерації процедурних карт, пошуку шляхів за допомогою поля потоків, карт відстаней та іншого аналізу карт [2].

2) Алгоритм Дейкстри, також відомий як пошук із єдиною вартістю, дозволяє нам визначати пріоритети шляхів для дослідження. Замість дослідження всіх можливих шляхів однаково, він віддає перевагу шляхам з меншою вартістю. Ми можемо призначити нижчу вартість, щоб сприяти руху по дорогах, вищу вартість, щоб уникати ворогів, і багато іншого. Коли вартість руху змінюється, ми використовуємо цей алгоритм замість пошуку у ширину [5].

3) A* – це модифікація алгоритму Дейкстри, яка оптимізована для одного пункту призначення. Алгоритм Дейкстри може знаходити шляхи до всіх місць; A* знаходить шляхи до одного місця або найближчого з кількох місць. Він надає пріоритет шляхам, які, здається, ведуть ближче до цілі [3].

Хоча основна ціль цих алгоритмів однакова – знайти найкоротший шлях між двома точками на карті, методи, за допомогою яких кожен з них досягає цієї мети, різняться. Це дає розробникам можливість вибрати алгоритм, найбільш адаптований під конкретні потреби та особливості їхніх ігор.

1.2 Аналіз алгоритмів пошуку найкоротшого шляху

Пошук у ширину (BFS)

Алгоритм пошуку у ширину (BFS) є одним з основних інструментів для аналізу та навігації по графам. Він починає свою роботу з визначеного вузла, який називається кореневим, і поступово просувається через граф, обробляючи кожен вузол на однаковій відстані від кореня перед переходом до наступного рівня. Цей методичний підхід дозволяє BFS досліджувати граф шар за шаром, що робить його ідеальним для задач, де потрібно забезпечити рівномірне та повне охоплення простору графа.

BFS використовує структуру даних у вигляді черги для зберігання вузлів, які потрібно обробити. Коли вузол обробляється, всі його сусідні вузли, які ще не були відвідані, додаються до черги. Це забезпечує, що алгоритм спочатку повністю обробляє всі вузли на поточному рівні, перш ніж перейти до наступного. Ця особливість робить BFS особливо корисним для знаходження найкоротших шляхів у графах, де всі ребра мають однакову вагу, оскільки він гарантовано знайде найкоротший шлях від кореневого вузла до будь-якого іншого вузла.

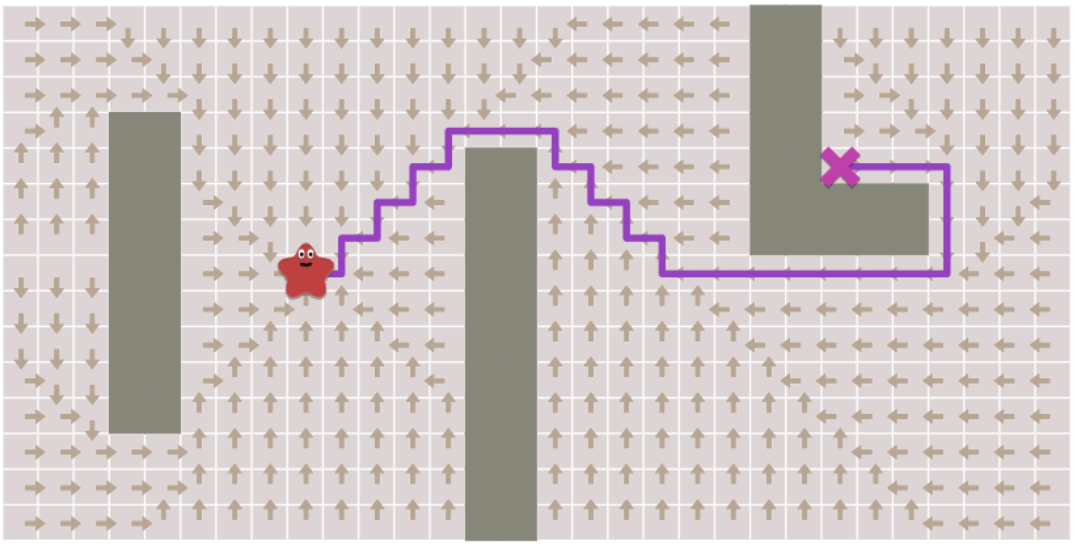


Рисунок 1.1 – Пошук шляху алгоритмом BFS

Це найпростіший алгоритм пошуку шляху. Він працює не лише на сітках, як показано тут, а й на будь-якій структурі графу. У підземеллі локації графу можуть бути кімнатами, а ребра графу - дверима між ними. У платформері локації графу можуть бути певними місцями, а ребра графу - можливими діями, такими як рух вліво, рух вправо, стрибок вгору, стрибок вниз. Загалом, можна уявляти граф як стани та дії, що змінюють стан.

Однак, BFS має свої обмеження, зокрема він може бути ресурсоємним у великих графах, оскільки потребує зберігання всіх вузлів на поточному рівні перед переходом до наступного. Незважаючи на це, його повнота та здатність забезпечувати рівномірний обхід роблять його незамінним інструментом у багатьох областях, включаючи комп'ютерні науки, аналіз соціальних мереж та, звісно, розробку відеоігор. У сфері ігрової індустрії BFS може використовуватися не тільки для пошуку шляхів, але й для генерації карт, створення полів потоків, карт відстаней та виконання інших видів аналізу карт, що робить його універсальним та цінним інструментом у руках геймдевелоперів [2].

Приклади застосування алгоритму BFS:

1) Гра The Legend of Zelda: Breath of the Wild – використовує алгоритм BFS для пошуку найкоротшого шляху до пункту призначення.

Алгоритм BFS використовується в The Legend of Zelda: Breath of the Wild для пошуку найбезпечнішого маршруту до пункту призначення:

1. Створення графа:

- Ігровий світ Breath of the Wild представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками – ребрами.
- Кожній вершині присвоюється вага, яка відповідає складності її досягнення: наявність ворожих істот, перешкод, відстань тощо.

2. Пошук маршруту:

- Визначається точка початку – поточне розташування Link та точка кінця – пункт призначення.

- Запускається з вершини точки початку.
- Обчислюється всі можливі шляхи від вершини точки початку до вершини точки кінця з урахуванням ваги ребер.
- Вибирається один з можливих шляхів, який буде найкоротшим або найбезпечнішим маршрутом до пункту призначення.

2) Гра Diablo II – використовує алгоритм BFS для пошуку найкоротшого маршруту до виходу з підземелля.

Алгоритм BFS використовується в Diablo II для пошуку найкоротшого маршруту до виходу з підземелля:

1. Створення графа

- Карта підземелля Diablo представлена у вигляді графа, де кожна кімната є вершиною, а тунелі та проходи між кімнатами – ребрами.
- Кожній вершині присвоюється вага, яка відповідає складності її досягнення (наявність монстрів, перешкод, відстань тощо).

2. Пошук маршруту

- Визначається кімната початку – поточне розташування персонажа та кімната кінця – вихід з підземелля.
- Запускається з вершини кімнати початку.
- Обчислюються всі можливі шляхи від вершини кімнати початку до вершини кімнати кінця з урахуванням ваги ребер.
- Вибирається один найкоротший маршрут до виходу з підземелля.

3) Гра Minecraft – використовує алгоритм BFS для генерації лабіринтів.

Алгоритм BFS використовується в Minecraft для генерації лабіринтів:

1. Створення графа

- Ігровий світ Minecraft представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками – ребрами.

- Кожній вершині присвоюється вага, яка відповідає складності її досягнення (наявність монстрів, перешкод, відстань тощо).

2. Генерація лабіринту

- Визначається точка початку – вхід лабіринту та точка кінця – вихід лабіринту.
- Запускається з вершини точки початку.
- Обчислює всі можливі шляхи від вершини точки початку до вершини точки кінця з урахуванням ваги ребер.
- Згенерований лабіринт ґрунтується на дереві BFS, де кожна вершина дерева відповідає кімнаті лабіринту, а ребра – тунелям та проходам.

Алгоритм Дейкстри

Видатний учений у сфері інформаційних технологій Едсгер Дейкстра у 1959 році представив свій алгоритм, який пізніше був названий на його честь. Цей алгоритм, що відноситься до класу жадібних алгоритмів, здатен знаходити найкоротші шляхи від однієї обраної точки графу до усіх інших точок, а також може бути адаптований для знаходження найкоротшого шляху між будь-якою парою точок [4].

Основна концепція алгоритму полягає в наступному:

- Процес починається з вузла старту, звідки алгоритм починає аналізувати граф для визначення найкоротших можливих шляхів до інших вузлів.
- Протягом роботи алгоритм зберігає інформацію про найкоротші відстані від початкової точки до кожної з інших точок, при цьому дані оновлюються кожного разу, коли знаходиться більш короткий шлях.
- Вузол вважається "відвіданим" та виключається з подальшого розгляду, коли для нього визначено найкоротший шлях.
- Алгоритм завершується, коли всі вузли були "відвідані" та найкоротші шляхи до них визначені, надаючи на виході повний набір найкоротших шляхів від стартового вузла до усіх інших точок графу.

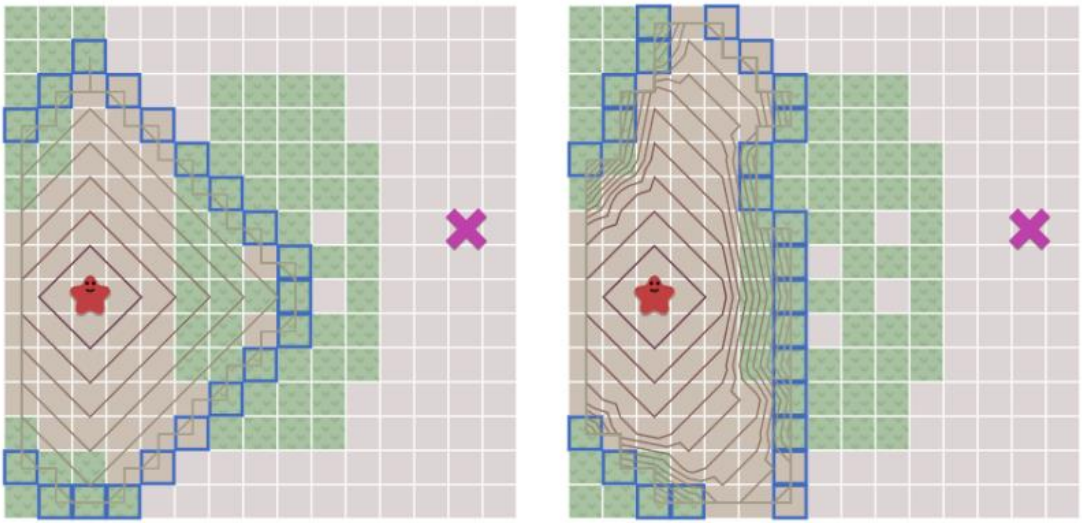


Рисунок 1.2 – Принцип роботи алгоритмів BFS – зліва та Дейкстри – зправа

Базова версія алгоритму Дейкстри вимагає $O(V^2)$ операцій, де V – кількість вершин у графі. У цій реалізації використовуються два масиви: масив дистанцій та масив відміток. На старті алгоритму, дистанції ініціалізуються дуже великим числом, що перевищує будь-яку можливу довжину шляху в графі, а масив відміток заповнюється нулями. Дистанція до початкової вершини встановлюється як нуль, і розпочинається основний цикл алгоритму.

У кожній ітерації циклу алгоритм вибирає вершину з найменшою дистанцією, що ще не була оброблена (прапорець рівний нулю), позначає цю вершину як оброблену (встановлюючи прапорець в одиницю), і переглядає всі її сусідні вершини. Якщо знаходиться більш короткий шлях до сусідньої вершини через поточну вершину, алгоритм оновлює дистанцію до цієї сусідньої вершини. Процес триває до тих пір, поки всі вершини не будуть оброблені – прапорець кожної вершини стане рівним 1.

Приклади застосування:

1) Гра Pac-Man – використовує алгоритм Дейкстри для пошуку найкоротшого шляху до їжі.

У грі Pac-Man алгоритм Дейкстри використовується для пошуку найкоротшого шляху від Pac-Man до їжі. Це робиться для того, щоб Pac-Man міг максимально ефективно збирати їжу і уникати привидів.

Ось як це працює:

1. Створення графа. Ігровий світ Pac-Man представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі між точками - ребрами.

2. Ваги ребрам, кожному ребру в графі присвоюється вага, яка відповідає довжині тунелю.

3. Запуск алгоритму Дейкстри відбувається з вершини, де знаходиться Pac-Man.

4. Пошук найкоротшого шляху – обчислює найкоротший шлях від Pac-Man до кожної їжі на карті.

5. Переміщення. Pac-Man переміщується по карті, слідуючи найкоротшому шляху до їжі.

2) Гра Diablo III – використовує алгоритм Дейкстри для пошуку найкоротшого шляху до монстрів.

В Diablo III алгоритм Дейкстри використовується для

- Пошуку найкоротшого шляху до монстрів – дозволяє персонажу гравця (PC) максимально ефективно атакувати монстрів.
- Пошуку найкоротшого шляху до виходів з рівня – допомагає PC швидко проходити рівні.
- Пошуку найкоротшого шляху до NPC – дозволяє PC ефективно взаємодіяти з NPC.

Ось як це працює:

1. Створення графа.

2. Ігровий світ Diablo III представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками – ребрами.

3. Ваги ребер

4. Кожному ребру в графі присвоюється вага, яка відповідає довжині шляху.

5. Запуск алгоритму Дейкстри

6. Алгоритм Дейкстри запускається з вершини, де знаходиться PC.

7. Пошук найкоротшого шляху

8. Алгоритм Дейкстри обчислює найкоротший шлях від PC до монстрів, виходів або NPC на карті.

9. Переміщення PC

PC переміщується по карті, слідуючи найкоротшому шляху до монстрів, виходів або NPC.

3) Гра Civilization – використовує алгоритм Дейкстри для пошуку найкоротшого шляху між містами.

В Civilization алгоритм Дейкстри використовується для:

- Пошуку найкоротшого шляху між містами, що дозволяє юнітам гравця максимально ефективно переміщуватися по карті.
- Планування маршрутів для торгових караванів

Алгоритм Дейкстри може використовується для планування маршрутів для торгових караванів, щоб вони могли максимально ефективно доставляти товари між містами.

- Розрахунок часу переміщення юнітів. Алгоритм Дейкстри використовується для обрахунку кількості ходів який потрібен юнітам для переміщення між містами.

Ось як це працює:

1. Створення графа: Ігровий світ Civilization представлений у вигляді графа, де кожне місто є вершиною, а дороги, що з'єднують міста - ребрами.

2. Ваги ребер. Кожному ребру в графі присвоюється вага, яка відповідає довжині шляху.

3. Алгоритм Дейкстри запускається з вершини, де знаходиться юніта гравця.

4. Пошук найкоротшого шляху. Алгоритм Дейкстри обчислює найкоротший шлях від юніта гравця до будь-якого іншого міста на карті.

5. Переміщення юніта гравця по карті, слідуючи найкоротшому шляху до пункту призначення.

Алгоритм Дейкстри широко використовується не тільки в області комп'ютерних наук, а й у сферах логістики, маршрутизації в мережах, геоінформаційних системах та багатьох інших, де потрібно знайти оптимальні шляхи між точками на мапі.

Цей алгоритм є основою для багатьох інших алгоритмів пошуку шляху і має важливе значення для розвитку сфери алгоритмічних досліджень.

Алгоритм пошуку A*

Алгоритм A* (відомий також як "A star") відноситься до категорії евристичних алгоритмів пошуку та широко застосовується для визначення найефективнішого маршруту між двома точками у графі з позитивною вагою кожного ребра. Цей алгоритм був розроблений Пітером Хартом, Нільсом Нільсоном та Бертрамом Рафаелем у 1968 році. Особливість A* полягає у використанні спеціальної допоміжної функції, званої евристикою, яка спрямовує пошук у більш перспективні напрямки та допомагає зменшити час, необхідний для знаходження рішення. Алгоритм гарантовано знаходить оптимальний шлях, якщо такий існує, завдяки своїй повноті.

Цей алгоритм класифікує вузли графу у три категорії: невідомі вузли, відомі вузли (OpenList) та повністю досліджені вузли (ClosedList). Спочатку, усі вузли, за винятком стартового, є невідомими. Відомі вузли, зі своїм потенційним шляхом, зберігаються у списку з пріоритетом, де з кожним кроком вибираються найбільш обнадійливі вузли для подальшого дослідження. Такий підхід дозволяє оптимізувати швидкість виконання алгоритму.

Коли вузол повністю досліджений, його сусідні вузли додаються до списку відомих вузлів, а сам вузол переміщається до списку повністю досліджених. Якщо сусідні вузли вже досліджені або є у списку відомих, алгоритм оновлює їхній статус, якщо через поточний вузол знайдений коротший шлях.

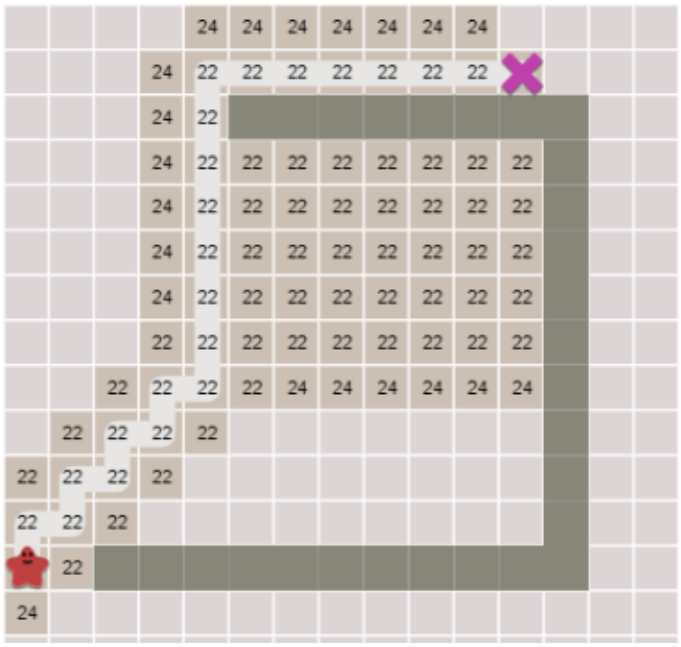


Рисунок 1.3 – Приклад робо алгоритму A*

Процес пошуку продовжується, поки кінцева вершина не опиниться у списку повністю досліджених вузлів, або ж поки список відомих вузлів не спорожніє, що свідчить про відсутність розв'язку. Шлях відновлюється у зворотньому порядку від кінцевої до стартової вершини, але може бути легко перевернутий для отримання шляху у правильному напрямку.

Припустима евристика оцінює вартість досягнення цільового вузла таким чином, що фактична вартість не перевищує оцінену. Це означає, що вона завжди дає нижню межу вартості шляху. Якщо евристика є припустимою, але не монотонною, то можуть виникати ситуації, коли доведеться повторно досліджувати деякі вузли.

Монотонна евристика, крім умови непереоцінення вартості, також повинна задовольняти нерівність трикутника. Це означає, що оцінка вартості від поточного вузла до цілі завжди менша або рівна сумі вартості переходу до сусіднього вузла та оцінки вартості від цього сусіднього вузла до цілі.

Алгоритм A* має кілька ключових властивостей:

1) завжди знаходить розв'язок, якщо такий існує;

2) розв'язок буде оптимальним, і це один із найбільш ефективних алгоритмів пошуку з точки зору кількості досліджених вузлів.

Швидкість виконання алгоритму значно залежить від точності евристичної функції та ефективності реалізації структур даних для зберігання відомих та повністю досліджених вузлів.

Найбільшим недоліком алгоритму A* є його вимога до пам'яті, оскільки потрібно зберігати велику кількість вузлів, що може бути непридатним для задач з великою кількістю можливих станів, таких як гра «П'ятнашки», де кількість унікальних станів досягає 16 факторіал [6].

Приклади використання алгоритму А в іграх:

1) Гра StarCraft – використовує алгоритм A* для пошуку найкоротшого шляху до юнітів противника.

В StarCraft алгоритм A* використовується для:

- Пошуку найкоротшого шляху до юнітів противника, що дозволяє юнітам гравця максимально ефективно атакувати противника.
- Пошуку найкоротшого шляху до ресурсів. Дозволяє юнітам гравця максимально ефективно збирати ресурси.

Алгоритм роботи:

1. Ігровий світ StarCraft представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками - ребрами.
2. Кожному ребру в графі присвоюється вага, яка відповідає довжині шляху.
3. Використовується евристична функція, яка оцінює, наскільки близько вершина знаходиться до юніта противника або ресурсу.
4. Алгоритм A* запускається з вершини, де знаходиться юніти гравця.
5. Алгоритм A* обчислює найкоротший шлях від юніта гравця до юніта противника або ресурсу, використовуючи евристичну функцію.

6. Юніт гравця переміщується по карті, слідуючи найкоротшому шляху.

2) Гра Warcraft III – використовує алгоритм A* для пошуку найкоротшого шляху до ресурсів.

В Warcraft III алгоритм A* використовується для:

- Пошуку найкоротшого шляху до ресурсів:, що дозволяє юнітам гравця максимально ефективно збирати ресурси.
- Пошуку найкоротшого шляху до юнітів противника, що дозволяє юнітам гравця ефективно атакувати противника.

Ось як це працює:

1. Створення графа. Ігровий світ Warcraft III представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками - ребрами.

2. Ваги ребер, кожному ребру в графі присвоюється вага, яка відповідає довжині шляху.

3. Використовується евристична функція, яка оцінює, наскільки близько вершина знаходиться до ресурсу або юніта противника.

4. Алгоритм A* запускається з вершини, де знаходиться юніти персонажу.

5. Алгоритм A* обчислює найкоротший шлях від юніта персонажу до ресурсу або юніта противника.

6. Юніт персонажу переміщується по карті, слідуючи найкоротшому шляху.

3) Гра Half-Life – використовує алгоритм A* для навігації NPC.

В Half-Life алгоритм A використовується для:

- Алгоритм A* використовується NPC для пошуку найкоротшого шляху до місця призначення, враховуючи перешкоди та інші NPC.
- Алгоритм A* використовується для пошуку місць на карті, де ворожі юніти мають найкращу видимість та доступ до гравця.

- Алгоритм A* використовується для планування маршрутів патрулів NPC, щоб вони могли максимально ефективно охоплювати ігровий локацію.

Ось як це працює:

1. Створення графа. Ігровий світ Half-Life представлений у вигляді графа, де кожна точка (перетин) на карті є вершиною, а тунелі, коридори та інші проходи між точками - ребрами.

2. Ваги ребер, кожному ребру в графі присвоюється вага, яка відповідає довжині шляху та складності його подолання (наявність ворогів, перешкод тощо).

3. Використовується евристична функція, яка оцінює, наскільки близько вершина знаходиться до місця призначення NPC.

4. Алгоритм A* запускається з вершини, де знаходиться NPC.

5. Алгоритм A* обчислює найкоротший шлях від NPC до місця призначення, використовуючи евристичну функцію.

6. NPC переміщується по карті, слідуючи найкоротшому шляху до місця призначення.

Таким чином, алгоритм A* є високоефективним рішенням для знаходження оптимального шляху у графах, забезпечуючи точність та швидкість у прийнятті рішень.

Алгоритм Беллмана–Форда

Алгоритм Беллмана–Форда — це алгоритм пошуку найкоротшого шляху в зваженому графі, де вага дуг може бути як позитивною, так і негативною.

Він названий на честь своїх розробників, Річарда Беллмана та Лестера Форда, а пізніше Едварда Мура ідеї цих трьох учених були об'єднані та розвинені, що призвело до створення алгоритму, який ми зараз знаємо як алгоритм Беллмана-Форда. Цей алгоритм став фундаментальним у теорії графів та використовується у багатьох областях, від маршрутизації в мережах до розподілу ресурсів у логістичних системах.

Основні етапи роботи алгоритму:

1. Ініціалізація

Вибрати початкову вершину і встановити відстань до неї як 0, а до всіх інших вершин — як нескінченність (або дуже велике число).

2. Релаксація ребер

Перевіряти всі ребра графа і робити релаксацію. Релаксація ребра (u, v) з вагою w полягає в перевірці, чи можна скоротити відстань до вершини v, порівнюючи поточне значення відстані до v з сумою відстані до u та ваги ребра w. Якщо distance[v] > distance[u] + w, тоді оновити distance[v] = distance[u] + w.

3. Повторення релаксації

Крок релаксації повторюється |V| - 1 разів, де |V| - кількість вершин у графі, що забезпечує знаходження найкоротшого шляху, навіть якщо він проходить через всі вершини.

4. Перевірка на цикли з негативною вагою

Після завершення всіх релаксацій, алгоритм перевіряє наявність циклів з негативною вагою. Це робиться шляхом ще одного проходу по всіх ребрах: якщо відстань може бути скорочена після |V| - 1 релаксацій, то в графі є цикл з негативною вагою.

Алгоритм має ряд переваг та недоліків, а саме:

1) Враховує негативні ваги

Алгоритм може обробляти графи з ребрами, що мають негативні ваги, що робить його корисним у ситуаціях, де інші алгоритми, такі як Дейкстри, не можуть бути застосовані.

2) Виявлення циклів з негативною вагою

Алгоритм може виявити наявність циклів з негативною вагою в графі, що є важливою властивістю при аналізі графів.

3) Застосовність до широкого спектру задач

Хоча алгоритм не є найшвидшим для знаходження найкоротших шляхів, він застосовний у широкому спектрі задач, де інші алгоритми можуть не дати

коректних результатів через специфічні обмеження (наприклад, наявність негативних ваг).

До недоліків відносять:

1) Висока часова складність:

Алгоритм має часову складність O(V*E), де V — кількість вершин, а E — кількість ребер у графі. Це робить алгоритм відносно повільним, особливо для графів з великою кількістю вершин і ребер.

2) Неефективний для густих графів:

У густих графах, де кількість ребер наближається до V^2, алгоритм може бути неефективним через високу часову складність.

3) Повільніше за інші алгоритми для специфічних задач:

Для графів без негативних ваг алгоритм Дейкстри та A* можуть забезпечити кращу продуктивність з меншою часовою складністю.

Алгоритм Беллмана-Форда застосовують у комп'ютерних іграх для різних цілей, враховуючи його здатність працювати з графами, які містять ребра з негативною вагою, та виявляти цикли з негативною вагою. Ось деякі приклади:

1) Маршрутизація NPC

В іграх з великими відкритими світами NPC можуть використовувати алгоритм Беллмана-Форда для знаходження найкоротшого шляху до персонажів, подорожей і інші.

2) Оптимізація ресурсів та маршрутів логістики

У стратегічних іграх алгоритм може бути використаний для оптимізації розподілу ресурсів та логістичних маршрутів, особливо коли враховуються витрати та ризики переміщення через різні території.

3) Балансування геймплею

У процесі розробки гри алгоритм може допомогти розробникам в аналізі та балансуванні геймплею, виявляючи потенційно нескінченні цикли або інші аномалії, які можуть вплинути на геймплей.

4) Планування та штучний інтелект

У іграх, де штучний інтелект повинен приймати складні рішення, наприклад, у торгівлі або плануванні стратегії, алгоритм може використовуватися для виявлення оптимальних шляхів і стратегій з урахуванням можливих змін у грі.

5) Аналіз мережевих ігор

В мережевих іграх, де необхідно аналізувати та оптимізувати передачу даних між вузлами, алгоритм може бути корисним для виявлення найефективніших шляхів передачі даних [7].

Загалом, алгоритм Беллмана-Форда є потужним інструментом в арсеналі алгоритмічного проектування, особливо цінним у ситуаціях, де необхідно розглядати складні вагові умови та динамічні зміни у графі.

1.3 Висновок до першого розділу

Алгоритм A* представляє собою евристичний метод пошуку шляху, який відрізняється своєю ефективністю та точністю. Тим не менш, в залежності від специфіки задачі та обмежень системи, можуть бути обрані інші алгоритми пошуку шляху. Наприклад, алгоритм Дейкстри підходить для ситуацій, де не визначена кінцева ціль, і його використання є доречним, коли евристичні оцінки не можливі. З іншого боку, жадібні алгоритми, які фокусуються на мінімізації поточної оцінки шляху, можуть пропонувати швидкість за рахунок точності оцінки.

Для сценаріїв, де обмеження пам'яті є критичним фактором, існують алгоритми як IDA*, RBFS, MA* та SMA*, які дозволяють оптимізувати використання пам'яті, хоча це може йти на шкоду здатності знаходити оптимальний шлях. Ці алгоритми знижують потребу в пам'яті, але за умови достатнього обсягу пам'яті та використання монотонної евристики, вони здатні забезпечити оптимальні результати.

Однак, в окремих випадках, де потрібна адаптація до динамічних змін у графі, алгоритм D* та його варіанти, які є удосконаленням алгоритму A*, можуть стати кращим вибором.

Щодо алгоритмів Дейкстри та BFS, Беллмана-Форда ці методи також надають важливі можливості для пошуку шляху. Алгоритм Дейкстри особливо корисний для пошуку найкоротшого шляху в зважених графах, тоді як BFS ідеально підходить для не зважених графів або графів, де всі ребра мають однакову вагу, пропонуючи гарантоване знаходження найкоротшого шляху в таких умовах. Алгоритм Беллмана-Форда вирізняється своєю здатністю обробляти графи з негативними вагами ребер та виявляти цикли з негативною вагою забезпечує надійність у визначенні найкоротших шляхів у більш складних графах, де алгоритми, засновані на простіших евристиках, можуть зазнавати невдачі.

У кінцевому підсумку, рішення про вибір алгоритму повинно базуватися на глибокому аналізі вимог до задачі, враховуючи такі фактори, як точність, швидкість, використання пам'яті та специфіку середовища.

РОЗДІЛ 2. АЛГОРИТМ ПОШУКУ НАЙКОРОТШОГО ШЛЯХУ В ІГРАХ НА ОСНОВІ МЕТОДУ A*

2.1 Алгоритм пошуку найкоротшого шляху A*

A* - це алгоритм пошуку шляху в зваженому графі, який визначає найкоротший шлях від початкової вершини до кінцевої. Він враховує вартість фактичного шляху (g) і використовує евристичну оцінку (h) для приблизного визначення вартості шляху до цільової вершини. Алгоритм обирає вершину з найменшим значенням комбінованої вартості (f = g + h) для продовження пошуку до моменту досягнення цільової вершини.

Алгоритм A* визначає оптимальний шлях в графі за допомогою трьох параметрів:

- g – вартість переміщення з початкової клітинки до поточної. Це сума вартості усіх клітинок, відвіданих з моменту виходу з початкової клітинки.
- h – також відоме як евристичне значення, це приблизна вартість переміщення з поточної клітинки до кінцевої. Фактична вартість не може бути обчислена, поки не буде досягнута кінцева клітинка. Отже, h – це оціночна вартість.
- f – сума g і h, тобто f = g + h.

Алгоритм використовує це значення для прийняття рішення. Він обирає клітину з найменшим значенням f і переходить у цю клітину. Цей процес триває до досягнення цільової клітини (рис. 2.1).

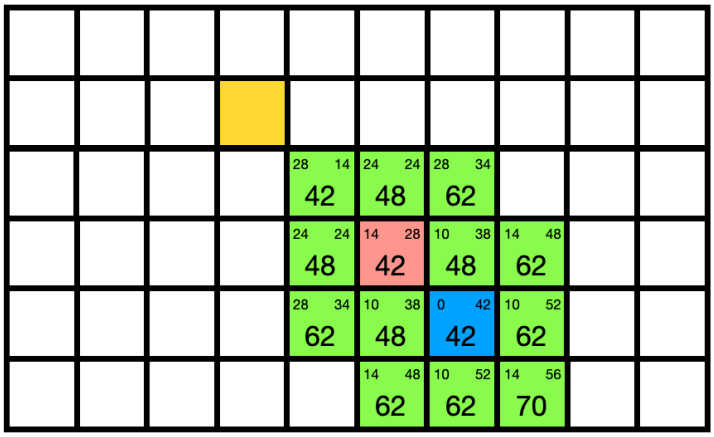


Рисунок 2.1 – Алгоритм пошуку шляху A*

Серед недоліків варто звернути увагу алгоритм A* зберігає всі згенеровані вузли в пам’яті. Складність алгоритму A* визначається кількістю вузлів, які потрібно зберігати в пам'яті під час його виконання. Алгоритм A* зберігає всі згенеровані вузли в пам'яті для подальшого використання під час пошуку шляху [13].

Однак це може стати проблемою в ситуаціях, коли граф дуже великий або коли доступна пам'ять обмежена. Наприклад, у великих сітках або графах міститься велика кількість вузлів, і зберігання їх у пам'яті може призвести до значного використання ресурсів.

Для вирішення цього питання можна розглядати деякі оптимізації або вдосконалення алгоритму A*. Наприклад, використання методів розпаралелювання даних для зменшення обсягу інформації, яка зберігається в кожному вузлу. Також можливе використання технік, які дозволяють динамічно вивільняти пам'ять для вузлів, які вже не потрібні для подальшого пошуку.

Щоб алгоритм A* був більш придатним для обробки великих графів або в умовах обмеженої пам'яті, важливо вдосконалювати його ефективність та робити компроміси між точністю та використанням ресурсів.

2.2. Модифікації алгоритму пошуку A*

Існують декілька існуючих модифікацій алгоритму A*. Розглянемо їх більш детально.

Алгоритм A* та метод оптимального пошуку в глибину об'єднують свої переваги в евристичному методі пошуку, відомому як алгоритм A* (IDA*). Завдяки цьому методу можна знаходити найкоротший маршрут між початковим і кінцевим станом в мережі або дереві. IDA* використовує менше пам'яті порівняно з A*, так як він відстежує лише поточний вузол і пов'язану з ним вартість, а не весь простір пошуку, який був досліджено [14].

З погляду використання пам'яті алгоритм IDA* виявляється більш ефективним. A* зберігає у пам'яті всю область пошуку, що може призвести до значного споживання пам'яті при опрацюванні великих областей. У порівнянні з ним, IDA* економить пам'ять, зберігаючи лише поточний вузол та пов'язану з ним вартість, а не усю область пошуку.

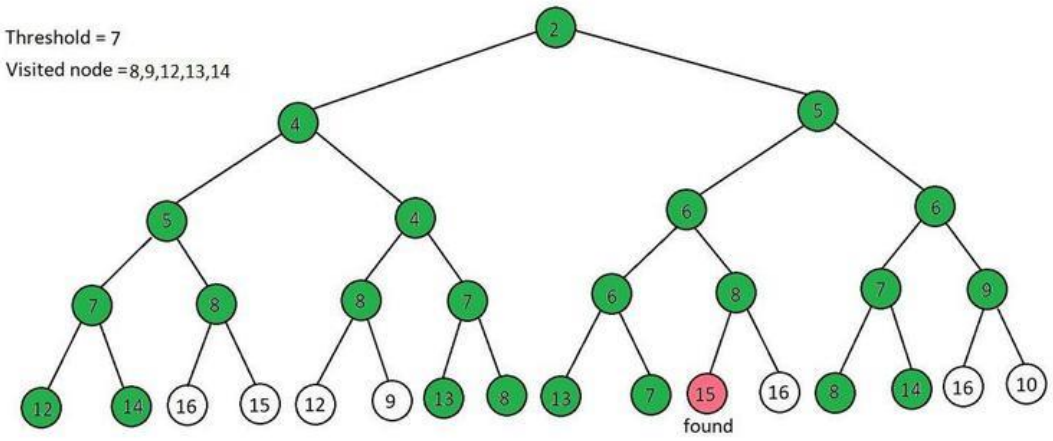


Рисунок 2.2 – Алгоритм IDA*

Для дослідження простору пошуку IDA* використовує пошук в глибину. Він встановлює порогове значення, що дорівнює очікуваній вартості евристичної функції від початкового до кінцевого вузла. Починаючи з цього

значення, алгоритм розширює вузли з ціною, менше або еквівалентної пороговому значенню, використовуючи пошук в глибину. Якщо цільовий вузол знайдено, алгоритм завершується. Якщо порогове значення перевищено, воно підвищується, алгоритм продовжує пошук. Цей процес повторюється до знаходження оптимального рішення.

Алгоритм IDA* реалізується за допомогою таких кроків:

- Початкове обмеження вартості: Починаємо з встановлення початкового обмеження вартості, яке зазвичай рівне евристичній оцінці вартості оптимального шляху до цільового вузла.
- Виконання пошуку в глибину в межах ліміту вартості: Здійснюємо пошук в глибину від початкового вузла до вузла з вартістю, що перевищує поточний ліміт вартості.
- Перевірка цільової вершини: Якщо під час пошуку була знайдена цільова вершина, алгоритм повертає оптимальний шлях до неї.
- Оновлення ліміту вартості: Якщо цільова вершина не знайдена, алгоритм оновлює ліміт вартості до мінімальної вартості будь-якої вершини, яка була розширена під час пошуку.
- Повторення процесу до досягнення мети: Алгоритм повторює цей процес, збільшуючи ліміт вартості, доки не буде досягнута цільова вершина [11].

Weighted A* (вагований A*) – це варіант модифікації алгоритму A* для пошуку шляху в графі, де окремі дії або ребра мають різні ваги чи вартості. У стандартному випадку A* використовується для знаходження найкоротшого шляху, і вага всіх ребер вважається однаковою.

В даному алгоритмі кожне ребро або дія може мати свою вагу. Це може відображати, наприклад, вартість переміщення в конкретному напрямку або складність виконання певної дії. Основна ідея Weighted A* залишається такою ж, як і в класичному A*, але функція f(n) тепер враховує вагу ребра або дії. Формула для f(n) у weighted варіанті може виглядати наступним чином [12]:

$$f(n) = g(n) + w \cdot h(n) \quad (2.1)$$

де:

- g(n) - фактична вартість шляху від початку до поточного вузла,
- h(n) - евристична оцінка вартості шляху від поточного вузла до кінцевого,
- w - вага, яка представляє собою значення ваги ребра або дії.

Weighted A* може бути ефективним інструментом для вирішення задач, де вартість різних шляхів варіюється, і важливо враховувати ці ваги при прийнятті рішення щодо вибору шляху.

*D** або динамічний A* – це алгоритм, який використовує концепції датчиків та пристосовується до динамічних перешкод, вносячи зміни в вагу ребер у реальному часі. Він постійно вирішує завдання обчислення найкоротшого шляху від поточної клітини до початкової, припускаючи, що клітини з невідомим статусом блокування є прохідними. D* підтримує список вузлів, який використовується для поширення інформації про зміни функції вартості в режимі реального часу.

Основна операція D* полягає в підтримці списку вузлів для оцінювання, відомого як OPEN list. Вузли позначаються як такі, що перебувають в одному з декількох станів: NEW, OPEN, CLOSED, RAISE, LOWER.

Алгоритм функціонує ітеративно, вибираючи вершину зі списку OPEN та оцінюючи її. D* розпочинає свій пошук, працюючи в зворотному напрямку від кінцевого вузла до початкового вузла. Кожна вершина, яка розширюється, має зворотний покажчик, що вказує на наступну вершину, яка веде до цільового вузла. Крім того, кожна вершина усвідомлює точну вартість до цільового вузла.

Локальне планування A* (LPA*) представляє собою еволюцію алгоритму A*, спрямовану на довготривале планування (рис. 2.3). Ця

інкрементна версія A* може адаптуватися до змін у графі, не перераховуючи весь граф. Під час поточного пошуку він оновлює значення g (відстань від старту) з попереднього пошуку, коригуючи їх при необхідності. LPA* використовує інформацію з попередніх запусків для значного скорочення кількості вершин, які потрібно досліджувати, порівняно з A*. Він оновлює лише ті значення g, які є важливими для обчислення найкоротшого шляху, що робить його більш ефективним у відношенні до обчислювальних ресурсів.

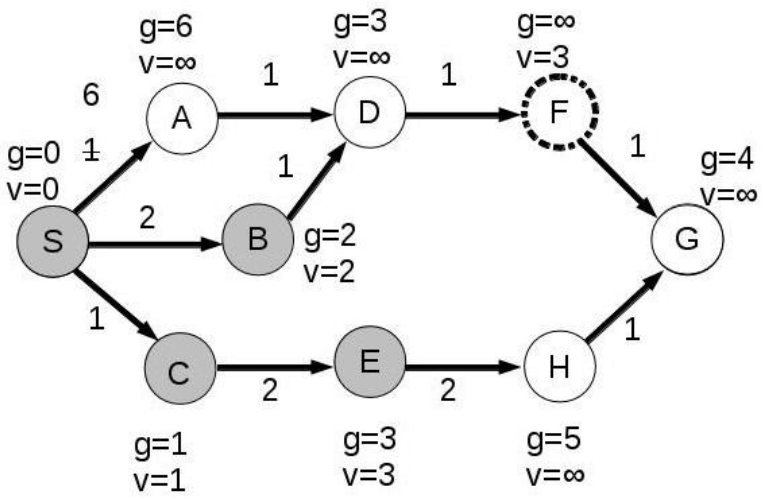


Рисунок 2.3 – Алгоритм LPA*

Принцип алгоритму LPA* (Lifelong Planning A*) базується на ідеї довготривалого планування та інкрементного оновлення інформації у графі. Основні принципи LPA* можна описати наступним чином:

1. Локальне Оновлення: LPA* проводить локальне оновлення значень g-функцій (відстань від старту) для вершин під час кожного пошуку. Це означає, що алгоритм оновлює значення g лише для тих вершин, які є важливими для поточного пошуку, а не для всіх вершин графа.

2. Глобальне Оновлення: Після локального оновлення LPA* може проводити глобальне оновлення, де він переглядає всі вершини і коригує їх значення g відповідно до нових умов. Це дозволяє адаптуватися до змін у графі та виправляти значення g, які можуть бути застарілими.

3. Збереження Інформації: LPA* зберігає інформацію з попередніх пошуків, таку як оптимальні значення g, що дозволяє ефективно зменшити кількість вершин, які потрібно повторно досліджувати під час нових пошуків.

4. Інкрементність: Алгоритм працює інкрементно, оновлюючи лише ті частини графа, які є необхідними для поточного пошуку, тим самим зменшуючи обчислювальні витрати.

5. Об’єднання зворотного та прямого Пошуку: LPA* працює як у прямому, так і у зворотному напрямках, що дозволяє ефективно знаходити шляхи в обидва напрямки у графі.

Ці принципи дозволяють LPA* ефективно пристосовуватися до змін у графі та забезпечувати оптимальні або приблизно оптимальні рішення в умовах довготривалого планування [12].

D* Lite є альтернативною реалізацією ідеї оригінального D*, зберігаючи при цьому його поведінку. Навіть якщо він не базується прямо на оригінальному D*, D* Lite втілює ту саму концепцію. Його привабливість полягає в простоті розуміння та можливості реалізації за меншу кількість коду, що обумовило назву "D* Lite". Цей інкрементальний евристичний алгоритм пошуку ґрунтується на концепціях довготривалого планування A* та повністю заміщує оригінальний D*. Таким чином, використання D* Lite вважається більш доцільним, оскільки він запозичує концепції та ефективно вирішує задачі, для яких раніше використовувався D*.

Спершу, необхідно змінити напрямок пошуку в LPA*. У LPA* пошук ведеться від початкової вершини до кінцевої, і, отже, його значення g вказує на оцінки початкових відстаней. На відміну від цього, D* Lite шукає від кінцевої вершини до початкової, і його значення g є оцінкою відстаней до цілей. Це досягається за допомогою модифікації шляху LPA*, замінюючи початкову та кінцеву вершини і змінюючи всі ребра у псевдокоді на протилежні.

У випадку зміни вартості ребра, D* Lite перераховує найкоротший шлях від поточної клітини до цільової клітини, оновлюючи лише ті відстані до

цілей, які змінилися або не були розраховані раніше. Це означає, що алгоритм обчислює лише g-значення тих вершин, які важливі для перерахунку найкоротшого шляху.

Важливо враховувати, що ефективність алгоритмів суттєво залежить від розташування зміненої вартості ребра відносно розташування цілі. Якщо змінена інформація про вартість знаходиться близько до фронту пошуку, то алгоритми працюють швидко. Однак, якщо змінена інформація розташована близько до цілі, алгоритми можуть значно уповільнитися, оскільки необхідно перерахувати практично всі попередньо розраховані шляхи, які проходять через цей регіон. Такі ситуації можуть призвести до великих витрат часу та ресурсів.

D* проти D* Lite: По-перше, D*-Lite розглядається як значно простіший в порівнянні з D*, і оскільки він завжди працює принаймні так само швидко, він практично витіснив D*.

D* Lite проти A*: D* Lite виконує пошук A* у зворотному порядку, починаючи від цільової точки та намагаючись повернутися до початку. Відмінності від повторного пошуку A* полягають в тому, що D* Lite уникає перепланування з нуля та поступованого відновлення шляху.

LPA* проти D* Lite: У LPA* пошук розпочинається зі стартової позиції, і напрямок пошуку відбувається від початкової конфігурації до кінцевої. У D* Lite сценарій є протилежним: пошук починається з позначеної "Цілі", а напрямок пошуку відбувається від конфігурації цілі до конфігурації старту.

Алгоритм рекурсивного пошуку найкращого (RBFS) є простим рекурсивним методом, спрямованим на імітацію роботи пошуку за зірками. У порівнянні із стандартним пошуком найкращого шляху, який використовує функцію оцінки для узагальненої вартості шляху та евристику, RBFS оперує лише лінійним простором, уникаючи експоненційної складності простору. Структура RBFS подібна до структури рекурсивного пошуку в глибину, що працює на дереві, але, на відміну від безкінечного спуску вниз по поточному шляху, використовує обмеження оцінки для відстеження кращого

альтернативного шляху, доступного від будь-якого предка поточного вузла (рис. 2.3).

Рисунок 2.4 – Робота алгоритму RBFS

Хоча структура RBFS подібна до структури рекурсивного пошуку в глибину, вона використовує обмеження оцінки, щоб відслідковувати найкращий альтернативний шлях, доступний від будь-якого предка поточного вузла. Відмінною особливістю є те, що RBFS, на відміну від A*, не застосовує динамічне програмування, що може призводити до дослідження одних і тих самих вузлів багато разів. Це може становити певний недолік, особливо при опрацюванні великих графів.

Незважаючи на це, RBFS є надійним і оптимальним (з прийнятною евристикою), але йому притаманна проблема надмірної регенерації вузлів через його обмежений обсяг пам'яті, що може призводити до значного збільшення часу обробки. За відповідних умов, RBFS може вирішувати завдання, які A* не здатний розв'язати, завдяки його високій витраті часу для вичерпання пам'яті [13].

Simplified Memory-Bounded A* (SMA*) - це модифікація алгоритму A*, спрямована на обмеження використаної пам'яті. SMA* був розроблений для

вирішення проблеми великих графів, де обмеження пам'яті може стати серйозним обмеженням для традиційного A*.

Основна ідея SMA* полягає в тому, щоб зберігати тільки найважливіші вузли у пам'яті, і в разі потреби видаляти менш важливі. Для цього SMA* використовує поняття "пріоритетів" для визначення, які вузли варто зберігати в пам'яті.

Основні кроки алгоритму SMA*:

1. Ініціалізація: Визначає початковий вузол та інші початкові параметри.
2. Цикл пошуку: Виконується цикл пошуку, де кожна ітерація включає кроки A*.
3. Розширення вузла: Вибирається вузол з відкритого списку за допомогою евристичної функції та оцінки вартості.
4. Оновлення сусідів: Оновлення вартості та шляху для сусідніх вузлів, якщо новий шлях є кращим.
5. Додавання вузла: Додає вибраний вузол до закритого списку, якщо він ще не там.
6. Оцінка пріоритетів: Вузли оцінюються за їхнім значенням пріоритету, який визначається за допомогою евристичної функції та фактичної вартості.
7. Очищення пам'яті: Якщо кількість вузлів в пам'яті перевищує задане обмеження, видаляються вузли з меншим пріоритетом.
8. Перевірка умови завершення: Перевіряється, чи досягнуто кінцевого вузла або іншої умови завершення [14].

Однією з ключових переваг SMA* є те, що він дозволяє ефективно працювати з обмеженим обсягом пам'яті, при цьому зберігаючи оптимальність та повноту алгоритму A*.

2.3 Вдосконалення алгоритму пошуку шляху A*

Для покращення алгоритму пошуку шляху цілком розмуним рішенням оптимізації це використовувати паралельні обчислення**.**

Паралельні обчислення - це організація виконання завдань на окремі потоки одночасно багатоядерної системи. Завдяки цьому розробник пришвидшити роботу програми шляхом застосування розпаралелювання одноманітних завдань на декілька потоків.

Розпаралелення алгоритму A* може відбуватися на різних етапах в залежності від конкретних вимог. Для покращення пропонується розпарелити обчислення параметрів та перевірки сусідів кожно вузла.

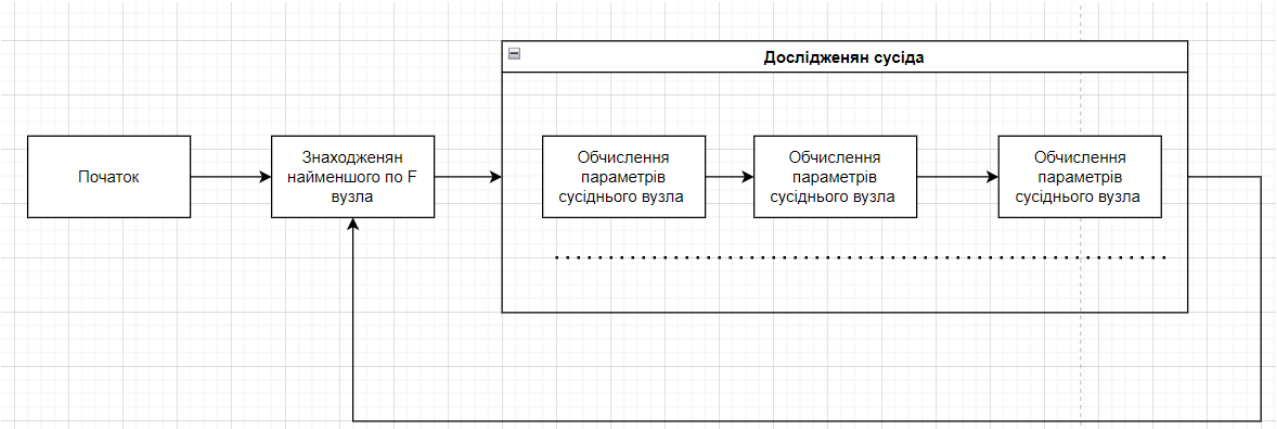


Рисунок 2.5 – Порядок виконання алгоритму.

Згідно послідовності виконання алгоритму сусідніх вузлів може бути декілька, і при реалізації звичайного алгоритму, обчислення будуть виконуватися послідовності, тому пропонується організувати процес виконання, таким чином, щоб обчислення сусідніх вузлів було паралельним.

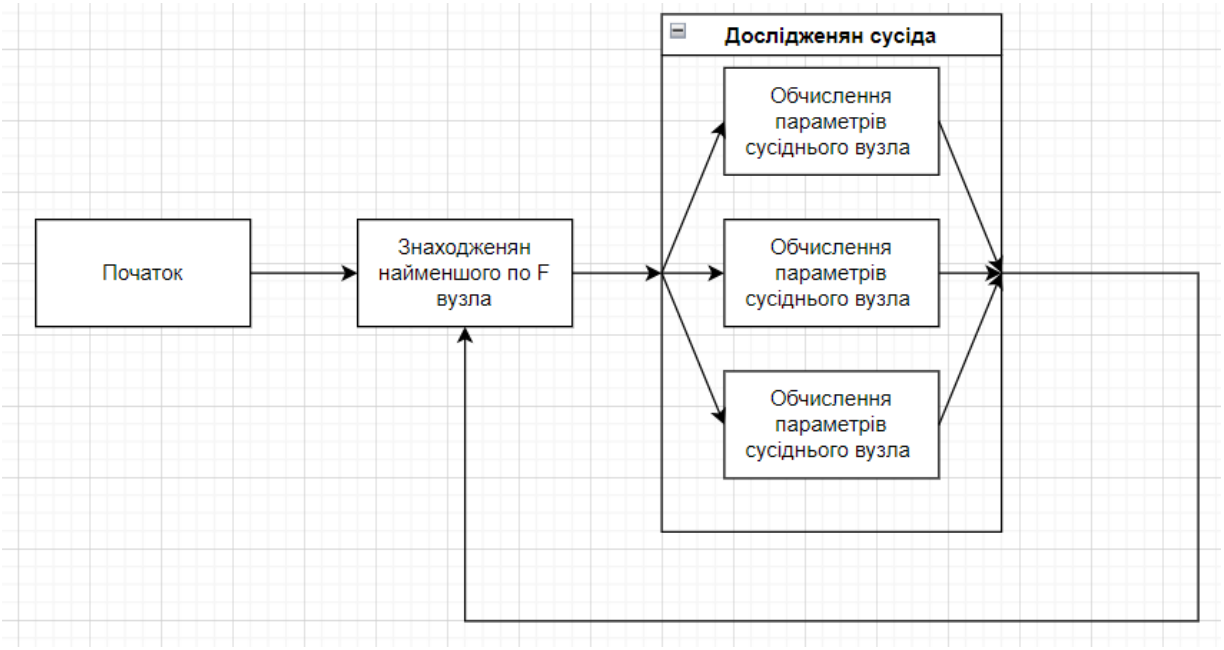


Рисунок 2.6 – Паралельне обчислення сусідів

Згідно блок-схеми швидкість роботи програми значно пришвидшиться якщо одночасно обчислювати декілька сусідніх вузлів.

Мова програмування C# надає широкі можливості створення паралельних обчислень. В C# розпаралелювання можна реалізувати за допомогою різних підходів, але однією з найбільш розповсюджених технік є використання Parallel-класу та бібліотеки Task Parallel Library (TPL) [10].

Parallel-клас надає прості методи для розпаралелювання циклів, ітерацій та інших операцій над колекціями. Один із основних методів цього класу - ForEach.

Task Parallel Library є частиною .NET Framework і надає високорівневий інтерфейс для створення та керування паралельними задачами. Основними елементами є Task та Task<T>, які представляють асинхронні операції.

Наприклад можливою реалізацією паралельних обчислень сусідів може виглядати наступним чином (рис. 2.7):

```
public static void CalculateNeighbors(List<Node> nodes)
{
    Parallel.ForEach(nodes, node =>
    {
        // Обчислення сусідів для кожного вузла
        node.Neighbors = GetNeighbors(node.X, node.Y, nodes);
    });
}
```

Рисунок 2.7 – Паралельне обчислення сусідніх вузлів

Отже важливо розуміти, що ефективність алгоритму може залежати від розміру та складності графа, а також від специфічних умов задачі. Враховуючи це, можливі оптимізації, такі як розпаралелювання обчислень, можуть допомогти поліпшити продуктивність алгоритму в різних умовах.

Серед покращень алгоритму також можна застосувати паралельний пошук шляху зі зворотної сторони починаючи з кінцевого вузла до стартового.

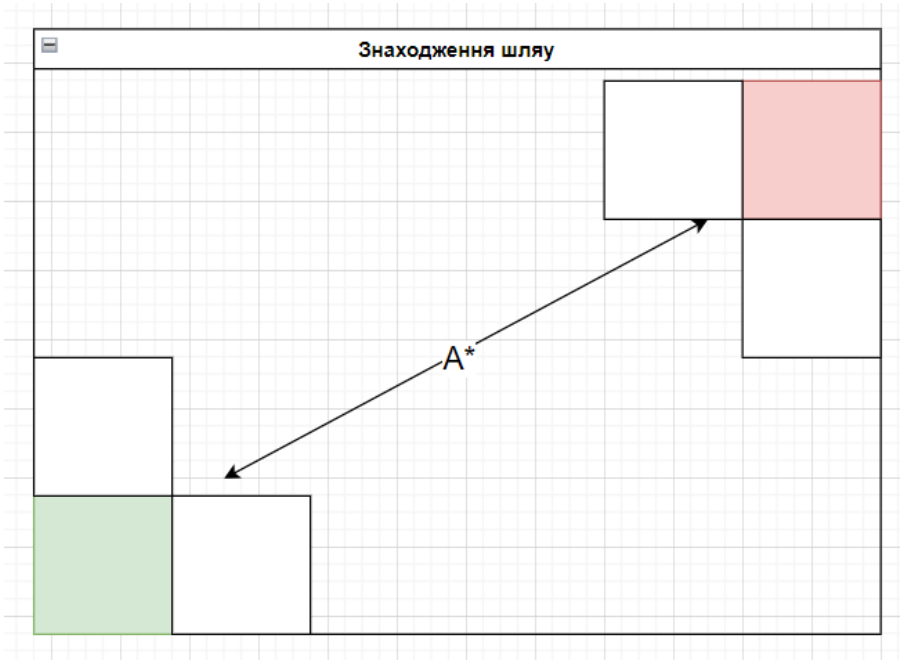


Рисунок 2.8 – Зворотній напрямок шляху

Завдяки такому вдосконалення фактично робота алгоритму поділена навпіл. Додатком до цього може слугувати і паралельне виконання у зворотному напрямку, що пришвидшить пошук шляху.

Можливим також є покращення додавання штучного інтелекту, що надає більшої гнучкості в знаходженні шляху, але значно навантажує систему та зменшує швидкість знаходження шляху, що є одним з головних недоліків алгоритму. Тому доцільно вважати, що для фіксованих не динамічних систем не застосовувати штучний інтелект для побудови шляху.

2.4 Висновок до другого розділу

Під час розгляду алгоритму пошуку найкоротшого шляху в іграх на основі методу A*, отримано важливі уявлення про ефективність та принципи функціонування цього алгоритму. A* представляє собою збалансований метод, який використовується для ефективного пошуку шляху.

Основні принципи A*, такі як використання евристичної функції та ефективні етапи роботи, роблять його ефективним інструментом для задач пошуку шляху в іграх. Особливу увагу слід звернути на правильний вибір евристичної функції, яка визначає точність та продуктивність алгоритму.

Інші модифікації, такі як IDA*, D*-Lite, та SMA*, розширюють можливості алгоритму A*, надаючи більшу гнучкість у вирішенні конкретних завдань. Кожен із цих алгоритмів має свої переваги та недоліки, і вибір конкретного залежить від конкретних вимог та характеристик задачі.

Загалом, вивчення алгоритмів створення шляху в іграх на основі методу A* дозволило отримати глибше розуміння принципів їх роботи та визначити оптимальні стратегії вирішення задач даного класу.

РОЗДІЛ 3 ПРОЕКТУВАННЯ ТА РОЗРОБКА АЛГОРИТМУ ПОШУКУ ШЛЯХУ В ПРОГРАМОВАНИХ ОБ'ЄКТАХ ГРИ

3.1 Інструменти реалізації алгоритму

Алгоритм A* – є ефективним алгоритмом пошуку найкоротшого шляху в графі або сітці, і використовується для розв'язання проблем шляхоутворення. A* надає можливість ефективно знаходити найкоротший шлях у графах або сітках, що робить його відмінним вибором для задач штучного інтелекту, геоінформаційних систем, комп'ютерних ігор і робототехніки.

Для реалізації алгоритму пошуку використовувалася мова програмування C#. Дана мова програмування вбудована в ігровий рушій Unity, що дозволяє інтегрувати даний алгоритм в ігрове середовище. Мова програмування C#, дозволяє розділити взаємодіючі об’єкти алгоритму на окремі складові – класи, це через те що дана мова програмування надає можливості реалізовувати принципи ООП (Об’єктно орієнтованого програмування).

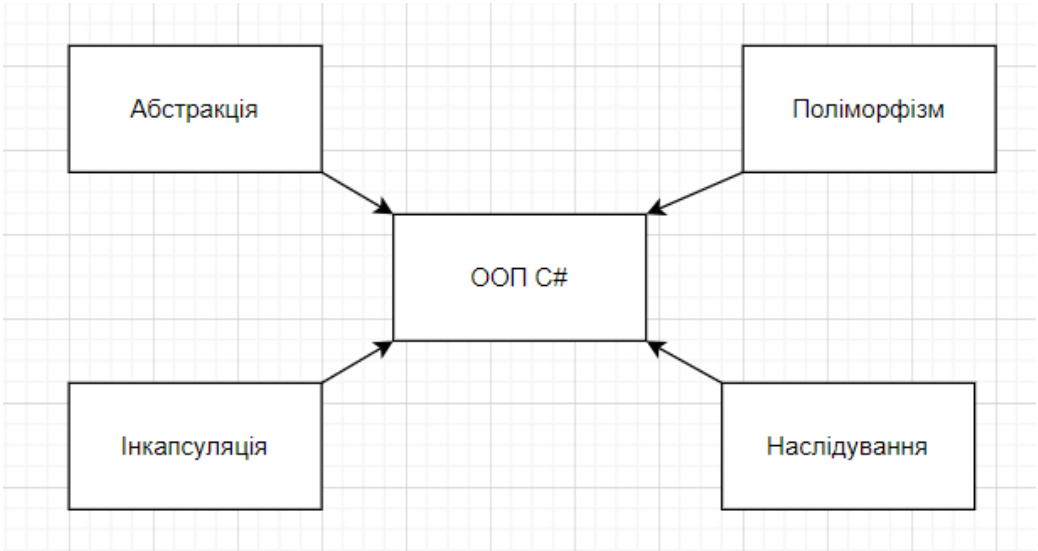


Рисунок 3.1 – ООП

Основним інструментом для візуалізації слугуватиме ігровий рушій Unity. Unity - це потужний ігровий рушій, який почав свою історію як інструмент для розробки ігор, але з часом став ширше використовуваним для розробки різноманітних візуальних додатків, таких як симулятори, навчальні програми, архітектурні візуалізації та інші проекти.

Даний рушій має широкі можливості з рендернингу візуальних ефектів, створення компонентів побудови ігрових світів та широка можливість створення алгоритмів, чи ручного програмування спец ефектів. Unity має широкі можливість по розробці крос-платформенних додатків, через що має значну популярність як рушій для розробки мобільних додатків (рис. 3.2).

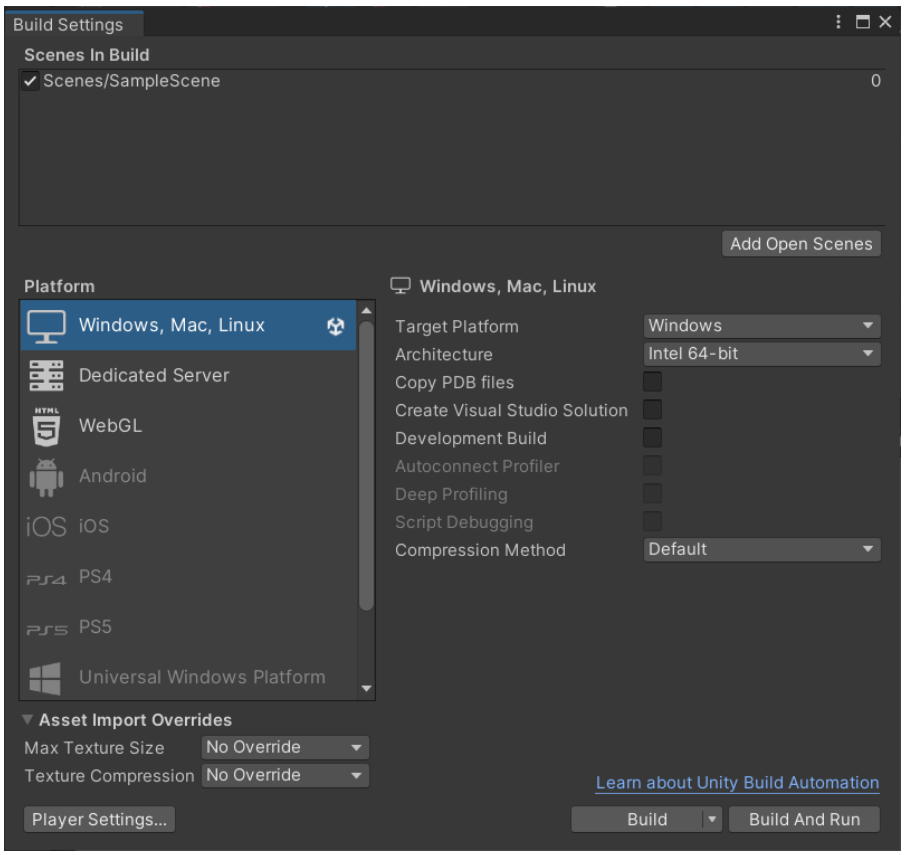


Рисунок 3.2 – Платформи розробки Unity

Більшість інструментів з моделювання такі як Blender, MagicaVoxel, сумісні з рушієм, що дозволяє значно простіше створювати графічні об’єкти та візуальні ефекти [9].

Завдяки можливості налаштуванню інтерфейсу, Unity постає доволі комфортним середовищем для розробки мобільних додатків (рис. 3.3).

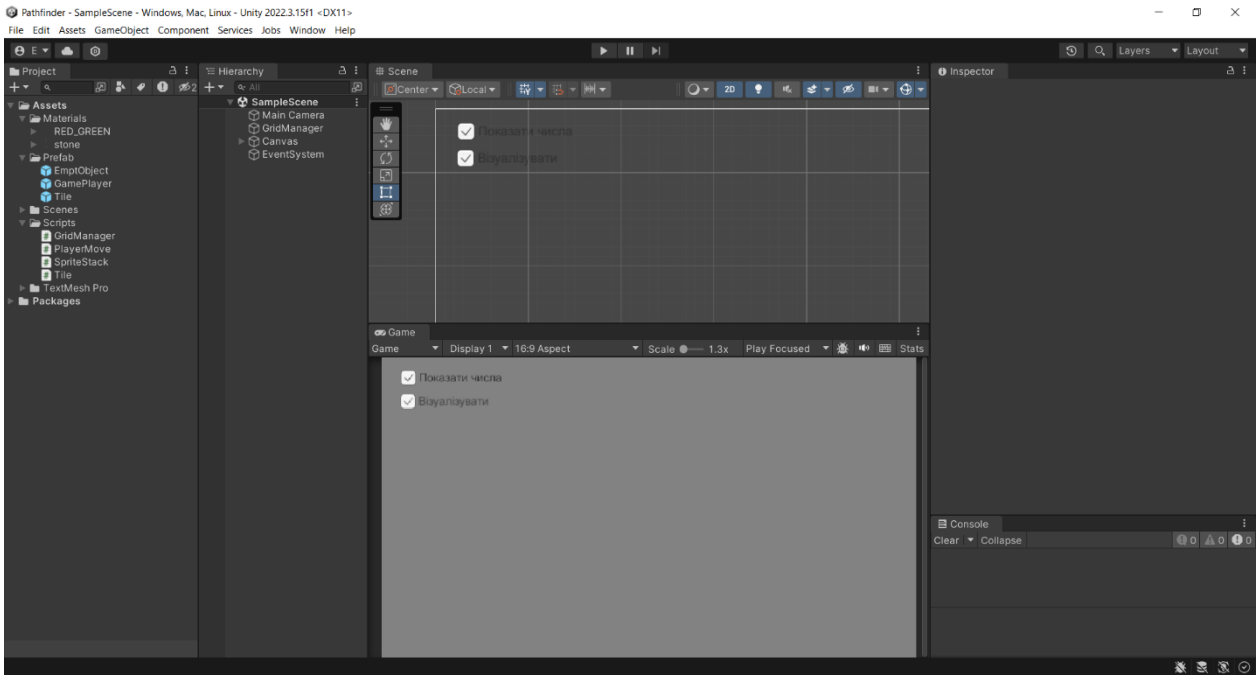


Рисунок 3.3 – Інтерфейс рушія Unity

Для реалізації алгоритму, потрібно створити ключові елементи, це поле з вагами, що означають вартість переходу на той чи інший елемент, реалізація функції алгоритму - A* Pathfinding (рис. 3.4).

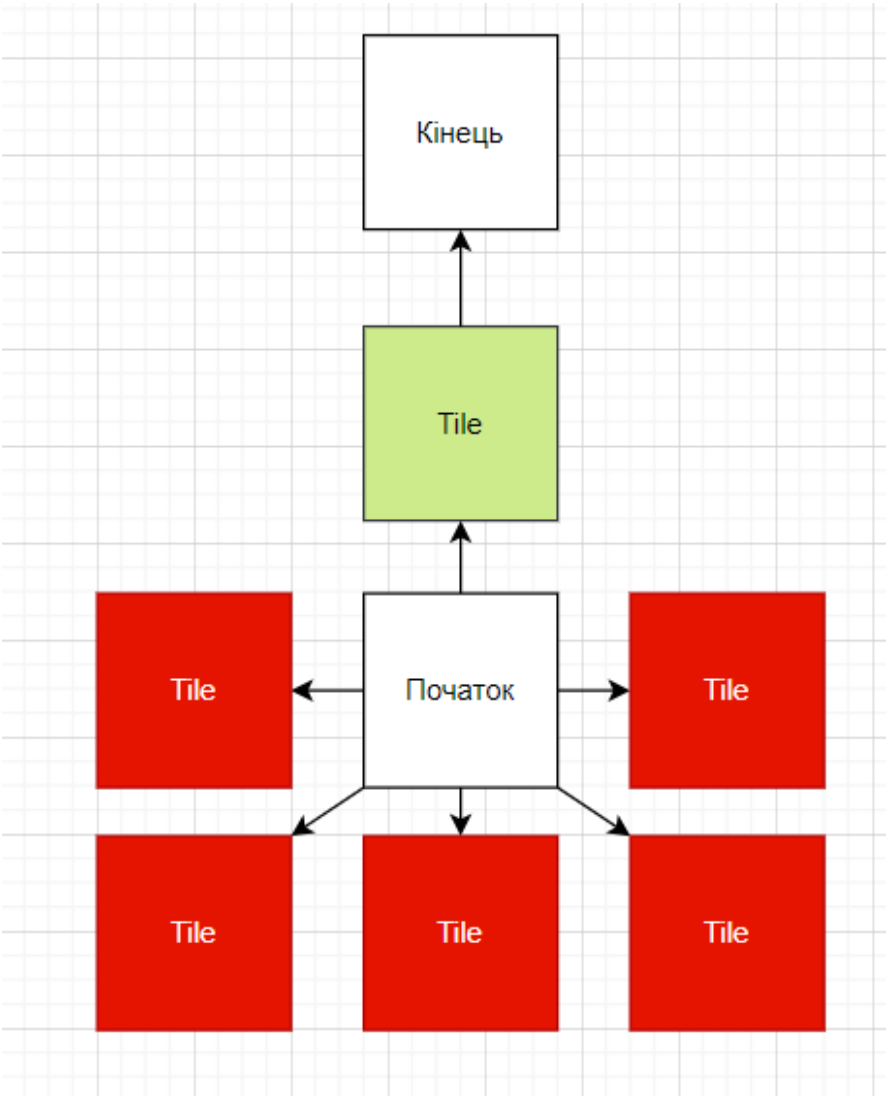


Рисунок 3.4 – Компоненти алгоритму A* Pathfinding.

Алгоритм A* використовується для знаходження найкоротшого шляху у графі з вагами. Початкова та кінцева точки визначаються, і створюються два списки вузлів - відкритий та закритий. Кожному вузлу призначаються значення g (вартість пройденого шляху), h (відстань від поточного вузла до кінцевого) і f (сума g та h).

Алгоритм входить у цикл, який продовжується до тих пір, поки відкритий список не порожній. На кожному кроці обирається вузол з найменшим значенням f, його переміщують з відкритого списку в закритий. Для кожного сусіднього вузла перевіряється, чи він вже відомий чи не відомий. У випадку нового вузла його додають до відкритого списку та

обчислюють значення f. Якщо вузол вже відомий, перевіряється, чи новий шлях до нього коротший, і відбувається оновлення значень, якщо це так.

Алгоритм завершує роботу, якщо досягнута кінцева точка або відкритий список порожній, вказуючи відсутність шляху (рис. 3.5).

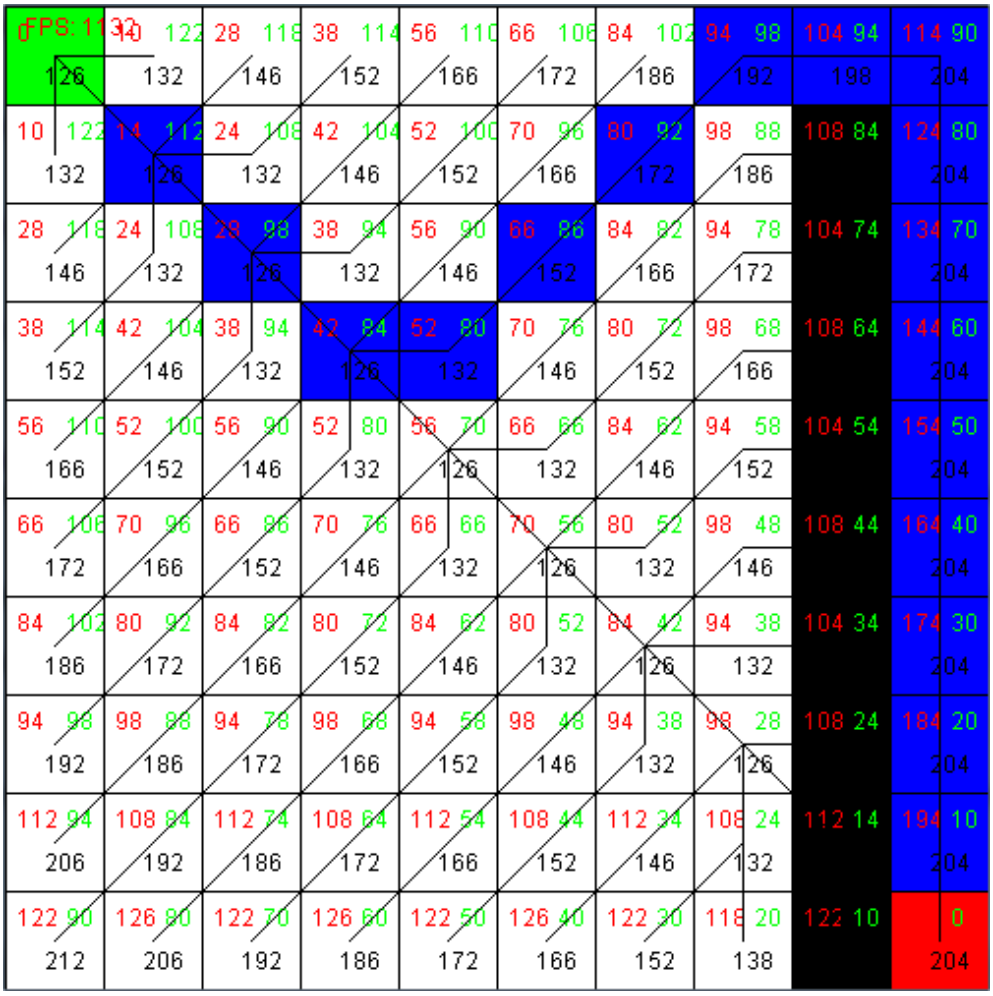


Рисунок 3.5 – Графічне зображення алгоритму.

Якщо представити алгоритм у вигляді блок-схеми то вона представлятиме собою наступний вигляд (рис. 3.6):

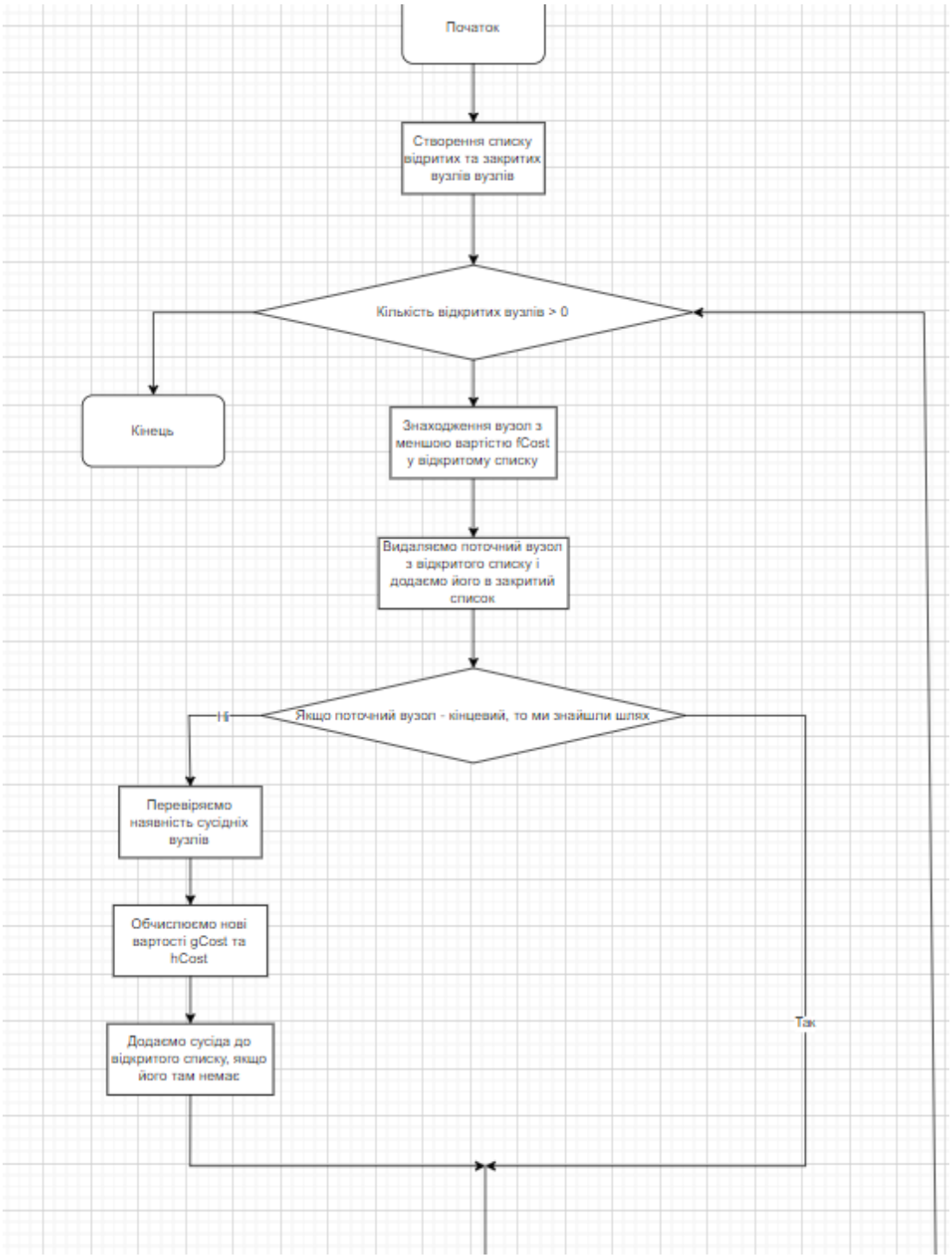


Рисунок 3.6 – Алгоритм пошуку шляху.

Для реалізації даного алгоритму потрібно виконати наступні вимоги:

6) Реалізувати систему вузлів, де кожен компонент має свої параметри обчслення відстані від початку шляху та його кінця.
7) Створити функцію побудови шляху у вигляді програмного коду.
8) Зобразити роботу алгоритму.

3.2 Розробка алгоритму пошуку шляху

Проект пошуку шляху має наступну структуру:

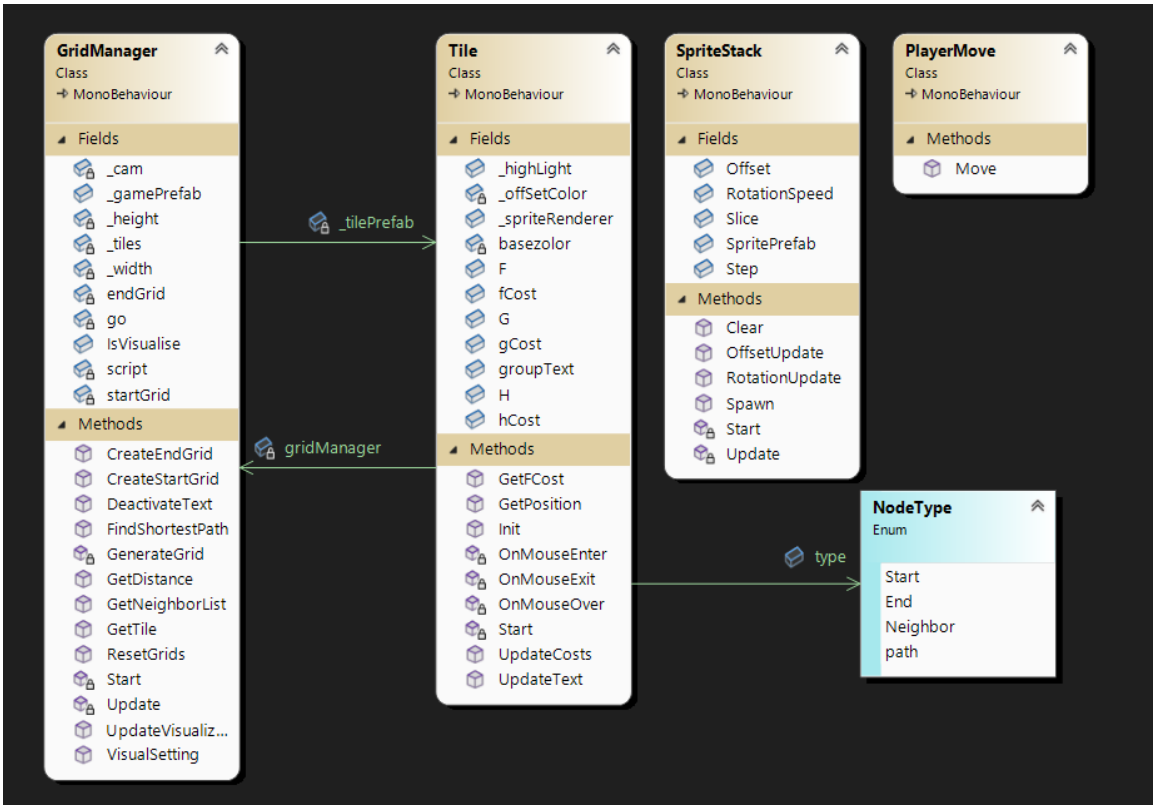


Рисунок 3.7 – Структура проекту пошуку шляху

Щоб створити сітку використовується клас GridManager.cs, що відповідає за створення сітки та алгоритму пошуку шляху. Для того щоб створити сітку вузлів використовується наступна функція (рис 3.8):

```
void GenerateGrid()
{
    for(int x = 0; x < _width; x++)
    {
        for(int  y = 0; y < _height; y++)
        {
            var tileSpawn = Instantiate(_tilePrefab, new Vector2(x,y), Quaternion.identity);
            tileSpawn.name = $"Tile {x} {y}";
            var isOffSet = (x % 2 == 0 && y % 2 != 0) || (x % 2 != 0 && y % 2 == 0);
            _tilePrefab.Init(isOffSet);
            _tiles.Add(tileSpawn);
        }
    }
    _cam.transform.position = new Vector3((float)_width / 2 - 0.5f, (float)_height / 2 - 0.5f, -10);
    _tiles[0].type = NodeType.Start;
    _tiles[70].type = NodeType.End;


    UpdateVisualization(_tiles[0]);
    UpdateVisualization(_tiles[70]);
}
```

Рисунок 3.8 – Функція GenerateGrid

Функція GenerateGrid відповідає за створення ініціальної сітки тайлів на сцені.

Створення Тайлів: У вкладених циклах for проганяється по всіх комбінаціях x та y, створюючи нові тайли для кожної позиції у сітці. Кожен тайл ініціалізується та додається до списку _tiles (рис. 3.9).

```
for(int x = 0; x < _width; x++)
{
    for(int  y = 0; y < _height; y++)
    {
        var tileSpawn = Instantiate(_tilePrefab, new Vector2(x,y), Quaternion.identity);
        tileSpawn.name = $"Tile {x} {y}";
        var isOffSet = (x % 2 == 0 && y % 2 != 0) || (x % 2 != 0 && y % 2 == 0);
        _tilePrefab.Init(isOffSet);
        _tiles.Add(tileSpawn);
    }
}
```

Рисунок 3.9 – Цикл створення сітки

Після створення тайлів, позиція камери _cam оновлюється так, щоб бути у центрі сітки, та встановлюється стартова та кінцева точка шляху (рис. 3.10):

```
_cam.transform.position = new Vector3((float)_width / 2 - 0.5f, (float)_height / 2 - 0.5f, -10);
_tiles[0].type = NodeType.Start;
_tiles[70].type = NodeType.End;
```

Рисунок 3.10 – Встановлення положення камери

Для створення сітки використовується компонент Tile (рис. 3.11):

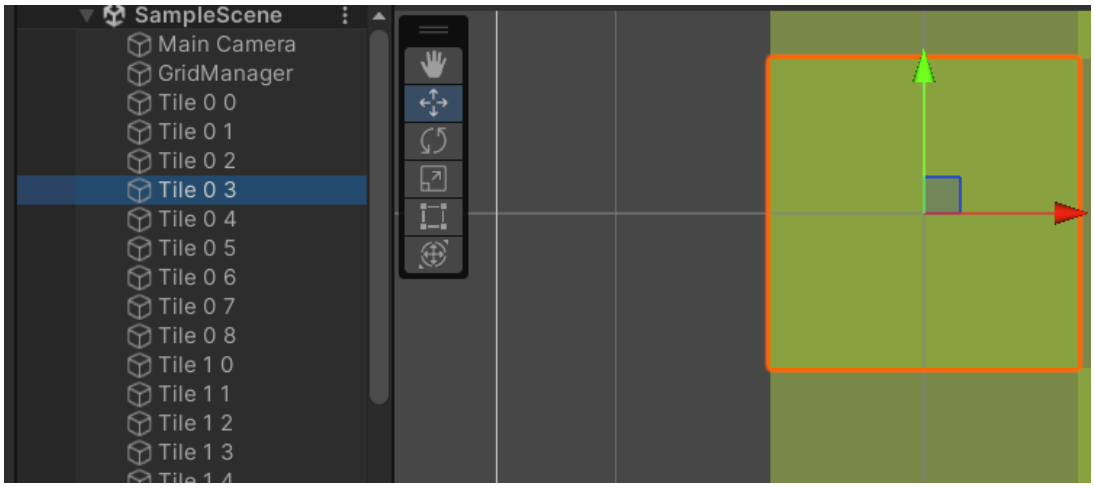


Рисунок 3.11 – Компонент Tile

Кожен з цих компонентів має клас Tile, який в свою чергу має декілька важливих функцій та параметрі. Серед них найважливішими є це оновлення параметрів gCost та hCost, отримання координат компоненти, та розрахування основної величини fCost (рис. 3.12).

```
public class Tile : MonoBehaviour
{
    //Текст
    public TextMeshProUGUI G;
    public TextMeshProUGUI H;
    public TextMeshProUGUI F;
    public GameObject groupText;
    // Start is called before the first frame update
    [SerializeField] private Color basezolor, _offSetColor;
    public GameObject _highLight;
    public SpriteRenderer _spriteRenderer;
    public NodeType type;
    public float gCost;
    public float hCost;
    public float fCost;
    private GridManager gridManager;

    Unity Message | 0 references
    private void Start()
    {
        gridManager = GameObject.Find("GridManager").GetComponent<GridManager>();
        G.text = gCost.ToString();
        H.text = hCost.ToString();
        F.text = fCost.ToString();
    }
    2 references
    public void Init(bool isOffset)
    {
        _spriteRenderer.color = isOffset ? _offSetColor : basezolor;
    }
    34 references
    public Vector3 GetPosition()
    {
        return this.transform.position;
    }
    5 references
    public float GetFCost()
    {
        fCost = gCost + hCost;
        return fCost;
    }
    1 reference
```

Рисунок 3.12 – Клас Tile

Кожен вузол поділяється на окремі категорії:

1) Звичайні вузли
2) Стартовий вузол
3) Кінцевий вузол
4) Вузли сусіди

Згідно цих категорій, вузли позначаються відповідним кольором для зображення процесу обчислення шляху. Для зручності використовується тип об’єкту enum NodeType (рис. 3.13)

```
public enum NodeType
{
    Start,
    End,
    Neighbor,
    path
}
```

Рисунок 3.13 - Enum NodeType

Завдяки цим параметрам значно легше визначати тип вузла та відносити її до певної категорій. В залежності від типу функція UpdateVisualization, визначає колір вузла за відповідним типом (рис. 3.14).

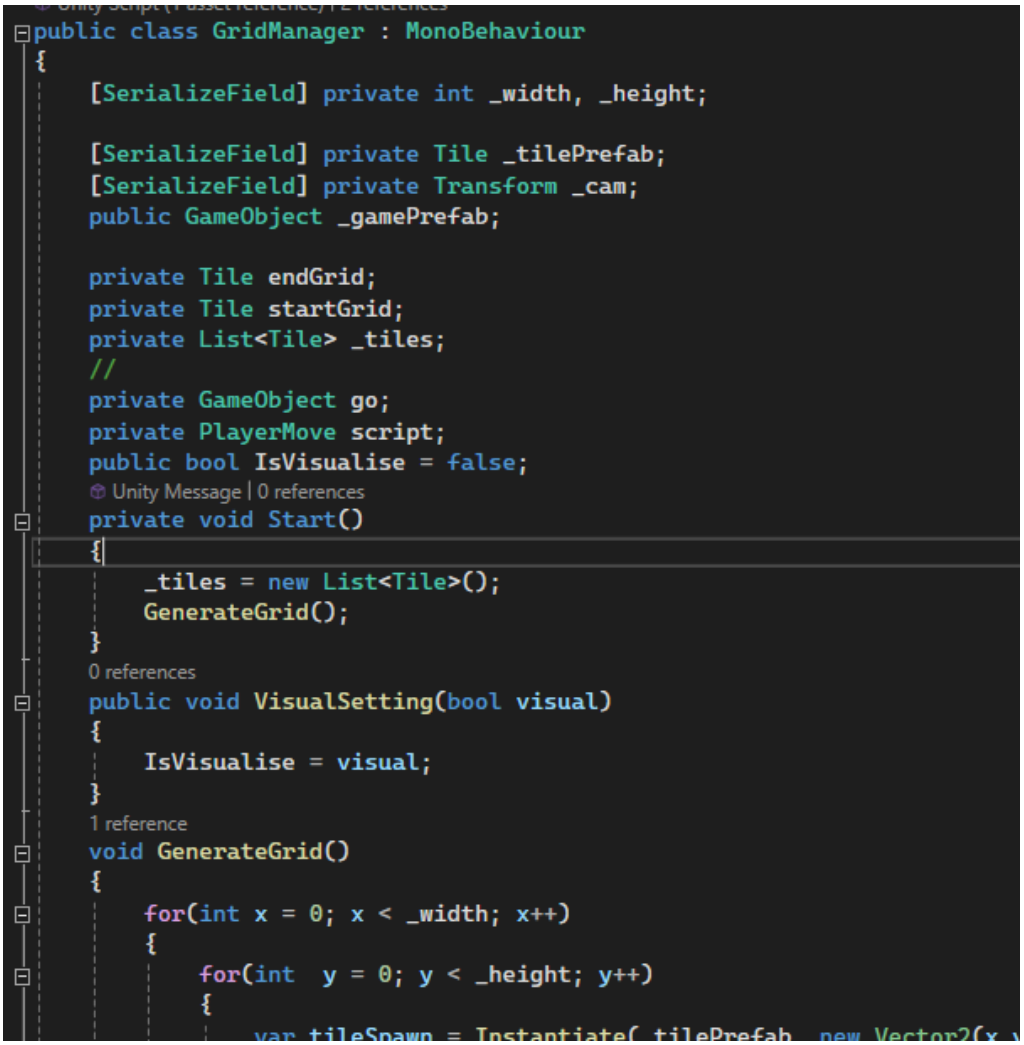
```
public void UpdateVisualization(Tile nodeTile)
{

    //Renderer renderer = GetComponent<Renderer>();
    NodeType nodeType = nodeTile.type;
    switch (nodeType)
    {
        case NodeType.Start:
            nodeTile._spriteRenderer.color = Color.green;
            break;

        case NodeType.End:
            nodeTile._spriteRenderer.color = Color.red;
            break;
        case NodeType.Neighbor:
            nodeTile._spriteRenderer.color = Color.blue;
            break;
        case NodeType.path:
            nodeTile._spriteRenderer.color = Color.green;
            break;
    }
}
```

Рисунок 3.14 – Функція UpdateVisualization

Наступний клас який потрібно розглянути GridManager (рис. 3.15).



```
public class GridManager : MonoBehaviour
{
    [SerializeField] private int _width, _height;

    [SerializeField] private Tile _tilePrefab;
    [SerializeField] private Transform _cam;
    public GameObject _gamePrefab;

    private Tile endGrid;
    private Tile startGrid;
    private List<Tile> _tiles;
    //
    private GameObject go;
    private PlayerMove script;
    public bool IsVisualise = false;
    Unity Message | 0 references
    private void Start()
    {
        _tiles = new List<Tile>();
        GenerateGrid();
    }
    0 references
    public void VisualSetting(bool visual)
    {
        IsVisualise = visual;
    }
    1 reference
    void GenerateGrid()
    {
        for(int x = 0; x < _width; x++)
        {
            for(int  y = 0; y < _height; y++)
            {
                var tileSpawn = Instantiate( tilePrefab, new Vector2(x,
```

Рисунок 3.15 – Клас GridManager

Даний клас визначає структур сітки та містть в собі алгоритм пошуку шляху між двома вузлами. Перед тим як розглянути функції пошуку шляху, потрібно звернути увагу спочатку на додаткові функції які забезпечують обчислення вагів, виявлення сусідів та відстані 2 об’єктів на сітці.

Для визначення відстані між 2 вузлами використовується функція GetDistance (рис. 3.16):

```
public float GetDistance(Tile nodeA, Tile nodeB)
{
    float dstX = Mathf.Abs(nodeA.GetPosition().x - nodeB.GetPosition().x);
    float dstY = Mathf.Abs(nodeA.GetPosition().y - nodeB.GetPosition().y);
    return Mathf.Sqrt(dstX * dstX + dstY * dstY);
}
```

Рисунок 3.16 – Функція GetDistance

Наступна функція використовується для визначення конкретного вузла за координатами x,y серед всіх вузлів (рис. 3.17).

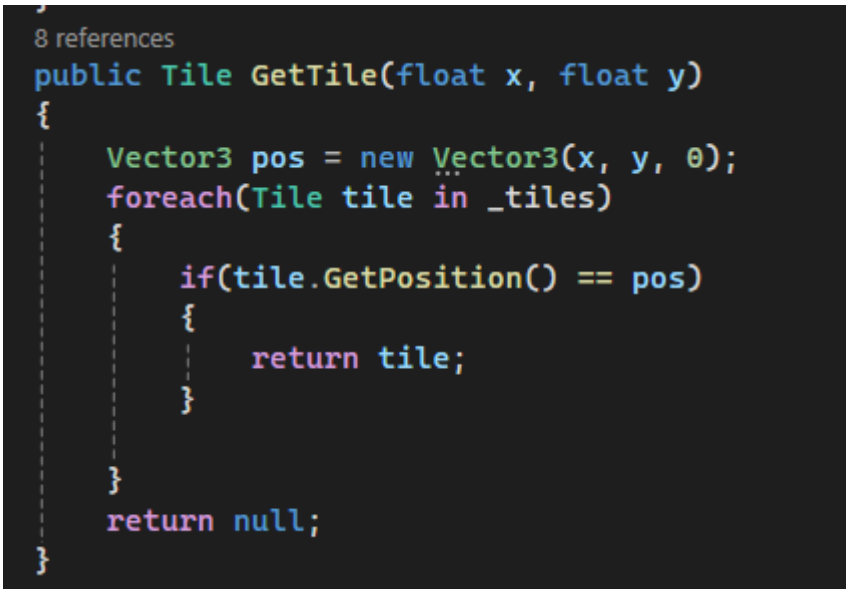


Рисунок 3.17 – Функція GetTile

Функція GetNeighborList (рис. 3.18) призначена для отримання списку сусідніх тайлів для заданого тайла currentNode у сітці.

Функція спочатку створює пустий список neighborList, в який потім додаються сусіди відповідно до різних напрямків: вліво, вправо, вгору, вниз та по діагоналі.

Умовні перевірки визначають, чи можна додати тайл у вказаному напрямку. Наприклад, для лівого напрямку перевіряється, чи тайл знаходиться на границі сітки по горизонталі, а для діагональних напрямків - чи тайл не виходить за межі і по вертикалі, і по горизонталі.

Після визначення сусідів, функція проходиться по списку сусідів у циклі foreach і встановлює їм тип NodeType.Neighbor. Крім того, викликається метод UpdateVisualization, щоб оновити візуальне представлення цих тайлів.

На завершення, функція повертає список сусідів neighborList.

```
1 reference
public List<Tile> GetNeighborList(Tile currentNode)
{
    List<Tile> neighborList = new List<Tile>();

    if (currentNode.GetPosition().x - 1 >= 0)
    {
        neighborList.Add(GetTile(currentNode.GetPosition().x - 1, currentNode.GetPosition().y)); //Left
        if (currentNode.GetPosition().y - 1 >= 0) neighborList.Add(GetTile(currentNode.GetPosition().x - 1, currentNode.GetPosition().y - 1)); //Le
        if (currentNode.GetPosition().y + 1 < _height) neighborList.Add(GetTile(currentNode.GetPosition().x - 1, currentNode.GetPosition().y + 1));
    }
    if (currentNode.GetPosition().x + 1 < _width)
    {
        neighborList.Add(GetTile(currentNode.GetPosition().x + 1, currentNode.GetPosition().y)); //Right
        if (currentNode.GetPosition().y - 1 >= 0) neighborList.Add(GetTile(currentNode.GetPosition().x + 1, currentNode.GetPosition().y - 1)); //Ri
        if (currentNode.GetPosition().y + 1 < _height) neighborList.Add(GetTile(currentNode.GetPosition().x + 1, currentNode.GetPosition().y + 1));
    }
    if (currentNode.GetPosition().y - 1 >= 0) neighborList.Add(GetTile(currentNode.GetPosition().x, currentNode.GetPosition().y - 1)); //Down
    if (currentNode.GetPosition().y + 1 <_height) neighborList.Add(GetTile(currentNode.GetPosition().x, currentNode.GetPosition().y + 1)); //Up

    foreach (Tile neighbor in neighborList) {
        neighbor.type = NodeType.Neighbor;
        UpdateVisualization(neighbor);
    }
    return neighborList;
}
```

Рисунок 3.18 – Функція GetNeighborList

Остання і найголовніша функція (рис. 3.19) – FindShortestPath, що й відповідає за обчислення та побудови шляху між 2 віддаленими вузлами.

```
1 reference
public List<Tile> FindShortestPath(Tile startNode, Tile endNode)
{
    // Ініціалізація списку відкритих вузлів
    List<Tile> openSet = new List<Tile>();
    // Ініціалізація списку вже розглянутих вузлів
    List<Tile> closedSet = new List<Tile>();

    // Додавання початкового вузла до списку відкритих
    openSet.Add(startNode);
    while (openSet.Count > 0)
    {
        // Обираємо вузол з найменшою вартістю fCost
        Tile currentNode = openSet[0];

        // Шукаємо вузол з меншою вартістю fCost у відкритому списку
        for (int i = 1; i < openSet.Count; i++)
        {
            if (openSet[i].GetFCost() < currentNode.GetFCost() ||
                (openSet[i].GetFCost() == currentNode.GetFCost() && openSet[i].hCost < currentNode.hCost))
            {
                currentNode = openSet[i];
            }
        }
        // Видаляємо поточний вузол з відкритого списку і додаємо його в закритий список
        openSet.Remove(currentNode);
        closedSet.Add(currentNode);
        foreach(Tile tile in closedSet)
        {
            tile.type = NodeType.path;
            UpdateVisualization(tile);
        }
        // Якщо поточний вузол - кінцевий, то ми знайшли шлях
        if (currentNode == endNode)
        {
            return null;
        }
```

Рисунок 3.19 – Функція FindShortestPath

Даний алгоритм створює 2 списки, це список вже розглянутих вузлів, та ті що потрібно розгялнути: List<Tile> closedSet та List<Tile> openSet (рис. 3.20).

```
// Ініціалізація списку відкритих вузлів
List<Tile> openSet = new List<Tile>();
// Ініціалізація списку вже розглянутих вузлів
List<Tile> closedSet = new List<Tile>();
```

Рисунок 3.20 – Списки closedSet та openSet

Алгоритм починає зі стартового вузла, додає його в список вузлів openSet, та знаходить в цьому списку вузол, що має найменше значення F (рис. 3.21). Якщо такий вузол знаходиться то його додають до списку closedSet, що має всі вузли побудованого шляху, та позначають їх зеленим кольором.

```
while (openSet.Count > 0)
{
    // Обираємо вузол з найменшою вартістю fCost
    Tile currentNode = openSet[0];

    // Шукаємо вузол з меншою вартістю fCost у відкритому списку
    for (int i = 1; i < openSet.Count; i++)
    {
        if (openSet[i].GetFCost() < currentNode.GetFCost() ||
            (openSet[i].GetFCost() == currentNode.GetFCost() && openSet[i].hCost < currentNode.hCost))
        {
            currentNode = openSet[i];
        }
    }
    // Видаляємо поточний вузол з відкритого списку і додаємо його в закритий список
    openSet.Remove(currentNode);
    closedSet.Add(currentNode);
    foreach(Tile tile in closedSet)
    {
        tile.type = NodeType.path;
        UpdateVisualization(tile);
    }
    // Якщо поточний вузол - кінцевий, то ми знайшли шлях
```

Рисунок 3.21 – Цикл знаходження вузла по F

Далі від відповідного вузла знаходиться його сусіди, та додаються відповідним синім кольором відповідно до його типу. Останнім кроком

оновлюються всі числові параметри G,H,F, та знову знаходимо найменший вузол по величині F (рис. 3.22).

Алгоритм повинен пройти декілька циклів до тих пір доки не виконається умова, що конкретний вузол співпадає з кінцевим.

```
// Якщо поточний вузол - кінцевий, то ми знайшли шлях
if (currentNode == endNode)
{
    return null;
}
foreach (Tile neighbor in GetNeighborList(currentNode))
{
    if (closedSet.Contains(neighbor))
        continue;

    // Обчислюємо нові вартості gCost та hCost
    float newGCost = currentNode.gCost + GetDistance(currentNode, neighbor);

    if (newGCost < neighbor.gCost || !openSet.Contains(neighbor))
    {
        // Оновлюємо вартості та попередника вузла
        neighbor.UpdateCosts(newGCost, GetDistance(neighbor, endNode));

        // Додаємо сусіда до відкритого списку, якщо його там немає
        if (!openSet.Contains(neighbor))
            openSet.Add(neighbor);
    }
}
```

Рисунок 3.22 – Цикл перебору сусідів

Для створення стартової та кінцевої точки використовуються функції: OnMouseEnter, OnMouseOver , OnMouseExit. Дані функції фіксують положення курсору миші, та автоматично оновлюють дані та надають координати положення вузла (рис. 3.23).

```
private void OnMouseEnter()
{
    _highLight.SetActive(true);
}
Unity Message | 0 references
private void OnMouseOver()
{
    if (Input.GetMouseButtonDown(0))
    {
        gridManager.CreateStartGrid(GetPosition());
    }
    if(Input.GetMouseButtonDown(1))
    {
        gridManager.CreateEndGrid(GetPosition());
    }
}
Unity Message | 0 references
private void OnMouseExit()
{
    _highLight.SetActive(false);
}
```

Рисунок 3.23 – Функції що фіксують положення курсору

Далі можна спостерігати (рис. 3.21), що викликаються наступні функції: CreateStartGrid, CreateEndGrid.

Функція CreateStartGrid використовується для позначення стартового вузла побудови шляху. Спочатку оновлюється сітка за допомогою функції ResetGrids(). Після цього знаходиться відповідний об’єкт по координатам та присвоюється відповідний тип стартового вузла. Далі створюється ігровий об’єкт, що поміщається на координати стартового вузла (рис. 3.24).

```
public void CreateStartGrid(Vector3 position)
{
    ResetGrids();
    startGrid = GetTile(position.x, position.y);
    startGrid.type = NodeType.Start;
    go = Instantiate(_gamePrefab, position, Quaternion.identity);
    script = go.GetComponent<PlayerMove>();
    UpdateVisualization(startGrid);
}
```

Рисунок 3.24 – Функція CreateStartGrid

Відповідно функція CreateEndGrid позначає кінцевий вузол шляху, який треба визначити (рис. 3.25):

```
1 reference
public void CreateEndGrid(Vector3 position)
{
    endGrid = GetTile(position.x,position.y);
    endGrid.type = NodeType.End;
    UpdateVisualization(endGrid);
}
```

Рисунок 3.25 – Функція CreateEndGrid

Після того як було встановлено стартовий та кінцевий вузли та обчислено шлях, об’єкт повинен рухатися, для цього передбачена функція Move.

```
1 reference
public void Move(List <Tile> tiles)
{
    foreach (Tile tile in tiles)
    {
        transform.position =  tile.GetPosition();
    }
}
```

Рисунок 3.26 – Функція Move

Дана функція має доволі просту реалізацію, в функції просто перебираються всі вузли розрахованого шляху визначаються їхні координати, та по ним переміщують сам ігровий об’єкт.

Для створення зовнішнього вигляду ігрового об’єкту використовується клас SpriteStack (рис. 3.27).

```
public class SpriteStack : MonoBehaviour
{
    // Start is called before the first frame update
    public Sprite[] Slice;
    public GameObject SpritePrefab;
    public Vector2 Offset;
    public Vector2 Step;
    public float RotationSpeed;
    Unity Message | 0 references
    void Start()
    {
        Spawn();
    }
    1 reference
    public void Spawn()
    {
        Clear();
        for (int i = 0; i < Slice.Length; i++)
        {
            GameObject SpawnedSlice = Instantiate(SpritePrefab, transform.position, transform.rotation);
            SpawnedSlice.GetComponent<SpriteRenderer>().sprite = Slice[i];
            SpawnedSlice.GetComponent<SpriteRenderer>().sortingOrder = i;
            SpawnedSlice.transform.parent = gameObject.transform;
        }
        OffsetUpdate();
    }
    1 reference
    public void Clear()
    {
        while (transform.childCount > 0)
```

Рисунок 3.27 – Клас SpriteStack

Даний клас має функції що дають можливість створювати 2D об’єкти з ефектом 3D. Для цього декілька спрайтів накладається один на один з певним відступом, щоб надати об’єкту об’єм. Це працює завдяки функцій Spawn та Clear (рис. 3.28).

```
public void Spawn()
{
    Clear();
    for (int i = 0; i < Slice.Length; i++)
    {
        GameObject SpawnedSlice = Instantiate(SpritePrefab, transform.position, transform.rotation);
        SpawnedSlice.GetComponent<SpriteRenderer>().sprite = Slice[i];
        SpawnedSlice.GetComponent<SpriteRenderer>().sortingOrder = i;
        SpawnedSlice.transform.parent = gameObject.transform;
    }
    OffsetUpdate();
}
1 reference
public void Clear()
{
    while (transform.childCount > 0)
    {
        DestroyImmediate(transform.GetChild(0).gameObject);
    }
}
```

Рисунок 3.28 – Функції Spawn та Clear

Побудова компонентів в Unity

Для створення сітки створюється відповідний компонент Tile, та з нього створюється шаблон (рис. 3.29). Основний компонент для побудови сітки.

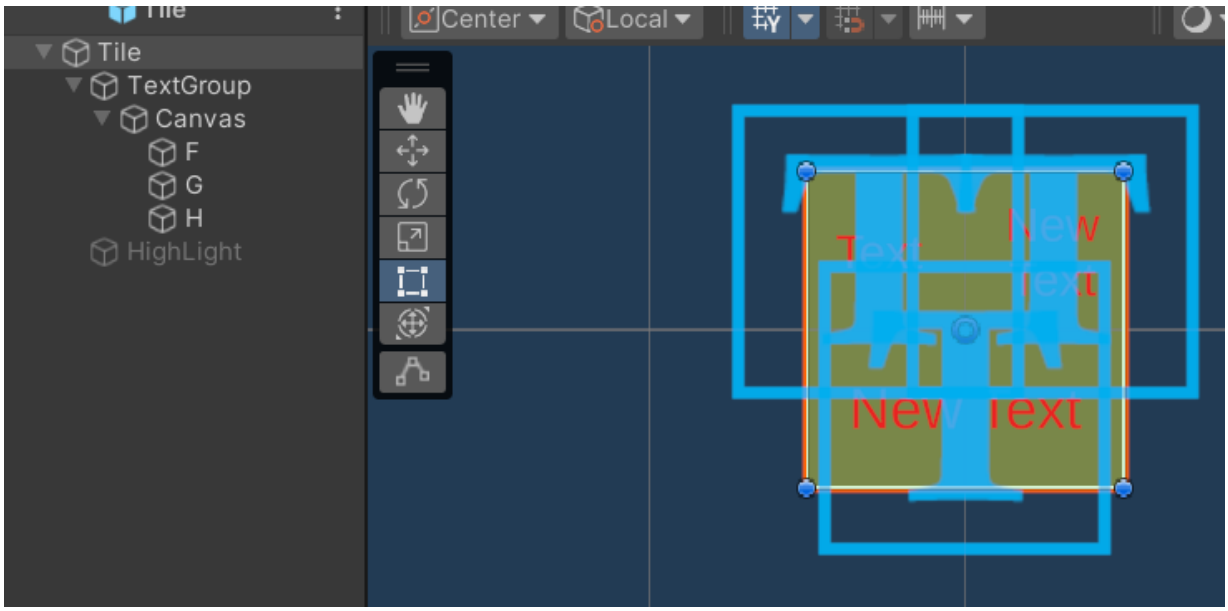


Рисунок 3.29 – Компонент Tile

Компонент має в собі спрайт зафарбований в зелений колір. В компоненті GridManager відповідно встановлюється цей шаблон, та задається кількість на ширину та висоту та ігровий об’єкт. (рис. 3.30):

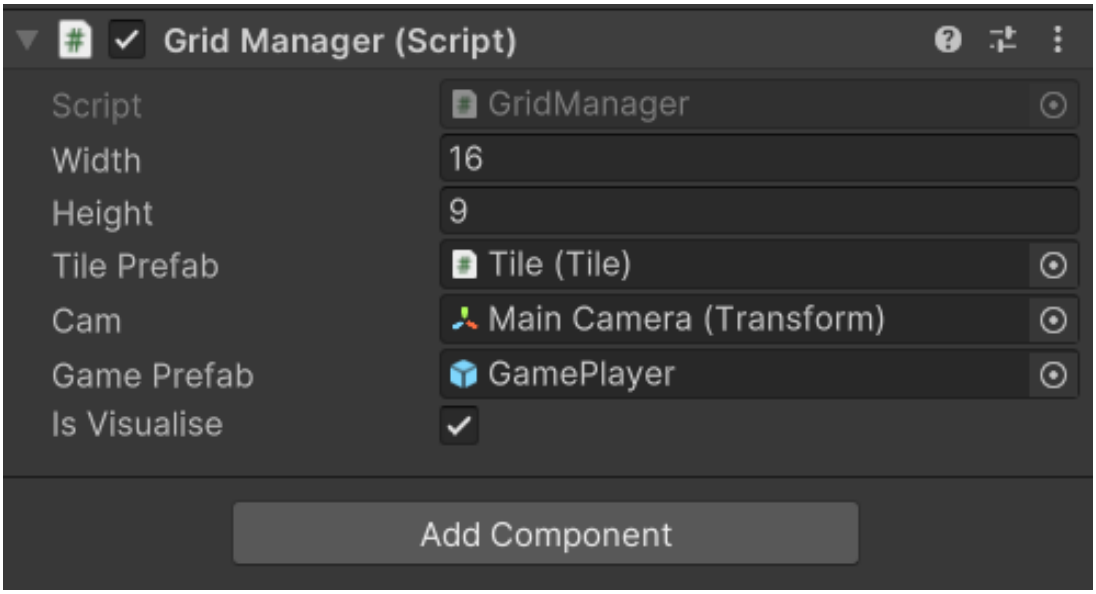


Рис. 3.30 – Встановлені параметри побудови сітки

Далі програма формує сітку на якій означено кінцеву та стартову точку (рис. 3.31).

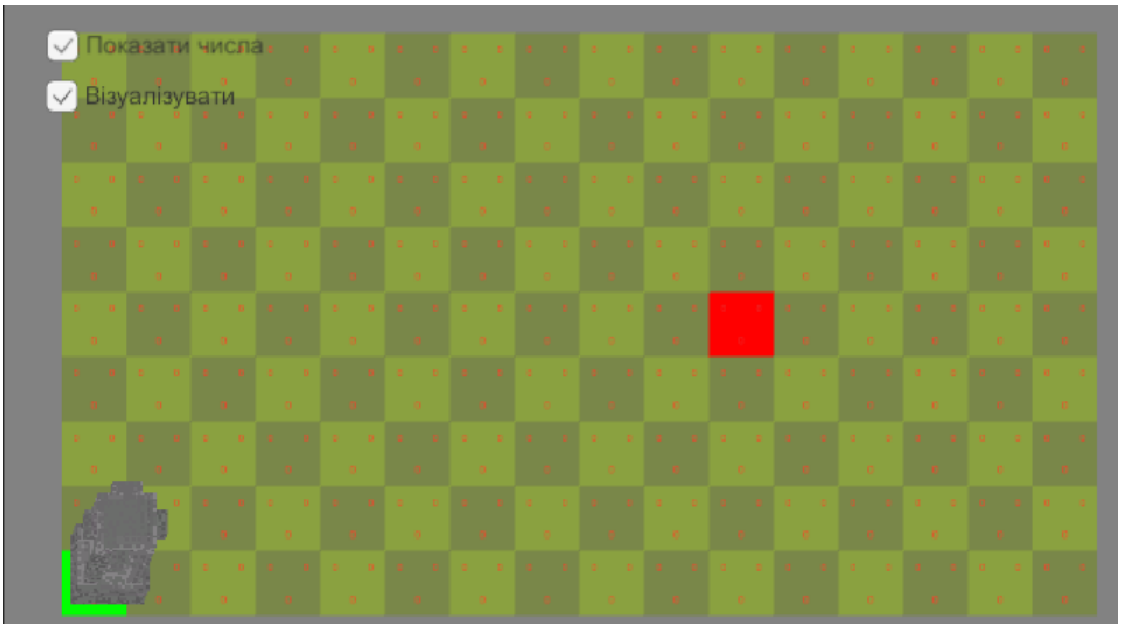


Рисунок 3.31 – Створення сітки вузлів

Відповідно після закінчення роботи алгоритму, зображується повний шлях та з відповідними сусідніми вузлами, що позначені синім кольором (рис. 3.32):

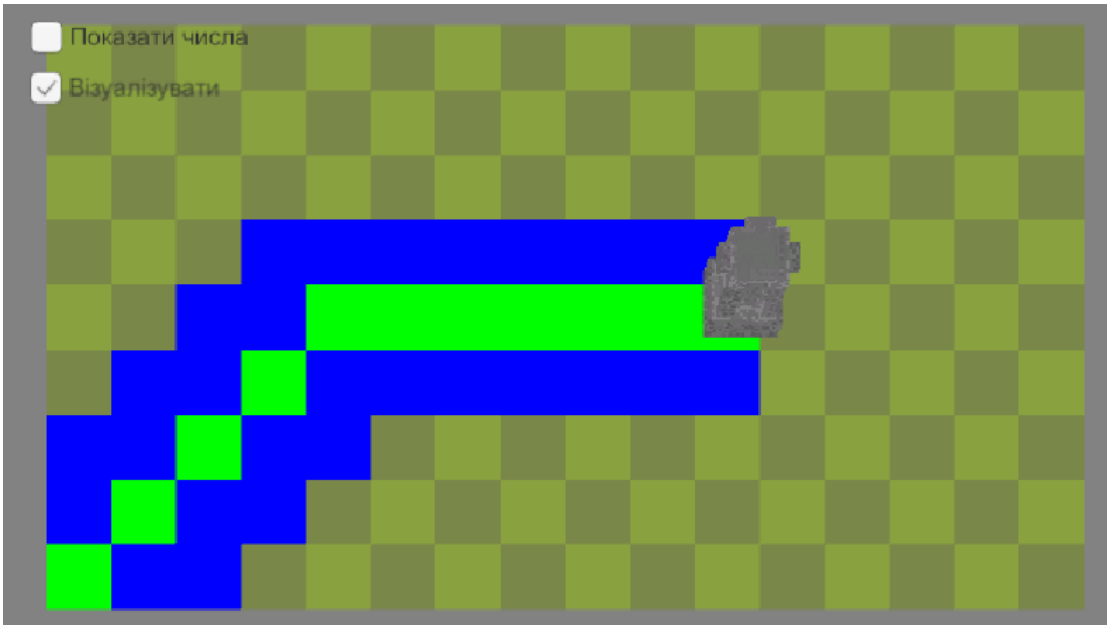


Рисунок 3.32 – Зображення повного шляху

Також можна спостерігати можливість увімкнути показ чисел величини при розрахунку шляху:

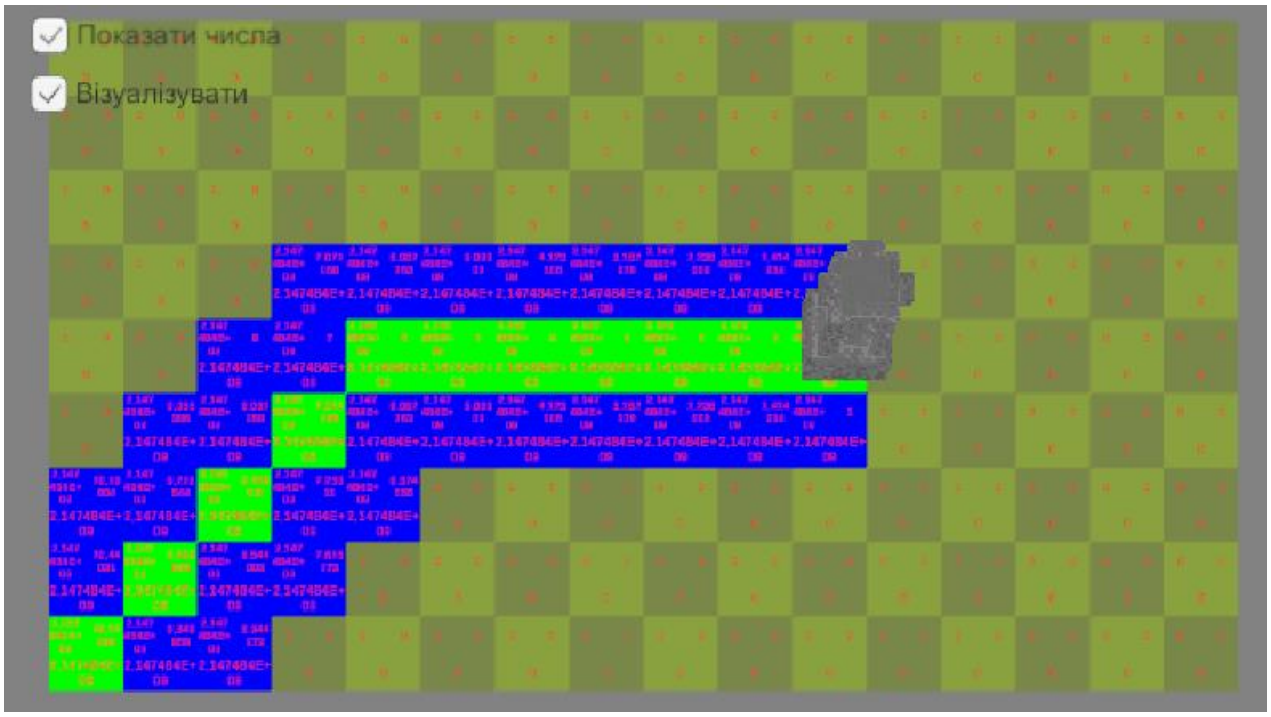


Рисунок 3.34 – Візуальне відображення чисел

Якщо умовно приблизи то можна спостерігати всі величини при яких обчислюється шлях (рис. 3.35).

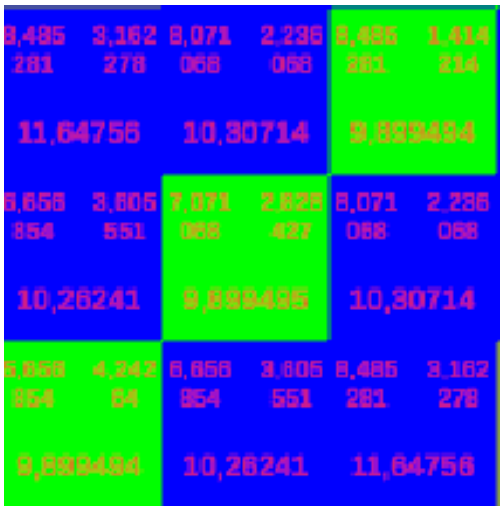


Рисунок 3.35 – Зображення всіх величин вузла

На кожному вузлі зображено перші 2 верхніх числа це g(відстань вузла від поточного) та h (відстань вузла від кінцевого), нижче число f(сума величини g+h).

3.3 Тестування алгоритму пошуку шляху

В загальному випадку алгоритм роботи програми виглядає наступним чином (рис. 3.36):

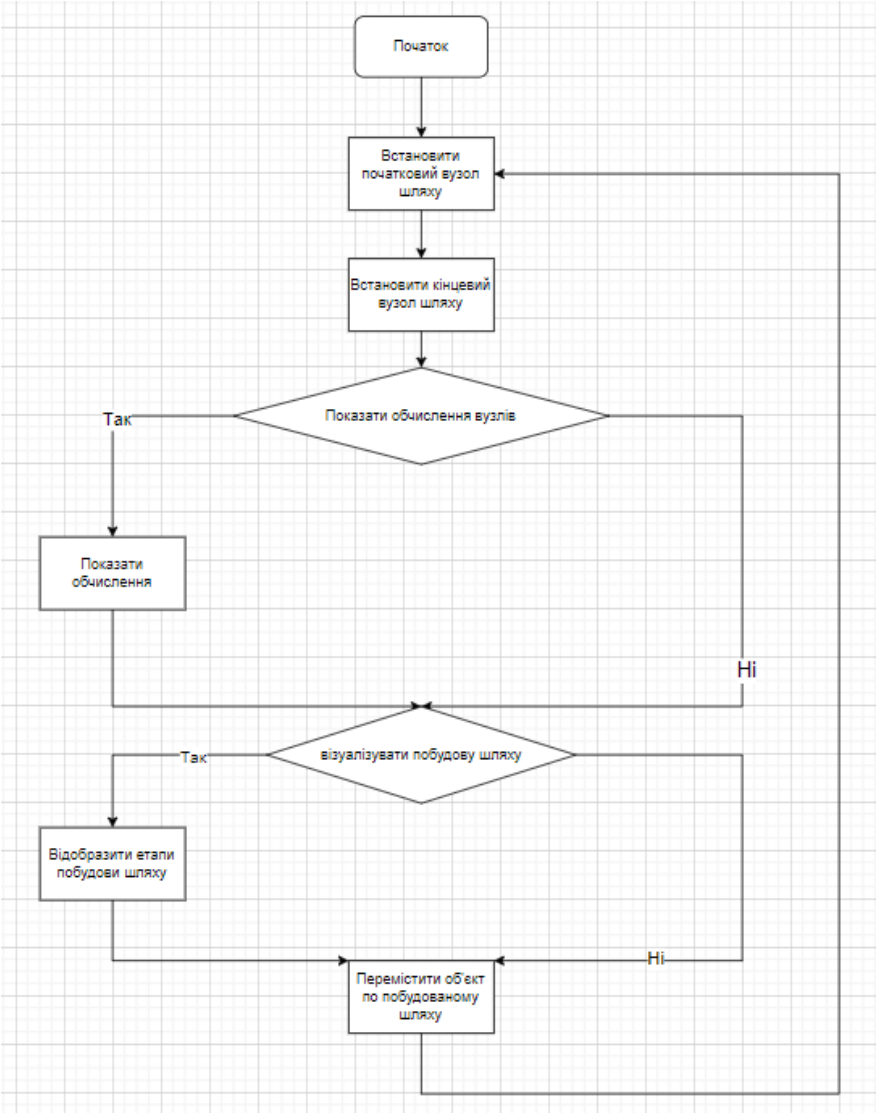


Рисунок 3.36 – Блок-схема роботи програми

Отже для початку роботи програми потрібно її запустити. Під час завантаження програми користувач спостерігатиме згенеровану сітку з якою він може взаємодіяти та налаштування в лівому верхньому куті (рис. 3.37).

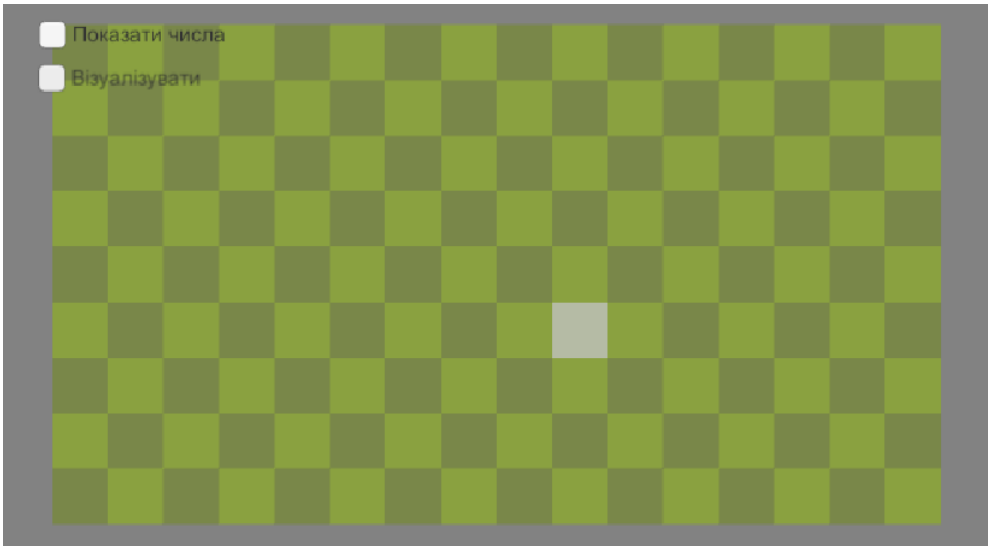


Рисунок 3.37 – Завантаження сітки

Після завантаження програми користувачу потрібно спочатку відмітити початкову точку лівою кнопкою миші. Як тільки користувач виконає ці дії з’явиться об’єкт який буде переміщуватися після того як буде створений шлях (рис. 3.38).

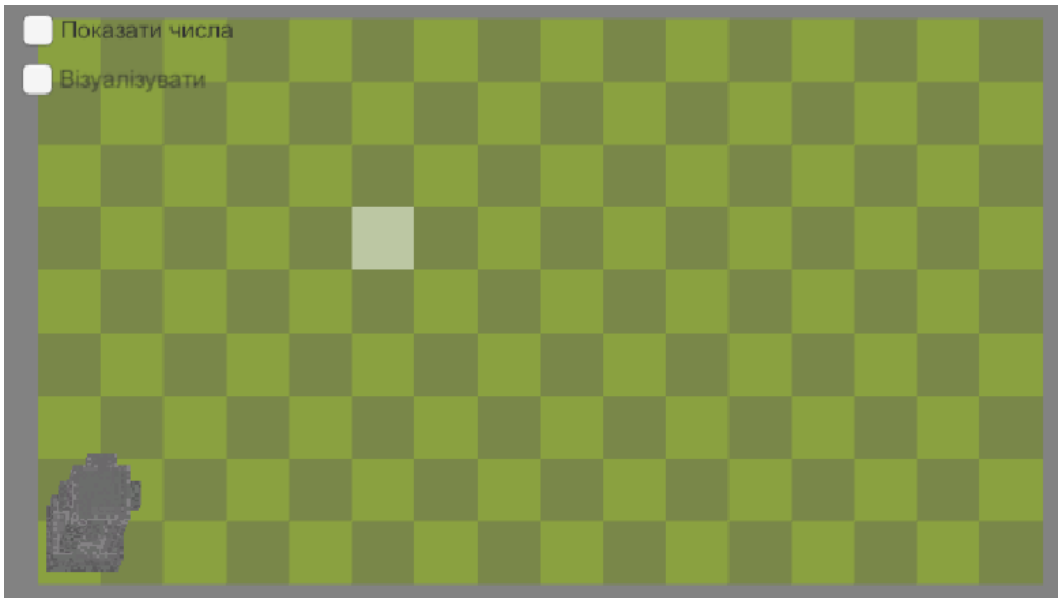


Рисунок 3.38 – Створення стартового вузла

Далі користувач повинен відмітити, правою кнопкою миші кінцевий вузол на який і буде переміщатися об’єкт (рис. 3.39).

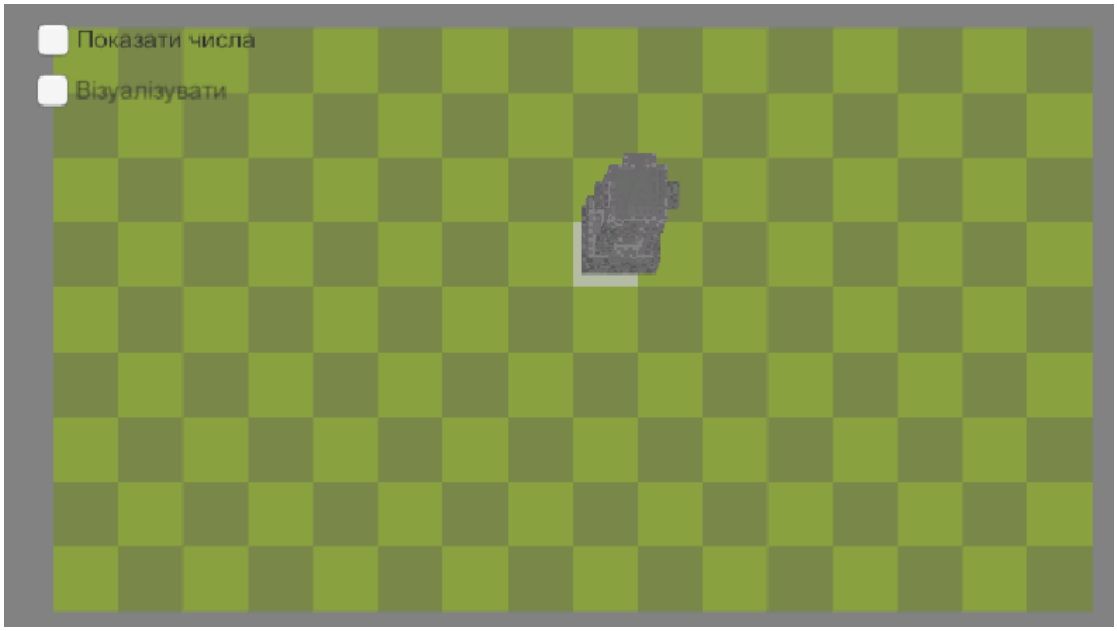


Рисунок 3.39 – Переміщення об’єкту по шляху

Якщо користувач має необхідність отримати дані яким чином обчислювалися значення для знаходження оптимального шляху йому потрібно увімкнути налаштування «Показати числа» та візуалізувати.

Відповідно далі користувач може візуально спостерігати яким чином знаходиться шлях.

Знову відмітимо стартову та кінцеву точку (рис. 3.40).

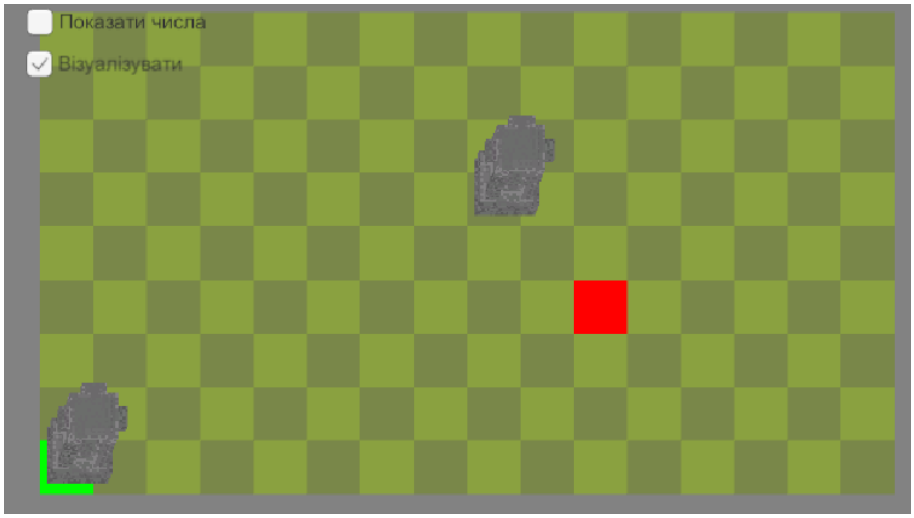


Рисунок 3.40 – Позначення стартової та кінцевої точки

Можна спостерігати (рис. 3.40), що стартова та кінцева точка позначилися зеленим та червоним кольором відповідно, далі можна спостерігати весь шлях після того як об’єкт перемістився (рис. 3.41):

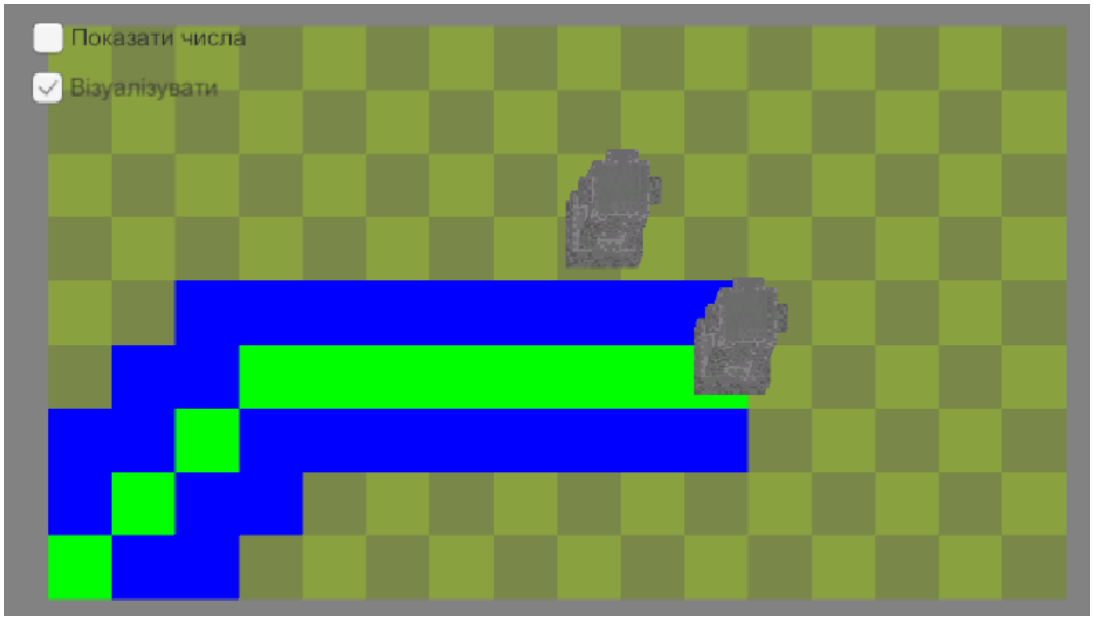


Рисунок 3.41 – Побудова оптимального шляху

Отже шлях на рисунку 3.41, позначений зеленим кольором, а сусіди які розглядалися як можливий варіант позначався синім кольором. Таким чином можна спостерігати як саме та через, які вузли проходив шлях.

Згідно алгоритму для шляху обирається той вузол серед сусідів, що має найменші ваги. Для того щоб впевнитися в цьому користувачу достатньо увімкнути налаштування «Показати числа», і таким чином показуються всі параметри кожного вузла при побудові шляху (рис. 3.42).

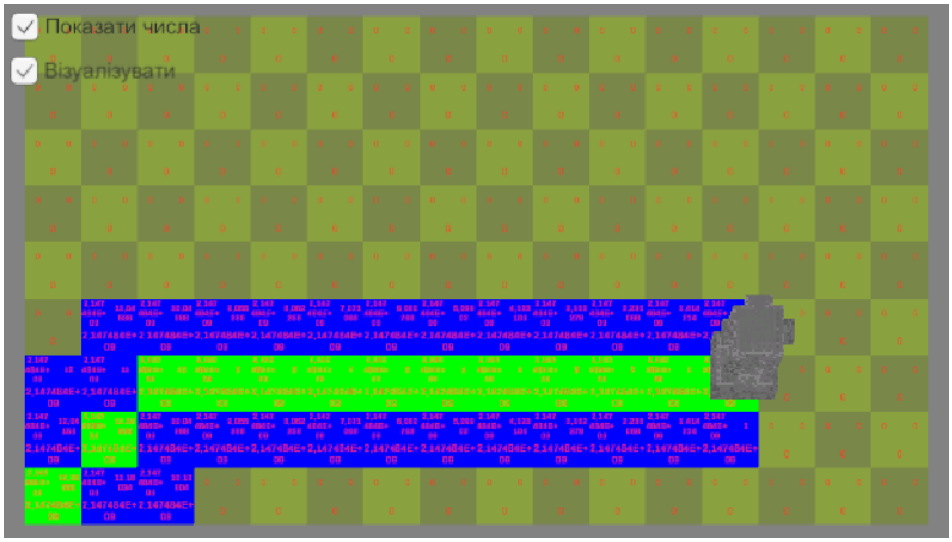


Рисунок 3.42 – Зображення шляху та обчислень

Таким чином користувач може чітко оцінити чому саме через цей вузол проходить шлях завдяки візуалізації параметрів.

Даний проект, повністю реалізує алгоритм оптимального пошуку шляху, але має певні недоліки, які є можливість покращити. Даний алгоритм має доволі не високу швидкість розрахунку, тому серед варіантів покращення можна вдосконалити систему обчислення вузлів додати паралельне виконання функцій. Загалом програма, краще могла застосовувати даний алгоритм при демонстрації реального ігрового процесу, що дало б кращого розуміння доцільності використання даного алгоритму.

3.4 Висновок до третього розділу

Розробка алгоритму A* для пошуку найкоротшого шляху в сітці Unity є важливим завданням в галузі комп'ютерних наук та розробки ігор. В ході розробки було успішно впроваджено ключові компоненти алгоритму, такі як обчислення вартостей gCost, hCost та fCost, визначення та оновлення сусідів для кожного вузла, врахування типів вузлів (початок, кінець, сусід), і візуалізація результатів на сцені Unity.

Алгоритм виявився ефективним у визначенні оптимальних шляхів між початковим та кінцевим вузлами, демонструючи свою актуальність в галузях, де шляхоутворення важливо, таких як ігрова розробка, симуляції та робототехніка. Гнучкість та налаштовуваність алгоритму дозволяють використовувати його для різноманітних завдань, а його реалізація в середовищі Unity надає можливість легко інтегрувати його в реальні проекти.

Результатом розробки є успішна імплементація та використання алгоритму A* у визначенні оптимальних шляхів в графах та сітках, підтверджуючи його значущість у сучасних застосуваннях у галузі комп'ютерних наук.

ВИСНОВОК

Отже, алгоритм A* вирізняється своєю здатністю знаходити оптимальний шлях з високою точністю та ефективністю завдяки використанню евристичного підходу, який дозволяє балансувати між вартістю шляху та евристичною оцінкою відстані до цілі.

Поряд з алгоритмом A*, в кваліфікаційній роботі були розглянуті його модифікації та альтернативні методи пошуку шляху, такі як алгоритми Дейкстри, жадібні алгоритми BFS, IDA*, RBFS, MA*, SMA*, D*-Lite, які мають свої специфічні переваги в залежності від контексту задачі, обмежень пам'яті, необхідності адаптації до динамічних змін у графі та інших факторів. Це підкреслює важливість вибору відповідного алгоритму або його модифікації в залежності від специфіки задачі та вимог до точності, швидкості, використання пам'яті та інших ключових параметрів.

Розробка та використання алгоритму A* в середовищі Unity для пошуку найкоротшого шляху в сітці демонструє його практичне значення в ігровій розробці, дозволяючи розробникам ігор створювати ефективні та оптимізовані системи шляхоутворення. Гнучкість алгоритму, зокрема його налаштовуваність та здатність інтегруватися в різні проекти, робить його цінним інструментом для широкого спектру застосувань.

Враховуючи різноманітність сценаріїв застосування та потенціал алгоритму A* та його модифікацій, можна зробити висновок про важливість глибокого аналізу задачі та обрання найбільш підходящого методу пошуку шляху. Це дозволить досягти оптимального балансу між ефективністю виконання, точністю результатів та ресурсними витратами, що є ключовим для успішної реалізації проектів в комп'ютерних науках, ігровій розробці, симуляціях та робототехніці. Особливо значимим алгоритм A* є для задач, де потрібно знаходити найбільш ефективний шлях в складних та динамічно

змінюваних середовищах, що є типовим для великої кількості сучасних застосувань.

Адаптація та оптимізація алгоритму під конкретні умови задачі може значно покращити продуктивність і ефективність рішення, особливо в умовах обмежених ресурсів або високих вимог до швидкодії. Використання евристичних функцій дозволяє не тільки прискорити пошук, але й забезпечити більш точне та цілеспрямоване досягнення цілей.

Таким чином, алгоритм A* та його варіації представляють собою важливі інструменти в арсеналі розробників програмного забезпечення, ігор, робототехніки та систем симуляції. Вони дозволяють ефективно вирішувати складні задачі пошуку шляху, адаптуючись до різноманітних вимог та обмежень. Ключ до успішного використання цих алгоритмів лежить у ретельному аналізі задачі, правильному виборі та налаштуванні алгоритму, що дозволить досягнути найкращого балансу між продуктивністю, ефективністю та ресурсними витратами, забезпечуючи високу якість та надійність рішень.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Daohong Liu. Research of the Path Finding Algorithm A* in Video Games // Highlights in Science, Engineering and Technology – Volume 39 (2023)

2. Introduction to the A* Algorithm [Електронний ресурс] – Режим доступу: https://www.redblobgames.com/pathfinding/a-star/introduction.html

3. How Pathfinding AI works in Video Games [Електронний ресурс]. – Режим доступу: https://www.gameprogrammingworkshop.com/how-pathfinding-ai-works-in-video-games/

4. Dijkstra's Algorithm - Shortest Path [Електронний ресурс]. – Режим доступу: https://gamedev.net/tutorials/programming/artificial-intelligence/dijkstras-algorithm-shortest-path-r3872/

5. Алгоритм Дейкстри [Електронний ресурс]. – Режим доступу: https://www.wikiwand.com/uk/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B8

6. Алгоритм пошуку A* [Електронний ресурс]. – Режим доступу: https://www.wikiwand.com/uk/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BF%D0%BE%D1%88%D1%83%D0%BA%D1%83_A*

7. Алгоритм Беллмана — Форда [Електронний ресурс]. – Режим доступу: https://www.wikiwand.com/uk/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D0%B5%D0%BB%D0%BB%D0%BC%D0%B0%D0%BD%D0%B0-%D0%A4%D0%BE%D1%80%D0%B4%D0%B0

8. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In Computer Science 1972 4(2):100–107.

9. Документація Unity3D [Електронний ресурс]. – Режим доступу: https://docs.unity.com

10.Документація C# [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/ru-ru/dotnet/csharp/

11.An Introduction to A* and IDA* [Електронний ресурс]. – Режим доступу: https://medium.com/smucs/an-introduction-to-a-and-ida-5f11c039abad

12.D*, D* Lite & LPA* [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/iterative-deepening-a-algorithm

13.A* Search: Concept, Algorithm, Implementation, Advantages, Disadvantages [Електронний ресурс]. – Режим доступу: https://www.brainkart.com/article/A--Search--Concept,-Algorithm,-Implementation,-Advantages,-Disadvantages_8883/

14.Iterative Deepening A* Algorithm (IDA*) [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/iterative-deepening-a-algorithm

ДОДАТОК А

```
using System.Collections;
using System.Collections.Generic;
using System.Runtime.InteropServices.WindowsRuntime;
using Unity.VisualScripting;
using UnityEditor.Experimental.GraphView;
using UnityEngine;
using static TreeEditor.TreeEditorHelper;

public class GridManager : MonoBehaviour
{
    [SerializeField] private int _width, _height;

    [SerializeField] private Tile _tilePrefab;
    [SerializeField] private Transform _cam;
    public GameObject _gamePrefab;

    private Tile endGrid;
    private Tile startGrid;
    private List<Tile> _tiles;
    //
    private GameObject go;
    private PlayerMove script;
    public bool IsVisualise = false;
    private void Start()
    {
        _tiles = new List<Tile>();
        GenerateGrid();
    }
```

```
    public void VisualSetting(bool visual)
    {
        IsVisualise = visual;
    }
    void GenerateGrid()
    {
        for(int x = 0; x < _width; x++)
        {
            for(int  y = 0; y < _height; y++)
            {
                var tileSpawn = Instantiate(_tilePrefab, new Vector2(x,y), Quaternion.identity);
                tileSpawn.name = $"Tile {x} {y}";
                var isOffSet = (x % 2 == 0 && y % 2 != 0) || (x % 2 != 0 && y % 2 == 0);
                _tilePrefab.Init(isOffSet);
                _tiles.Add(tileSpawn);
            }
        }
        _cam.transform.position = new Vector3((float)_width / 2 - 0.5f, (float)_height / 2 - 0.5f, -10);
    }
    public void ResetGrids()
    {
        foreach(Tile tile in _tiles)
        {
            float x = tile.GetPosition().x;
            float y = tile.GetPosition().y;
            var isOffSet = (x % 2 == 0 && y % 2 != 0) || (x % 2 != 0 && y % 2 == 0);
            tile.Init(isOffSet);
```

```
    }
}
public void CreateStartGrid(Vector3 position)
{
    ResetGrids();
    startGrid = GetTile(position.x, position.y);
    startGrid.type = NodeType.Start;
    go = Instantiate(_gamePrefab, position, Quaternion.identity);
    script = go.GetComponent<PlayerMove>();
    UpdateVisualization(startGrid);
}
public void CreateEndGrid(Vector3 position)
{
    endGrid = GetTile(position.x,position.y);
    endGrid.type = NodeType.End;
    UpdateVisualization(endGrid);
}
private void Update()
{
    if(Input.GetKey(KeyCode.Space))
    {
       script.Move( FindShortestPath(startGrid,endGrid));
    }
}
public void UpdateVisualization(Tile nodeTile)
{
    if(!IsVisualise)
    {
        return;
    }
```

```
    //Renderer renderer = GetComponent<Renderer>();
    NodeType nodeType = nodeTile.type;
    switch (nodeType)
    {
        case NodeType.Start:
            nodeTile._spriteRenderer.color = Color.green;
            break;

        case NodeType.End:
            nodeTile._spriteRenderer.color = Color.red;
            break;
        case NodeType.Neighbor:
            nodeTile._spriteRenderer.color = Color.blue;
            break;
        case NodeType.path:
            nodeTile._spriteRenderer.color = Color.green;
            break;
    }
}
public List<Tile> FindShortestPath(Tile startNode, Tile endNode)
{
    // Ініціалізація списку відкритих вузлів
    List<Tile> openSet = new List<Tile>();
    // Ініціалізація списку вже розглянутих вузлів
    List<Tile> closedSet = new List<Tile>();
    foreach(Tile tile in _tiles)
    {
        tile.gCost = int.MaxValue;
        tile.GetFCost();
```

```
        }
        // Додавання початкового вузла до списку відкритих
        openSet.Add(startNode);
        while (openSet.Count > 0)
        {
            // Обираємо вузол з найменшою вартістю fCost
            Tile currentNode = openSet[0];

            // Шукаємо вузол з меншою вартістю fCost у відкритому списку
            for (int i = 1; i < openSet.Count; i++)
            {
                if (openSet[i].GetFCost() < currentNode.GetFCost() ||
                    (openSet[i].GetFCost() == currentNode.GetFCost() &&
openSet[i].hCost < currentNode.hCost))
                {
                    currentNode = openSet[i];
                }
            }
            // Видаляємо поточний вузол з відкритого списку і додаємо його в
закритий список
            openSet.Remove(currentNode);
            closedSet.Add(currentNode);
            foreach(Tile tile in closedSet)
            {
                tile.type = NodeType.path;
                UpdateVisualization(tile);
            }
            // Якщо поточний вузол - кінцевий, то ми знайшли шлях
            if (currentNode == endNode)
            {
```

```
            return closedSet;
        }
        foreach (Tile neighbor in GetNeighborList(currentNode))
        {
            bool isCont = openSet.Contains(neighbor);



            // Обчислюємо нові вартості gCost та hCost
            float newGCost = currentNode.gCost + GetDistance(currentNode, neighbor);

            if (newGCost < neighbor.gCost || !isCont)
            {

                // Додаємо сусіда до відкритого списку, якщо його там немає
                if (!isCont)
                    // Оновлюємо вартості та попередника вузла
                    neighbor.UpdateCosts(newGCost, GetDistance(neighbor, endNode));
                    openSet.Add(neighbor);
                }
            }
        }
        return null;
    }
    public List<Tile> GetNeighborList(Tile currentNode)
    {
        List<Tile> neighborList = new List<Tile>();

        if (currentNode.GetPosition().x - 1 >= 0)
```

```
        {
            neighborList.Add(GetTile(currentNode.GetPosition().x - 1,
currentNode.GetPosition().y)); //Left
            if (currentNode.GetPosition().y - 1 >= 0)
neighborList.Add(GetTile(currentNode.GetPosition().x - 1,
currentNode.GetPosition().y - 1)); //Left Down
            if (currentNode.GetPosition().y + 1 < _height)
neighborList.Add(GetTile(currentNode.GetPosition().x - 1,
currentNode.GetPosition().y + 1)); //Left Up
        }
        if (currentNode.GetPosition().x + 1 < _width)
        {
            neighborList.Add(GetTile(currentNode.GetPosition().x + 1,
currentNode.GetPosition().y)); //Right
            if (currentNode.GetPosition().y - 1 >= 0)
neighborList.Add(GetTile(currentNode.GetPosition().x + 1,
currentNode.GetPosition().y - 1)); //Right Down
            if (currentNode.GetPosition().y + 1 < _height)
neighborList.Add(GetTile(currentNode.GetPosition().x + 1,
currentNode.GetPosition().y + 1)); //Right Up
        }
        if (currentNode.GetPosition().y - 1 >= 0)
neighborList.Add(GetTile(currentNode.GetPosition().x,
currentNode.GetPosition().y - 1)); //Down
        if (currentNode.GetPosition().y + 1 <_height)
neighborList.Add(GetTile(currentNode.GetPosition().x,
currentNode.GetPosition().y + 1)); //Up

        foreach (Tile neighbor in neighborList) {
            neighbor.type = NodeType.Neighbor;
```

```
            UpdateVisualization(neighbor);
        }
        return neighborList;
    }
    public Tile GetTile(float x, float y)
    {
        Vector3 pos = new Vector3(x, y, 0);
        foreach(Tile tile in _tiles)
        {
            if(tile.GetPosition() == pos)
            {
                return tile;
            }

        }
        return null;
    }
    public float GetDistance(Tile nodeA, Tile nodeB)
    {
        float dstX = Mathf.Abs(nodeA.GetPosition().x - nodeB.GetPosition().x);
        float dstY = Mathf.Abs(nodeA.GetPosition().y - nodeB.GetPosition().y);
        return Mathf.Sqrt(dstX * dstX + dstY * dstY);
    }
    public void DeactivateText(bool isActive)
    {
        foreach(Tile tile in _tiles)
        {
            tile.groupText.SetActive(isActive);
        }
    }
```

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


public enum NodeType
{
    Start,
    End,
    Neighbor,
    path
}
public class Tile : MonoBehaviour
{
    //Текст
    public TextMeshProUGUI G;
    public TextMeshProUGUI H;
    public TextMeshProUGUI F;
    public GameObject groupText;
    // Start is called before the first frame update
    [SerializeField] private Color basezolor, _offSetColor;
    public GameObject _highLight;
    public SpriteRenderer _spriteRenderer;
    public NodeType type;
    public float gCost;
    public float hCost;
    public float fCost;
    private GridManager gridManager;
```

```
    private void Start()
    {
        gridManager =
GameObject.Find("GridManager").GetComponent<GridManager>();
        G.text = gCost.ToString();
        H.text = hCost.ToString();
        F.text = fCost.ToString();
    }
    public void Init(bool isOffset)
    {
        _spriteRenderer.color = isOffset ? _offSetColor : basezolor;
    }
    public Vector3 GetPosition()
    {
        return this.transform.position;
    }
    public float GetFCost()
    {
        fCost = gCost + hCost;
        return fCost;
    }
    public void UpdateCosts(float newGCost, float newHCost)
    {
        gCost = newGCost;
        hCost = newHCost;
        fCost = gCost + hCost;
        UpdateText(gCost, hCost, fCost);
    }
    public void UpdateText(float g, float h, float f)
```

```
    {
        G.text = gCost.ToString();
        H.text = hCost.ToString();
        F.text = fCost.ToString();
    }
    private void OnMouseEnter()
    {
        _highLight.SetActive(true);
    }
    private void OnMouseOver()
    {
        if (Input.GetMouseButtonDown(0))
        {
            gridManager.CreateStartGrid(GetPosition());
        }
        if(Input.GetMouseButtonDown(1))
        {
            gridManager.CreateEndGrid(GetPosition());
        }
    }
    private void OnMouseExit()
    {
        _highLight.SetActive(false);
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SpriteStack : MonoBehaviour
```

```
{
    // Start is called before the first frame update
    public Sprite[] Slice;
    public GameObject SpritePrefab;
    public Vector2 Offset;
    public Vector2 Step;
    public float RotationSpeed;
    void Start()
    {
        Spawn();
    }
    public void Spawn()
    {
        Clear();
        for (int i = 0; i < Slice.Length; i++)
        {
            GameObject SpawnedSlice = Instantiate(SpritePrefab, transform.position, transform.rotation);
            SpawnedSlice.GetComponent<SpriteRenderer>().sprite = Slice[i];
            SpawnedSlice.GetComponent<SpriteRenderer>().sortingOrder = i;
            SpawnedSlice.transform.parent = gameObject.transform;
        }
        OffsetUpdate();
    }
    public void Clear()
    {
        while (transform.childCount > 0)
        {
            DestroyImmediate(transform.GetChild(0).gameObject);
        }
```

```
    }
    public void OffsetUpdate()
    {
        float oof = Offset.y;
        float oox = Offset.x;
        for (int i = 0; i < transform.childCount; i++)
        {
            transform.GetChild(i).localPosition = new Vector2(oox, oof);
            oof += Step.y;
            oox += Step.x;
        }
    }
    // Update is called once per frame
    void Update()
    {
        RotationUpdate();
    }
    public void RotationUpdate()
    {
        for (int i = 0; i < transform.childCount; i++)
        {
            transform.GetChild(i).Rotate(0, 0, RotationSpeed * Time.deltaTime);
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMove : MonoBehaviour
```

```
{
    // Start is called before the first frame update
    public void Move(List <Tile> tiles)
    {
        foreach (Tile tile in tiles)
        {
            transform.position =  tile.GetPosition();
        }
    }
}
```