КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота

на здобуття освітнього рівня бакалавра
за спеціальністю 121 Програмна інженерія
на тему:

АЛГОРИТМИ ПОШУКУ ШЛЯХУ У ВІДЕОІГРАХ

Виконав: студент 4-го курсу
Владислав ПРОКОПЧУК

(підпис)

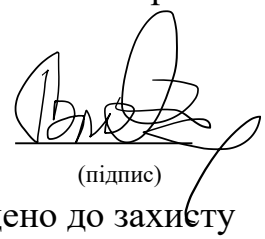
Науковий керівник:
асистент, доктор філософії
з прикладної математики
Віктор СТОВБА

_____

(підпис)

Засвідчую, що в цій курсовій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент



(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«25» травня 2022 р.,
протокол № 10

Завідувач кафедри
Олександр ПРОВОТАР

_____

(підпис)

КИЇВ-2022

РЕФЕРАТ

Обсяг роботи 34 сторінки, 26 ілюстрацій, 25 використаних джерел.

Ключові слова: АЛГОРИТМ, ВІДЕОГРА, НАВЧАЛЬНИЙ ПРОЄКТ, НАЙКОРОТШИЙ ШЛЯХ, ОПТИМАЛЬНИЙ ШЛЯХ, ПОШУК ШЛЯХУ, ШТУЧНИЙ ІНТЕЛЕКТ, A*, ACO, D* LITE, GENETIC ALGORITHM.

Метою кваліфікаційної роботи є ознайомлення з основними алгоритмів погуку шляху, які застосовуються при розробці відеоігор, їхніми перевагами та недоліками. Ознайомлення з основними типами представлення карт ігрового середовища. Використання отриманих знань при створенні навчального проєкту.

Інструментом створення є ігровий двигун Unity 2021.3.3f1 та інтегроване середовище розробки програмного забезпечення Microsoft Visual Studio 2022. Мова програмування C# 9.0.

Результат роботи: досліджено основні алгоритми пошуку шляху, які використовуються при розробці відеоігор. Вивчено основні типи представлення карт ігрового світу та проблеми які виникають при цьому. Отримані теоретичні знання використані для розробки навчального проєкту. Реалізовано алгоритм пошуку шляху A* з оптимізацією на основі бінарної купи.

Розроблений проєкт може застосовуватися в навчальному процесі з вивчення основних принципів роботи алгоритмів пошуку шляху у відеоіграх та в якості бази при розробці власних ігор.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

A* – відомий алгоритм пошуку шляху;

ACO – Ant Colony Optimization, оптимізація мурашиної колонії;

AI – Artificial intelligence, штучний інтелект;

BFS – Best-First-Search, пошук за першим найкращим збігом;

D* – Dynamic A*, динамічний A*;

DFS – Depth-First Search, пошук в глибину;

GA – Genetic Algorithm, генетичний алгоритм;

IDA* – Iterative deepening A*, одна з модифікацій A*;

IDE – Integrated Development Environment, інтегроване середовище розробки;

LPA* – Lifelong Planning A*, планування на все життя A*;

LTS – Long Term Support, довгострокова підтримка;

NPC – Non-Player Character, неігровий персонаж;

RTP – Real Time Pathfinding, пошук в реальному часі.

ВСТУП

Оцінка сучасного стану об’єкта розробки. Індустрія відеоігор є однією з наймолодших на сучасному ринку. З кожним роком сфера відеоігор зацікавлює все більше нових людей різних поколінь, збільшуючи свої прибутки та розширюючи свої горизонти. Ігри ніколи не були тільки для дітей, незважаючи на стереотипи. Спершу взагалі це були науковці й ентузіасти, які вивчали нові можливості комп’ютерів, та інколи відволікались на ігри. Кожного року проводяться турніри з різних кіберспортивних дисциплін, які збирають мільйони глядачів по всьому світу. З таким стрімким розвитком сфери відеоігор абсолютно точно стаєш впевненим у її майбутньому та можна лише допомогти їй у цьому.

Актуальність роботи та підстави для її виконання. Сьогодні поруч із розвитком штучного інтелекту можна спостерігати появу сучасних високих технологій, які потребують створення нового та модифікації старого програмного забезпечення. Наразі активно досліджуються робототехніка, GPS-навігація, космічна галузь та багато інших сфер, в яких можливе застосування алгоритмів пошуку шляху. Проте неможливо розвивати автономно один науковий напрям, він обов’язково буде пов'язаний з іншим. Тому дослідження алгоритмів пошуку шляху важливе не лише для відеоігор.

Мета і завдання роботи. Метою кваліфікаційної роботи є ознайомлення з темою алгоритмів пошуку шляху та їх використанням у відеоіграх. Отримання нових знань у сфері створення відеоігор та їх використання при розробці власного навчального проєкту.

Можливі сфери застосування. Кінцевий продукт може використовуватись як навчальний матеріал при дослідженні теми алгоритмів пошуку шляху та в якості допоміжного матеріалу при розробці власних проєктів.

РОЗДІЛ 1. АЛГОРИТМИ ПОШУКУ ШЛЯХУ

1.1 Що таке пошук шляху

Пошук шляху (англ. Pathfinding) є відомою проблемою, яка полягає в знаходженні оптимального шляху між двома локаціями. На перший погляд, здається, що це просте завдання, проте, коли ми додамо такі речі, як перешкоди разом із іншими параметрами чи змінними, наприклад, складність та час проходження через місцевість, то стає очевидним, що задача є складнішою, ніж просто провести пряму через дві точки.

В контексті відеоігор, оптимальний шлях між двома будь-якими локаціями – це шлях найменшої вартості, а не найкоротший шлях за відстанню. Можна пояснити це тим, що шлях уздовж дороги є кращим з точки зору часу, за шлях через пагорби, водойми або інші перешкоди, оскільки їх подолання займає більше часу, ніж подорож через рівну дорогу. Однак сам по собі ігровий світ не містить такої інформації (рисунок 1.1а). Загальне рішення полягає в тому, щоб накласти на карту іншу, із інформацією про вартість переміщення у ігровому світі, як показано на рисунку 1.2б, й адаптувати алгоритм пошуку, для знаходження шляху з найменшою вартістю.

Сучасні рішення для пошуку шляху у відеоіграх або забезпечують високу швидкість пошуку, жертвуючи точністю, або створюють оптимальний шлях, але з використанням більшої кількості часу та ресурсів [1]. Як знайти оптимальний шлях більш ефективно, до цих пір залишається предметом для вивчення.

а) *б)*

Рисунок 1.1 – Частина міста Stormwind у грі Word of WarCraft: а – без навігаційної сітки; б – із навігаційною сіткою

1.2 Штучний інтелект та його зв’язок з проблемою пошуку шляху

Штучний інтелект (ШІ, англ. Artificial intelligence, AI) – це область комп’ютерних наук, яка ставить за ціль розвиток людино-подібного інтелекту у комп’ютера. Основною ціллю є можливість змоделювати імітацію інтелектуальної поведінки людини та його здатність мислити. Наука про AI розробляє теорії і методології, які дозволяють оцінювати навколишнє середовище і реагувати на різноманітні ситуації так, як би на них реагувала людина.

У відеоіграх AI вирішує багато проблем, які можуть виникнути під час виконання гри. Однією з таких проблем є проблема пошуку шляху. Наступний рисунок 1.2 ілюструє всі ті проблеми, які вирішує AI, а також місце проблеми пошуку шляху в ньому.

Модель AI розділяється на три частини: прийняття рішень, переміщення і стратегія. Прийняття рішень відповідає за те, що персонаж буде робити наступним. Переміщення – як персонаж рухається в просторі. Стратегія відповідає за групову стратегію, яка необхідна для координації певної групи або команди. Переміщення і прийняття рішень включають в

себе алгоритми, які працюють як для ігрових, так і для неігрових персонажів (англ. Non-Player Character, NPC). Пошук шляху є частиною переміщення, за винятком кінематичного руху та поведінки рульового керування. Для вирішення проблеми пошуку шляху були розроблені загальні алгоритми пошуку шляху, такі як: Дейкстри, A*, генетичний алгоритм та оптимізація колонії мурах. Існують два типи пошуку шляху для NPC в іграх. Перший – це статичний пошук шляху, де ціль разом з персонажем не рухаються. Другий – це динамічний пошук шляху або пошук шляху в реальному часі, де як і ціль, так і персонаж, можуть рухатись вільно та випадково.

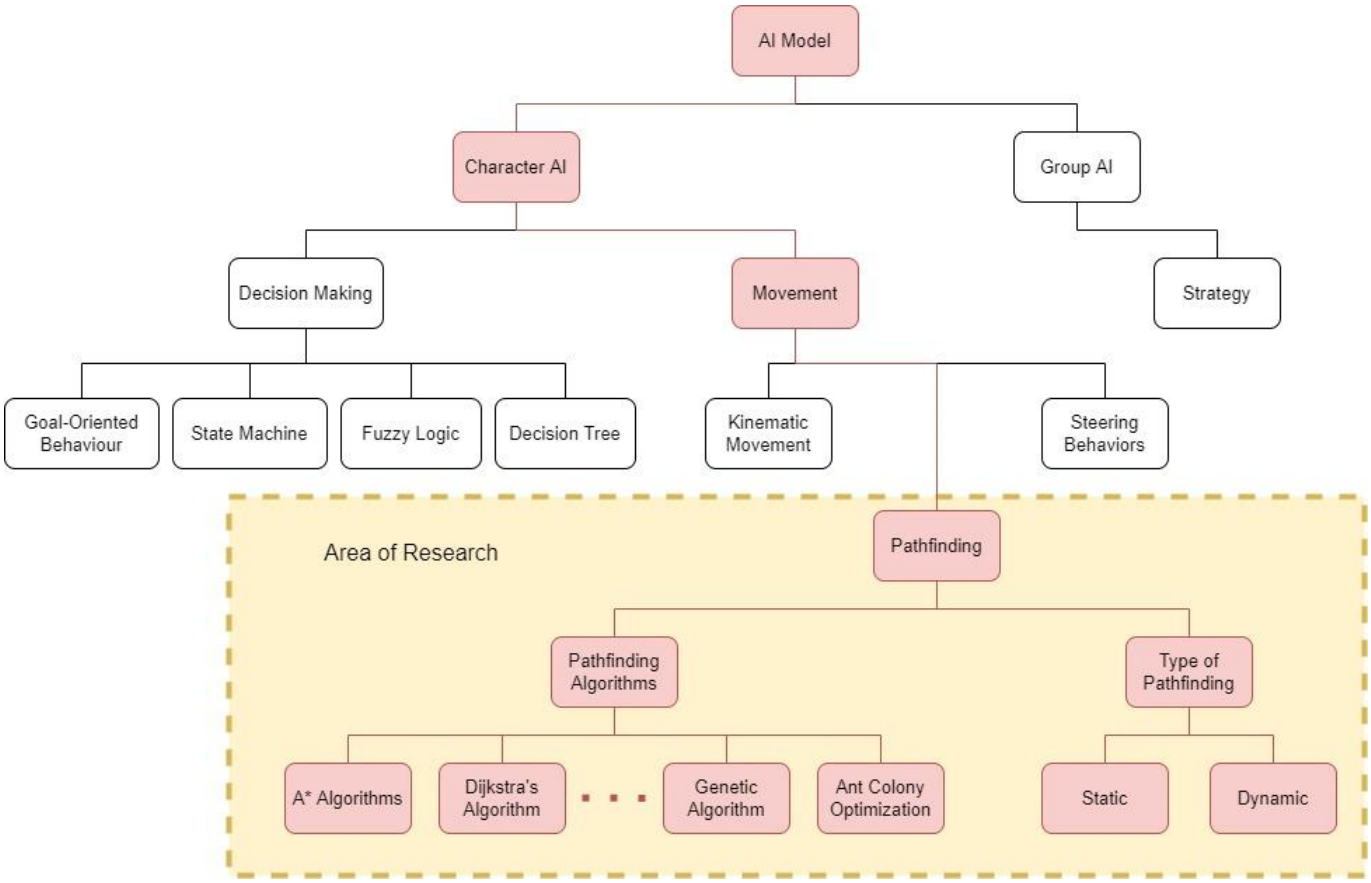


Рисунок 1.2 – Область дослідження

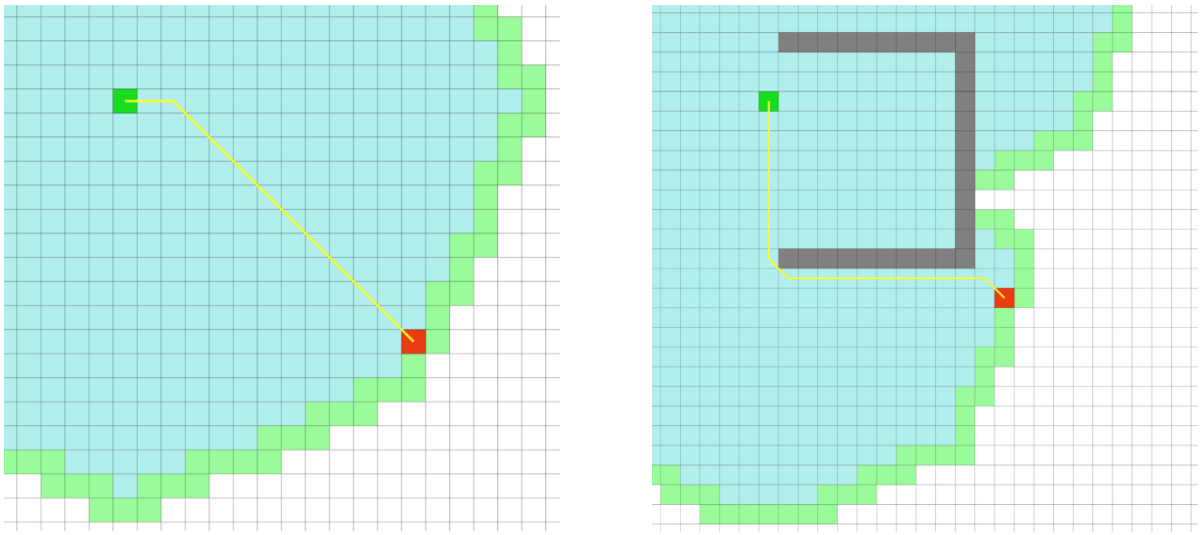
1.3 Алгоритми Дейкстри та жадібний BFS

Голландський вчений у галузі комп’ютерних наук Едсгер Вібе Дейкстра опублікував у 1959 році статтю, де описав названий на його честь алгоритм [2]. Алгоритм належить до класу жадібних знаходить найкоротші шляхи від одного вузла графу до всіх інших, проте може бути використаний і для знаходження найкоротшого шляху між двома вузлами. Суть алгоритму наступна [3]:

- Стартуємо з початкового вузла й аналізуємо граф для знаходження найкоротших шляхів до всіх інших вузлів;
- Зберігаємо всі на разі відомі найкоротші відстані від кожного вузла до початкового та оновлюємо дані, при знаходженні коротшого;
- Позначаємо вузол "відвіданим", якщо знайдено найкоротший шлях до нього;
- Коли всі вузли позначені як "відвідані", то алгоритм завершується і на виході маємо найкоротші шляхи до всіх вузлів.

Алгоритм Дейкстри гарантовано знаходить найкоротший шлях (за винятком, коли вартість шляху є від’ємною), проте, як можна помітити на рисунку 1.3, він є досить повільним й, за великої кількості вузлів, кількість операцій значно зростає. На цих і наступних зображеннях, результати виконання алгоритмів зроблені завдяки [4], де:

- Зелений квадрат – це початковий вузол, а червоний – кінцевий;
- Жовта лінія – знайдений шлях;
- Сірі квадрати – перешкоди;
- Блакитні квадрати – відвідані вузли, а світло-зелені квадрати – відкриті вузли.



а) *б)*

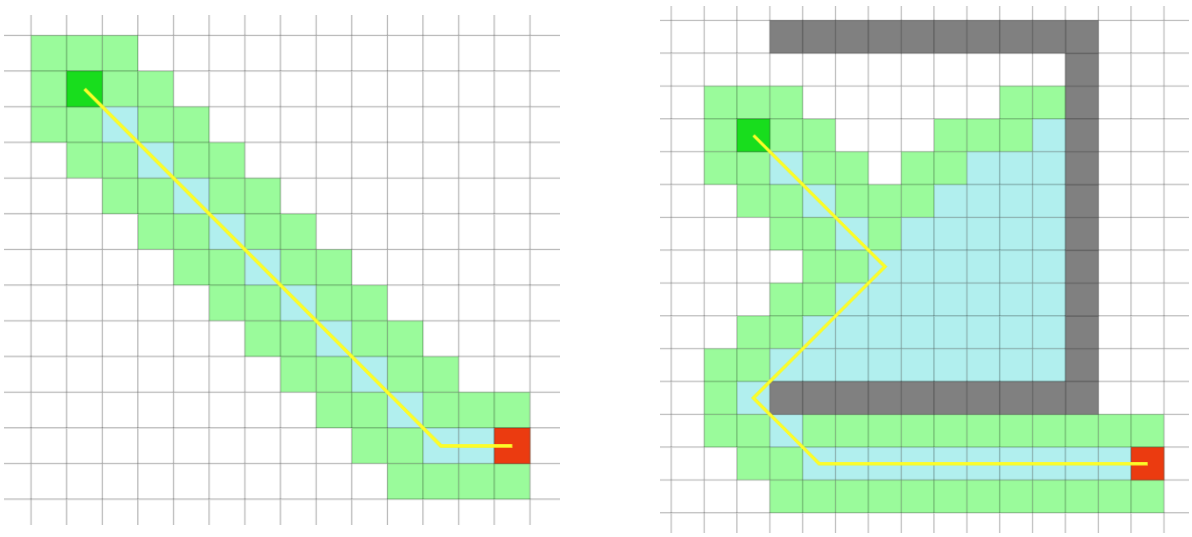
Рисунок 1.3 – Результат виконання алгоритму Дейкстри: а – без перешкод; б – із перешкодами

Жадібний Best-First-Search (BFS) [5] алгоритм працює схожим чином до алгоритму Дейкстри, за винятком того, що він має деяку оцінку, евристику, яка показує наскільки далеко поточний вузол знаходиться від кінцевого. Замість того, щоб обирати найближчий вузол до початкового, він обирає вузол найближчий до кінцевого. Даний алгоритм не гарантує, що знайде найкоротший шлях, проте, він виконується значно швидше алгоритму Дейкстри, завдяки використанню евристики. На рисунку 1.4 можна побачити результат роботи Жадібного BFS алгоритму, де помічаємо, що даний алгоритм дуже добре працює, коли нема перешкод, проте, за їх наявності, шлях може бути далеко не найкоротшим.

1.4 Алгоритм A* та його модифікації

Алгоритм A* (вимовляється як "A зірочка", англ. "A star") – популярний комп'ютерний алгоритм, який широко використовується для пошуку шляху та обходу графів [6]. Він поєднує в собі найкраще від двох попередніх алгоритмів: від алгоритму Дейкстри – гарантує знаходження

найкоротшого шляху, а від Жадібного BFS – застосування евристики для збільшення продуктивності й зменшення кількості операцій.



а) *б)*

Рисунок 1.4 – Результат виконання Жадібного BFS: а – без перешкод; б – із перешкодами

Головною перевагою A* над попередніми алгоритмами є використання функції $f(n)$ для кожного вузла, при виборі наступного підходящого кандидата до оптимального шляху:

$$f(n) = g(n) + h(n),$$

де n – вузол, що розглядається;

$g(n)$ – точна вартість шляху від початкового вузла до вузла n;

$h(n)$ – евристика, оціночна вартість шляху від n до цільового вузла.

Найцікавішою частиною даної функції $f(n)$ є функція евристики $h(n)$, адже функція $g(n)$ завжди має точну вартість і обчислюється досить просто, використовуючи значення з попередніх кроків, проте евристику можна обчислювати різними способами. Дуже важливо мати хорошу

функцію евристики, бо в залежності від того, як ми її обчислюємо вона може значно змінити показники швидкості й точності алгоритму.

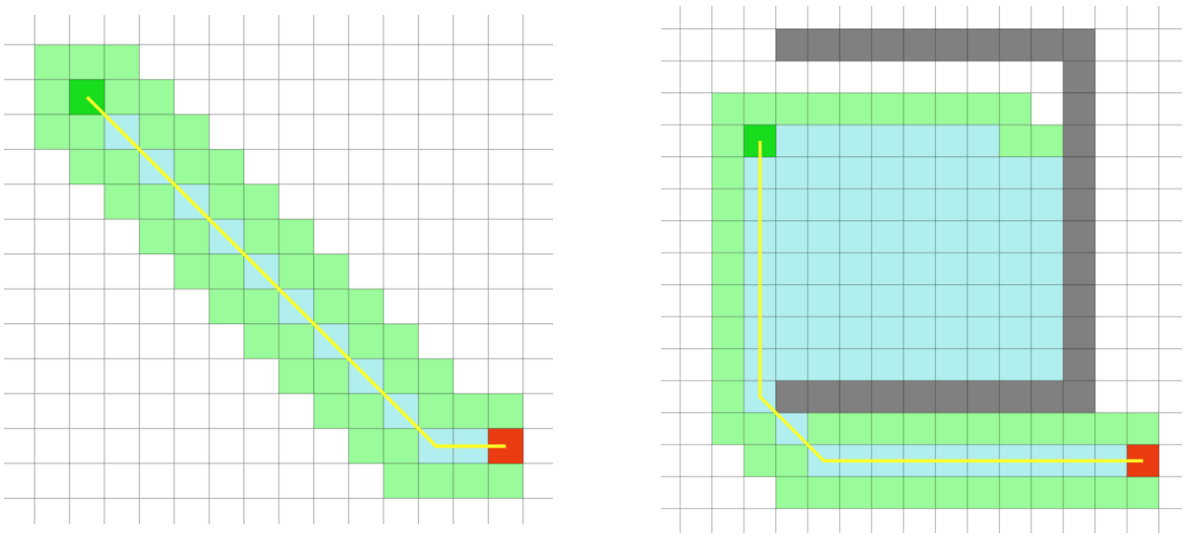
Нехай l – точна вартість шляху від n до цільового вузла, тоді за різних значень евристики маємо:

- Якщо $h(n) = 0$, то вклад до $f(n)$ має лише $g(n)$ і A* перетворюється на алгоритм Дейкстри.
- Якщо $h(n)$ завжди менша або рівна l, то A* гарантовано знаходить найкоротший шлях. Чим далі від точної вартості шляху, тим більше вузлів буде перевірено, що сповільнює алгоритм.
- Якщо $h(n) = l$, то A* буде рухатись лише найкращим шляхом, при цьому без перевірки інших, повільніших шляхів. Це є найкращим випадком для знаходження найкоротшого шляху, який на жаль важко досягнути без жертв у часі.
- Якщо $h(n)$ інколи більша за l, то A* не гарантує знаходження найкоротшого шляху, проте швидкість алгоритму зростає.
- Якщо $h(n)$ – значно більша у порівнянні з $g(n)$, тоді лише значення $h(n)$ стає важливим для $f(n)$ і A* перетворюється в Жадібний BFS.

Проаналізувавши всі варіанти, перед нами постає вибір, де ми обираємо який результат хочемо від A*. Іноді, при розробці гри, в нас можуть бути ситуації, коли нам краще надати перевагу швидкості та мати не найкоротший шлях. Але інколи потрібно саме найкоротший шлях і бажано за мінімальних витрат у часі. Ось чому знання цих властивостей алгоритму є дуже корисним для різних ситуацій.

За тих же самих умов що й раніше, покажемо результати виконання алгоритму A* (рисунок 1.5). Ми помічаємо, що хоч і перевіряємо більше клітинок в порівнянні з Жадібним BFS, але при цьому знаходимо

найкоротший шлях за значно меншу кількість перевірених вузлів, ніж у алгоритмі Дейкстри.



а) *б)*

Рисунок 1.5 – Результат виконання A:*
а – без перешкод; б – із перешкодами

Існує велика кількість різних модифікацій алгоритму A*, але далі буде розглянуто лише певна частина з цікавими властивостями та можливими напрямками покращень основного алгоритму.

1.4.1 IDA*

Почнемо ми розгляд різних модифікацій A* із алгоритму IDA* (англ. Iterative deepening A*). Вперше був представлений Річардом Корфом у 1985 році [7]. Коротко охарактеризувати даний алгоритм, як алгоритм пошуку в глибину (англ. Depth-First Search, DFS) з поєднанням ідеї використання функції евристики при обранні шляху до цілі як в A*. Працює IDA* наступним чином: при кожній ітерації використовуємо пошук в глибину не беручи до уваги гілку, коли її вартість $f(n)$ перевищує

задану межу. Початкове значення межі визначаємо як вартість в початковому стані та з кожною ітерацією це значення буде зростати і обчислюється як мінімальна вартість всіх значень, що перевищили дану межу.

Недоліком використання IDA* у відеоіграх є те, що він збільшує кількість виконуваних операцій за рахунок зменшення використовуваної пам’яті, коли самі вузли зазвичай є простими координатами і не займають багато пам’яті. Такий підхід не надає нам значних переваг в порівнянні зі звичайним алгоритмом A*.

1.4.2 Зважений A*

Суть наступної модифікації полягає у використанні вагового коефіцієнту при обчисленні функції $f(n)$. Так у Зваженому A* (англ. Weighted A*) використовується вага w більша за одиницю для збільшення ефекту евристики:

$$f(n) = g(n) + w * h(n)$$

Таким чином ми пришвидшуємо роботу алгоритму, але знаходимо не найкоротший, або субоптимальний шлях.

Також цікавим підходом є використання так званого динамічного зважування, тобто можна змінювати значення вагового коефіцієнту в залежності від властивостей вузла, що розглядається:

$$f(n) = g(n) + w(n) * h(n)$$

Як показало дослідження [8], краще застосовувати вагу біля кінця шляху. В дослідженні було представлено й порівняно два підходи (рисунок 1.6): $pxWU$, який зменшує вагу біля кінця шляху, та $pxWD$, який збільшує вагу

біля кінця шляху. Результати показали, що варіант зі збільшенням працює краще.

```
f(n) = g(n) < (2*w - 1) * h(n)
     ? g(n) / (2*w - 1) + h(n)
     : (g(n) + h(n)) / w
```

а)

```
f(n) = g(n) < h(n)
     ? g(n) + h(n)
     : (g(n) + (2*w - 1) * h(n)) / w}
```

б)

Рисунок 1.6 – Функція вартості вузла за різних підходів: а – pxWU; б – pxWD

1.4.3 D* та LPA*

Наступні модифікації є алгоритмами інкрементного евристичного пошуку, які дозволяють зміни в навколишньому світі після того, як знайдено оригінальний найкоротший шлях.

LPA* (англ. Lifelong Planning A*) – вперше описаний у 2001 році Свеном Кьонінгом та Максимом Ліхачовим [9]. Його призначення полягає у використанні результатів з попередніх застосувань при зміні вартості шляху без повної перебудови графу.

Алгоритм D* має три варіації у реалізації: оригінальний D* [10], Цілеспрямований D* (англ. Focused D*) [11] та Легкий D* (англ. D* Lite) [12]. Так, перший алгоритм був запропонований Ентоні Стенцом у 1994 році, чия назва походить від "Динамічного A*" (англ. "Dynamic A*"). Другий алгоритм є покращенням ідей першого та реалізований автором оригінального у 1995 році. Останній же алгоритм не базується на попередніх двох і є простішим для реалізації та розуміння. Назва походить від меншої кількості коду, а за продуктивністю він є таким самим або навіть кращим (в залежності від конкретної задачі) за Цілеспрямований D*. Ґрунтується алгоритм на основі LPA*.

Всі три алгоритми вирішують однакове завдання – планування шляху для робота до заданої цілі при невідомому ландшафті. Вони широко застосовуються для мобільних роботів й автономних транспортних засобів. Завдяки своїм перевагам найчастіше застосовують Легкий D*. Насправді навіть автор оригінального алгоритму використовує "легку" версію у своїх дослідженнях. Для прикладу можна навести прототипні навігаційні системи, випробувані на марсоходах Opportunity та Spirit.

Однак і D*, і LPA* вимагають багато місця для зберігання необхідної інформації по шляху для швидкої перебудови. Для гри з великою кількістю рухомих одиниць дані алгоритми краще не застосовувати. Вони більше призначені для одного робота, якому не потрібно визначати шлях для когось ще. Проте якщо у вашій грі лише один або кілька персонажів, яким необхідно знаходити шлях, то, як варіант, можна розглянути запропоновані алгоритми.

1.4.4 Планування шляху під будь-яким кутом

Іноді наша карта представлена не так як ми хотіли б рухатись по ній, наприклад, маючи сітку, ми рухаємось лише з однієї клітинки в іншу, коли ми б хотіли одразу через кілька клітинок для більш плавного руху та коротшого шляху. Можливо, краще просто перемкнутись на граф видимості та застосовувати A* до нього, але додавання нових карт збільшує використовувану пам’ять та витрачає час на їх імплементацію. Найпростіше, що можна зробити, це взяти знайдений шлях та згладити його (вилучивши проміжні вузли), але найкоротший шлях під будь-яким кутом не завжди збігається з найкоротшим шляхом на сітці.

Всі наступні алгоритми теж ґрунтуються на алгоритмі A*. Для покращення ми мусимо взяти до уваги під час пошуку довжини сегментів шляху під будь-яким кутом. Так, алгоритм Theta* [13] враховує це та при

побудові вказівників на батьківські вузли, він вказуватиме безпосередньо на вузол, між яким є пряма видимість, пропускаючи вузли між ними. Даний алгоритм знаходить коротший шлях, ніж просте згладження шляху, але без гарантії, що він буде найкоротшим. Також він працює лише на сітках з однаковою вартістю. Лінива Theta* (англ. Lazy Theta*) [14] є покращенням, яке має менше перевірок на пряму видимість між вузлами за рахунок лінивих обчислень.

Алгоритм Anya [15] знаходить найкоротший шлях під будь-яким кутом, обмежуючи простір пошуку до натягнутих шляхів (англ. Taut Paths, шлях, де кожна зміна курсу щільно обгортається навколо якоїсь перешкоди). Це найшвидший відомий онлайн оптимальний метод. Алгоритм Polyanya [16] є різновидом Anya, який працює для навігаційного мешу (див. розділ 2.4) замість звичайної сітки.

1.5 Генетичний алгоритм

Генетичний алгоритм (англ. Genetic Algorithm, GA) належить до метаевристичних алгоритмів, ідея якого основана на процесі природного відбору, що є частиною більшого класу еволюційних алгоритмів. GA був представлений Джоном Холландом [17]. Його застосовують в багатьох областях, таких як біометричні винаходи, інженерний дизайн та робототехніка. Він використовує оператори, натхненні біологічними процесами, як мутація, схрещення та відбір. При розробці відеоігор даний алгоритм також можна застосовувати як алгоритм пошуку шляху. Так, дослідження [18] показало, що GA має кращу продуктивність на мапі з перешкодами, ніж Жадібний BFS. Інше дослідження показало, що пошук в реальному часі з GA (RTP-GA) є швидшим за A* [19].

1.6 ACO

Алгоритм оптимізації мурашиної колонії (англ. Ant Colony Optimization, ACO) – метаевристична техніка, натхненна поведінкою мурах, при просуванні шляхом від мурашнику до їжі, що вперше була запропонована Марко Доріго [20]. Спершу, кожна мураха починає хаотично рухатись та намагається знайти їжу. Знайшовши їжу, вони починають спілкуватись один з одним за допомогою феромонів, які мурахи виділяють в навколишнє середовище, для визначення найкоротшого шляху. ACO має застосування у багатьох сферах як календарне планування, робототехніка, маршрутизація транспорту, оптимізація форми антен тощо. Однією з переваг ACO над A* є можливість динамічної адаптації до змін у реальному часі. Одне з досліджень показало, що ACO може бути кращим за GA у часі [21].

РОЗДІЛ 2. ВАРІАЦІЇ КАРТ

2.1 Важливість обрання карти

Алгоритми пошуку шляху, як правило, гірші за лінійні: якщо подвоїти відстань до цілі, то пошук шляху займе часу більше ніж у два рази. Загалом, чим менше вузлів у представленні карти, тим швидше буде працювати алгоритм. Карта, що використовується для пошуку шляху, не мусить збігатися з картою, що використовується для інших речей у грі. Однак використання одного й того ж представлення є хорошим початком, поки не виникне необхідність в чомусь іншому.

Обране представлення карти може значно змінити продуктивність і якість знайденого шляху.

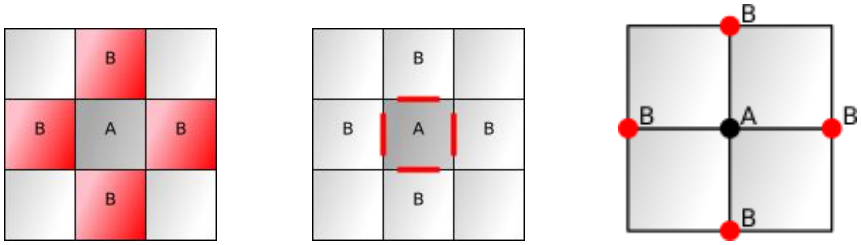
2.2 Сітка

Сітки рівномірно розділяють ігровий світ на невеликі правильні фігури, які інколи називають "плитками" (англ. "tiles"). Зазвичай використовують квадратні (рисунок 2.1) та шестикутні (рисунок 2.2) сітки. Таке представлення є легким для розуміння та використовується в багатьох іграх. Використання сітки також може бути корисним, коли вартість переміщення по плиткам не є однорідною.

Навіть при переміщенню по сітці є три можливі способи руху (рисунок 2.3): по плиткам, по ребрам і по вершинам. Вибором за замовчуванням є рух по плиткам, особливо коли в грі можливий рух тільки через центр плитки. Як варіант, можна дозволити рух по діагоналі за таку ж або більшу вартість. Якщо ви можете рухатись будь-де в межах сітки (не тільки через центри плиток), то рух по ребрам та вершинам може бути кращим варіантом.

Рисунок 2.1 – Квадратна сітка у грі Mario + Rabbids Kingdom Battle

Рисунок 2.2 – Шестикутна сітка у грі Civilization VI



а) *б)* *в)*

Рисунок 2.3 – Рух по: а – плиткам; б – ребрам; в – вершинам

2.3 Полігональні карти

Найпоширенішою альтернативою до сітки є полігональне представлення (рисунок 2.4). Якщо вартість переміщення на великих площах є рівномірною, і якщо можливий вільний рух не по сітці, то можна використати дане представлення. В даному випадку найкоротший шлях буде проходити через кути перешкод і також необхідно додати до графу початок та кінець руху, які будуть додаватись кожен раз заново за виклик пошуку шляху. Також нам необхідно якось ці точки з’єднати, і найпростішим рішенням є побудова графу видимості: з’єднуємо пари точок, які видно одна з одної. За вартість переміщення можна взяти довжину ребра.

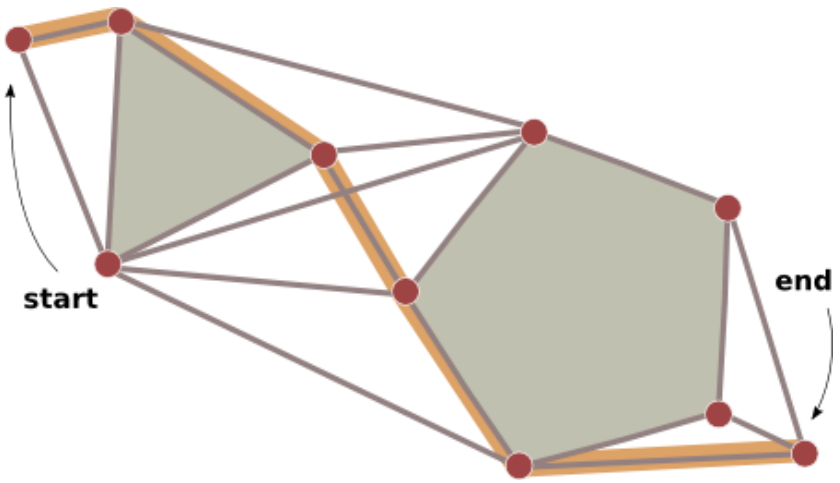


Рисунок 2.4 – Знаходження шляху при полігональному представленні

Попередній рисунок є простим прикладом, проте можуть виникати проблеми на деяких картах з великою кількістю відкритих ділянок або довгих коридорів (рисунок 2.5). Це виникає у зв'язку з тим, що за n для вершин у нас може бути до n^2 ребер. Велика кількість ребер впливає на швидкість алгоритмів пошуку та розміри використовуваної пам'яті. Іншим недоліком є постійне додавання та вилучення початкових та кінцевих вузлів разом з ребрами.

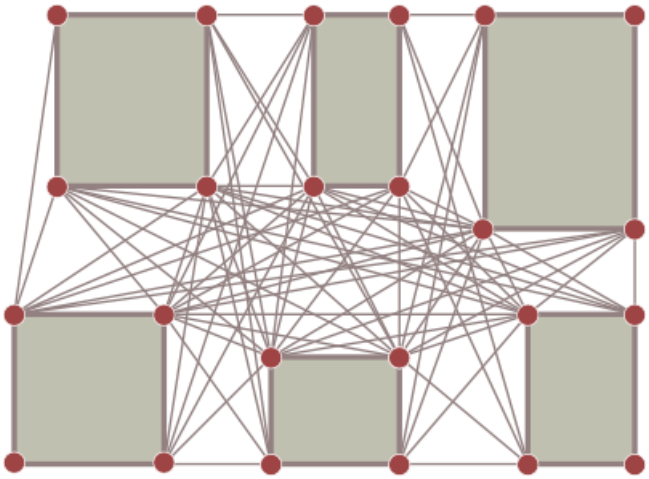


Рисунок 2.5 – Проблема великої кількості ребер при полігональному представлені

2.4 Навігаційний меш

Навігаційний меш або навігаційна сітка (англ. navmesh або navigation mesh) представляє у вигляді опуклих багатокутників, які між собою не перетинаються, область по якій можна пересуватись. Можна сказати, що даний спосіб є протилежним до попереднього, адже він представляв у вигляді полігонів перешкоди. На рисунку 2.6 показано попередній приклад, але вже використовуючи навігаційний меш.

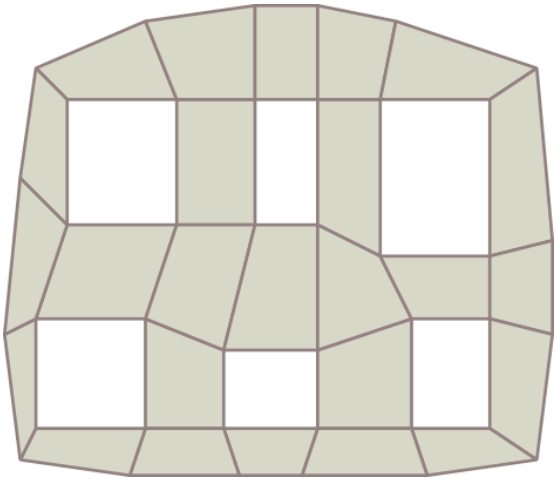


Рисунок 2.6

Полігони навігаційного мешу можуть мати різну додаткову інформацію (наприклад, "вартість пересування – 3" або "потребує здатності плавати"). Так само як і для сітки, у нас є можливість пересуватись через центри, ребра або вершини полігонів.

При переміщенні через центри полігонів, спочатку необхідно додати початковий та кінцевий вузол разом з ребром до центру відповідного полігону (рисунок 2.7). На прикладі жовтим показано знайдений шлях таким способом, а рожевим – найкоротший шлях.

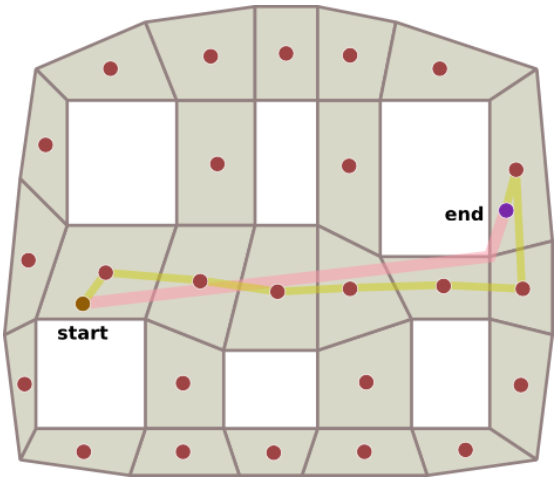


Рисунок 2.7 – Переміщення через центри полігонів

Зазвичай, нам не потрібно рухатись через центри полігонів. Замість цього можна спробувати рухатись через ребра між сусідніми полігонами. На рисунку 2.8 показаний рух через центри ребер, але можна і збільшити кількість обраних точок на ребрі.

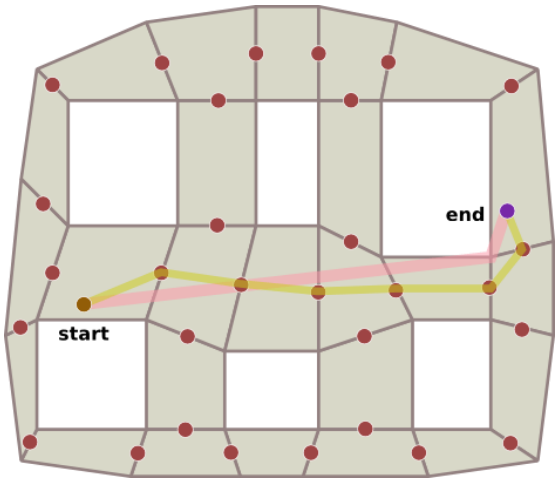


Рисунок 2.8 – Переміщення через центри ребер

Ще одним способом є переміщення через вершини полігонів. На рисунку 2.9 спостерігаємо, що шлях був би найкоротшим, якби ми з’єднали б одразу початковий вузол і вершину перешкоди, а так маємо зайві кроки.

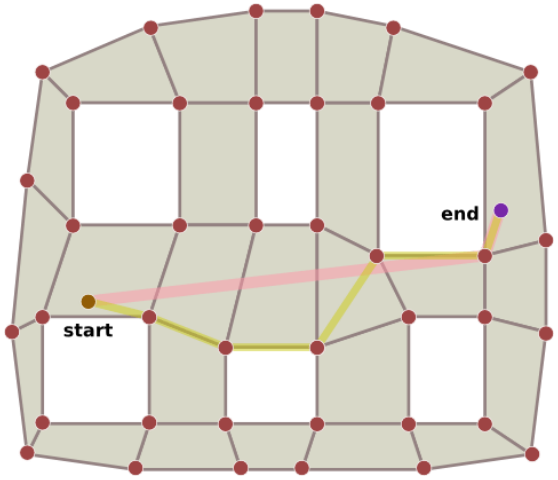


Рисунок 2.9 – Переміщення через вершини полігонів

Можливе також поєднання різних способів, як на рисунку 2.10, де поєднуються переміщення через вершини та через ребра, але варто враховувати, що це збільшує кількість можливих шляхів, які потрібно перевірити, збільшує затрати пам’яті та ускладнює реалізацію.

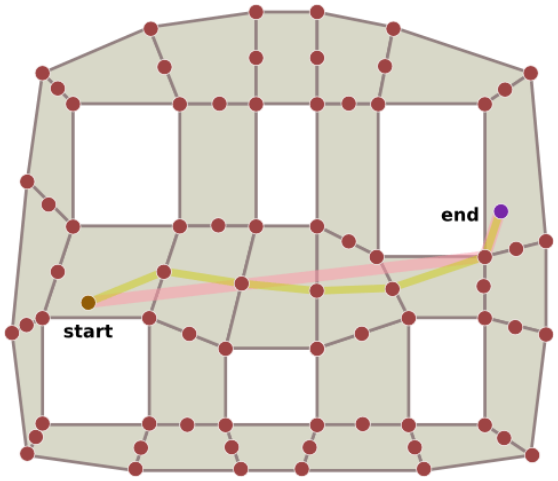


Рисунок 2.10 – Переміщення через вершини та центри ребер

До наведених вище прикладів можливо застосувати згладження шляху (вилучення проміжних вузлів), як і у випадку сітки, що дасть нам кращий шлях і в деяких випадка буде збігатись з найкоротшим, проте це не гарантує нам отримання найкоротшого шляху.

2.5 Інші проблеми з картами

При побудові карт для пошуку шляху можливий випадок, коли карти мають складну структуру та є великими за розмірами. У таких випадках може допомогти ієрархічна структура для пошуку шляху. Наприклад, нам потрібно дістатись до магазину в іншому місті. Так, можна розділити проблему: для кожного міста локально знайти найкоротший шлях по вулицям і окремо знайти найкоротший шлях між містами. Іншим

прикладом може бути наявність багаторівневих локацій, де можливе переміщення між рівнями.

В деяких випадках у нас може бути відсутній шлях, тобто локації між собою не з’єднані. В таких випадках варто перед застосуванням алгоритму пошуку робити перевірку чи можливий шлях, інакше алгоритму доведеться перевірити всі вузли в графі, перш ніж дати негативну відповідь.

РОЗДІЛ 3. РОЗРОБКА НАВЧАЛЬНОГО ПРОЄКТУ

3.1 Вибір технологій та реалізація

Основним критерієм при обранні технологій для створення навчального проєкту були популярність, актуальність у сучасному світі та можливість використання як основу для більших проєктів. Так, одними з найпопулярніших й найвідоміших ігрових двигунів є Unity [22] та Unreal Engine [23]. Unreal Engine використовують частіше для розробки високобюджетних ігор досвідченою великою командою, а Unity має репутацію як двигун, який обирають маленькі інді студії для розробки невеликих ігор з набагато меншим бюджетом. Саме тому було обрано Unity 2021.3.3f1 як останню LTS версію. Мовою програмування була обрана C# 9.0 згідно з вимогами даної версії Unity. В якості IDE виступає Microsoft Visual Studio 2022, оскільки Unity має хорошу інтеграцію з даним редактором коду.

Як показує дослідження (рисунок 3.1) [24], алгоритм A* був і залишається найпопулярнішим алгоритмом для реалізації пошуку шляху при розробці ігор, тому було вирішено реалізувати саме його. Цей алгоритм хороший для навчального проєкту ще тим, що є відносно зрозумілим та має велику кількість модифікацій, що дозволяє доповнювати проєкт і краще засвоїти матеріал при вивченні алгоритмів пошуку.

Реалізований навчальний проєкт містить симуляцію знаходження найкоротшого шляху для кількох об’єктів до різних цілей на сітці, використовуючи алгоритм A* (рисунок 3.2). Червоним зображено цілі, зеленим – стартові вузли, а оранжеві лінії з фіолетовими вузлами – найкоротші шляхи. Програма знаходить оптимальні шляхи для кожного стартового вузла, звертаючись до менеджера задач, та здійснює переміщення до цілей за знайденими шляхами.

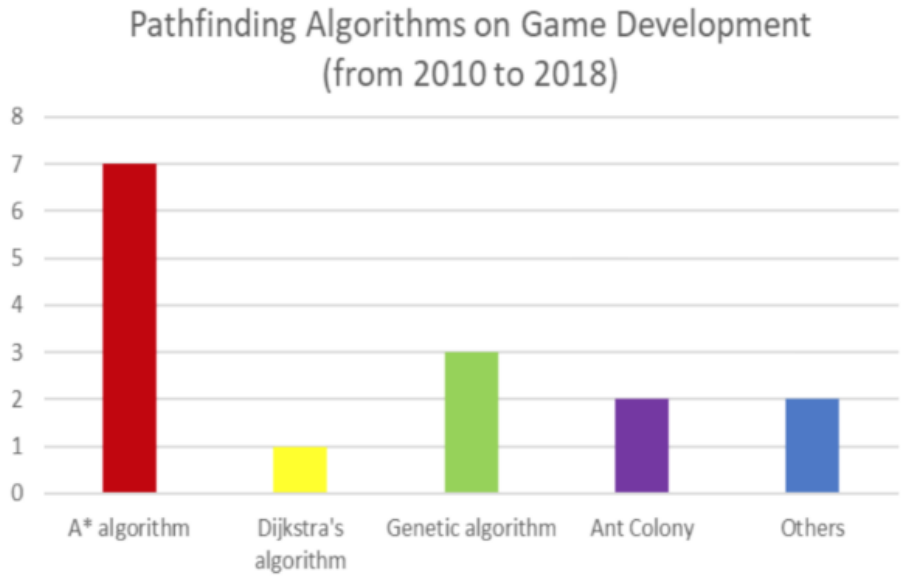
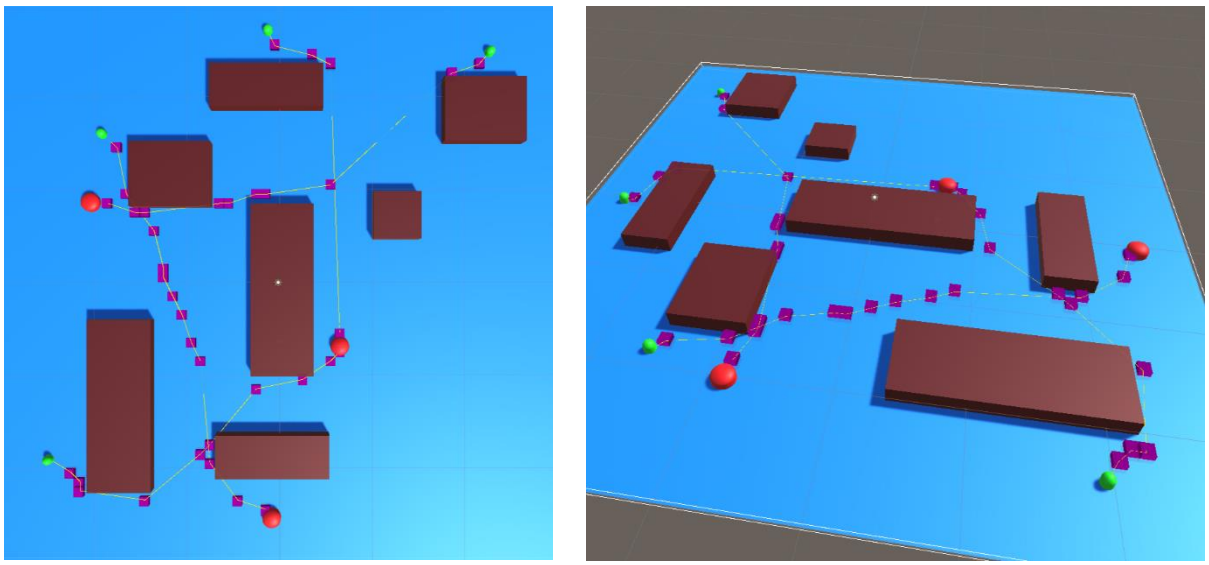


Рисунок 3.1



а) *б)*

Рисунок 3.2 – Симуляція A в навчальному проєкті з різних ракурсів: а – вид зверху; б – вид збоку*

Найповільнішою частиною алгоритму є постійне обрання вузлів в циклі за найменшим значенням функції $f(n)$. Саме тому було вирішено оптимізувати дану частину коду реалізувавши таку структуру даних, як

двійкова купа для зберігання вузлів (про доцільність використання цієї структури даних [25]).

3.2 Подальший розвиток

Надалі планується реалізація інших алгоритмів пошуку шляху в навчальному проєкті. З розвитком AI з’являються нові та модифікуються старі алгоритми пошуку, які на даний момент мають невелику кількість літератури, тому планується провести порівняння таких алгоритмів.

ВИСНОВКИ

а. У результаті роботи було розглянуто поняття алгоритму пошуку шляху і його застосування у сфері відеоігор.

б. Розглянуто поняття штучного інтелекту та його зв’язок з алгоритмами пошуку шляху.

в. Детально розглянуто основні алгоритми пошуку шляху, які застосовуються при розробці відеоігор, описано їх основний принцип роботи, проведено деякі порівняння та наведено їхні недоліки та переваги.

г. Розглянуто основні типи представлення карт ігрового середовища та вплив певного типу на швидкість роботи алгоритмів і необхідні для цього затрати пам’яті. Наведено приклади проблем, які виникають при реалізації карт ігрових рівнів.

д. Обрано інструменти розробки та алгоритм пошуку для реалізації навчального проєкту на основі популярності й актуальності в сучасному світі.

е. Отримані теоретичні знання використані для розробки навчального проєкту. Реалізовано алгоритм пошуку шляху A* з оптимізацією на основі бінарної купи для можливості дослідження його властивостей роботи і можливості створення на його основі різних модифікацій.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Bultiko V. Real-time Heuristic Search for Pathfinding in Video Games. / V. Bultiko, Y. Bjornsson, N. R. Sturtevant, R. Lawrence. – Artificial Intelligence for Computer Games, Springer, 2011. – pp. 1-30.
2. Dijkstra E.W. A note on two problems in connexion with graphs / E.W. Dijkstra – Numer, Math, 1959. – Vol. 1. – pp. 269-271.
3. Dijkstras shortest path algorithm visual introduction [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://www.freecodecamp.org/
4. Pathfinding.js [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://qiao.github.io/PathFinding.js/visual/
5. Russell S. J. Artificial Intelligence: A Modern Approach / S. J. Russell, P. Norvig – Upper Saddle River, New Jersey: Prentice Hall. – 2nd ed. 2003. – pp. 94-95 (note 3). – ISBN 0-13-790395-2.
6. A* Search [Електронний ресурс] // Brilliant. – 2022. – Режим доступу до ресурсу: https://brilliant.org/
7. Korf R. E. Depth-first Iterative-Deepening: An Optimal Admissible Tree Search / R. E. Korf – Artificial Intelligence, 1985. – Vol. 27. – pp. 97-109.
8. Chen J. Necessary and Sufficient Conditions for Avoiding Reopenings in Best First Suboptimal Search with General Bounding Functions / J. Chen, N. R. Sturtevant – AAAI Conference on Artificial Intelligence, 2021. – Vol. 35(5). – pp. 3688-3696.
9. Koenig S. Incremental A* / S. Koenig, M. Likhachev – Neural Information Processing Systems, 2001. – pp. 1539-1546.

10. Stentz A. Optimal and Efficient Path Planning for Partially-Known Environments / A. Stentz – International Conference on Robotics and Automation, 1994. – pp. 3310-3317.
11. Stentz A. The Focussed D* Algorithm for Real-Time Replanning / A. Stentz – International Joint Conference on Artificial Intelligence, 1995. – pp. 1652-1659.
12. Koenig S. Fast Replanning for Navigation in Unknown Terrain / S. Koenig, M. Likhachev – Transactions on Robotics, 2005. – Vol. 21(3). – pp. 354-363.
13. Nash A. Theta*: Any-Angle Path Planning on Grids / A. Nash, K. Daniel, S. Koenig, A. Felner – AAAI Conference on Artificial Intelligence, 2007. – pp. 1177-1183.
14. Nash A. Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D / A. Nash, S. Koenig, C. Tovey – AAAI Conference on Artificial Intelligence, 2010.
15. Harabor D. Optimal Any-Angle Pathfinding In Practice / D. Harabor, A. Grastien, D. Oz, V. Aksakalli – Journal of Artificial Intelligence Research, 2016. – Vol. 56. – pp. 89-118.
16. Cui M. Compromise-free Pathfinding on a Navigation Mesh / M. Cui, D. Harabor, A. Grastien – Twenty-Sixth International Joint Conference on Artificial Intelligence, 2017. – pp. 496-502.
17. Car J. An Introduction to Genetic Algorithms. / J. Car – Artif. Life, 2014. – Vol. 3(1). – pp. 63-65.
18. Santos U. O. Pathfinding Based on Pattern Detection Using Genetic Algorithms / U. O. Santos, A. F. V. Machado, E. W. G. Clua – XI Brazilian Symp. Games Digit. Entertain., 2012. – pp. 64-72.
19. Machado A. F. V. Real time pathfinding with genetic algorithm / A. F. V. Machado, U. O. Santos, H. Vale – Brazilian Symp. Games Digit. Entertain. SBGAMES, 2011. – pp. 215-221.

20. Dorigo M. Ant colony optimization theory: A survey / M. Dorigo, C. Blum – Theor. Comput. Sci., 2005. – Vol. 344(2–3). – pp. 243-278.
21. Hunkeler I. fairGhosts – Ant colony controlled ghosts for Ms. Pac-Man / I. Hunkeler, F. Schär, R. Dornberger, T. Hanne – IEEE Congress on Evolutionary Computation (CEC), 2016. – pp. 4214-4220.
22. Unity [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://unity.com/.
23. Unreal Engine [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://www.unrealengine.com/.
24. Rafiq A. Pathfinding Algorithms in Game Development / A. Rafiq, T. A. A. Kadir, S. N. Ihsan – IOP Conf. Ser.: Mater. Sci. Eng., 2020. – Vol. 769. – 012021.
25. Using Binary Heaps in A* Pathfinding [Електронний ресурс]:[Веб-сайт] – Режим доступу до ресурсу: https://web.archive.org/web/20171015123739/http://www.policyalmanac.org/games/binaryHeaps.htm