Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Використання штучного інтелекту
у розробці мобільних додатків
(тема)

Виконав:
студент 2 курсу, групи СШІм-20-3
Стрельченя Д. Ю.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту
(повна назва спеціалізації)

Керівник доц. Узлов Д. Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ В.О. Філатов
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Штучного інтелекту
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системи штучного інтелекту (СШІ)
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«_____» _____ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Стрельчені Данилу Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Використання штучного інтелекту у розробці мобільних додатків

затверджена наказом університету від 28 березня 20 22 р. № 414 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 16 травня 20 22 р.

3. Вихідні дані до роботи книги, документація середовища розробки, курси розробки ігор, доповіді, інтернет джерела, курси геймдизайну, освітній ступінь першого рівня

4. Перелік питань, що потрібно опрацювати в роботі проаналізувати предметну область, вивчити теоретичний матеріал, вивчити алгоритми створення штучного інтелекту, спроектувати додаток, вибрати середовище розробки, реалізувати повноцінний продукт, проаналізувати результат

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Рисунок 1.1 – Розділи штучного інтелекту, Рисунок 1.10 – Порівняння алгоритмів Дейкстри та A*, Рисунок 2.7 – Макет для хвильового алгоритму, Рисунок 2.13 – Результати роботи хвильового алгоритму, Рисунок 3.10 – Функціонування основного ігрового циклу, Рисунок 3.15 – Тіло методу «SetCustomerDestination», Рисунок 3.16 – Реалізація курутини «CheckCustomersState», Рисунок 3.17 – Завершення усіх заказів, Рисунок 3.18 – Перевірка завершення поточного замовлення, Рисунок 3.21 – Курутина «SellingProcess» обслуговування клієнтів, Рисунок 3.22 – Курутина «CheckCustomerExitState», Рисунок 3.23 – Метод «FindWorkerTarget», Рисунок 3.24 – Структура умов зміни станів співробітників магазину, Рисунок 3.25 – Демонстрація роботи ШІ покупців, Рисунок 3.26 – Демонстрація роботи ШІ співробітників

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання	28.03.2022	Виконано
2	Аналіз предметної області	28.03.2022–29.03.2022	Виконано
3	Аналіз теоретичного матеріалу	29.03.2022–31.03.2022	Виконано
4	Вивчення алгоритмів	01.04.2022–03.04.2022	Виконано
5	Постановка задачі	04.04.2022	Виконано
6	Вибір програмного забезпечення для розробки	05.04.2022	Виконано
7	Проектування додатка	06.04.2022–08.04.2022	Виконано
8	Розробка повноцінного продукта	09.04.2022–23.04.2022	Виконано
9	Публікація продукта	24.04.2022	Виконано
10	Оформлення пояснювальної записки	25.04.2022–03.05.2022	Виконано
11	Попередній захист	13.05.2022	Виконано
12	Захист кваліфікаційної роботи	16.05.2022	Виконано

Дата видачі завдання 28 березня 2022 р.

Студент _____
(підпис)

Керівник роботи _____ _____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 79 с., 55 рис., 3 дод., 10 джерел.

ДОДАТОК, ІГРИ, ШТУЧНИЙ ІНТЕЛЕКТ, ANDROID, CONSTRUCT3, GAMEDEV, HYPERCASUAL, IOS, UNITY

Об'єкт дослідження – мобільний ігровий додаток зі штучними інтелектуальними агентами.

Мета роботи – створення hypercasual ігрового додатку для мобільних пристроїв у середовищі розробки Unity, з подальшою публікацією на популярних майданчиках.

Область застосування – розваги та відпочинок у мобільному гаджеті для людей будь-якого віку.

Методи дослідження – аналіз статей, технічної літератури, доповідей, теоретичного матеріалу, вивчення та реалізація алгоритмів, вивчення документації, аналіз ринку мобільних hypercasual ігор, прототипування і практична реалізація.

В ході роботи проведено аналіз існуючих мобільних ігрових hypercasual додатків, були отримані практичні навички роботи у середовищі Unity, вивчені основні алгоритми реалізації штучних інтелектуальних агентів та розроблені вимоги до додатка. В результаті розроблено багатоплатформний додаток симулятор магазину з інтелектуальними агентами у вигляді покупців та співробітників, зроблено експорт під операційну систему Android, здійснена публікація до мобільного майданчика Google Play та вже отримано більше 10 тисяч завантажень.

ABSTRACT

Explanatory note: 79 p., 55 fig., 3 ann., 10 sources.

ANDROID, APPLICATION, ARTIFICIAL INTELLIGENCE, CONSTRUCT3, GAMEDEV, GAMES, HYPERCASUAL, IOS, UNITY

The object of study – a mobile gaming application with artificial intelligence agents.

The purpose of the work is to create a hypercasual gaming application for mobile devices in the Unity development environment, with subsequent publication on popular sites.

Scope – entertainment and recreation in a mobile gadget for people of all ages.

Research methods – analysis of articles, technical literature, reports, theoretical material, study and implementation of algorithms, study of documentation, market analysis of mobile hypercasual games, prototyping and practical implementation.

In the course of the work the analysis of the existing mobile game hypercasual applications was carried out, practical skills of work in the Unity environment were received, the basic algorithms of realization of artificial intelligent agents were studied and requirements to the application were developed. As a result, a multi-platform store simulator application with intelligent agents in the form of customers and employees was developed, exported to the Android operating system, published to the Google Play mobile platform, and more than 10,000 downloads have already been received.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Макет – Layout – Заздалегідь підготовлений набір об'єктів;

ОС – Операційна система;

Референс – Допоміжний файл для вивчення, у якому може знаходиться приблизний бажаний результат;

ШІ – Штучний інтелект;

App Store – Сервіс для розміщення додатків під ОС IOS;

C# – Мова програмування;

Construct 3 – Програмне забезпечення для розробки двовимірних ігор.

Free-to-play – Система монетизації та спосіб розповсюдження ігор, при якому користувач може грати без обов'язкового внесення грошових коштів;

Google Play – Сервіс для розміщення додатків під ОС Android;

HTML5 – HyperText Markup Language, version 5 – стандартизована мова розмітки документів в інтернеті;

Hypercasual – Спосіб монетизації (іноді виділять як жанр), в основі якого закладено основний заробіток на показі реклами;

Idle – Жанр в іграх;

In-App Purchases – Покупка товарів, послуг або цифрового контенту у додатку;

Unity – Програмне забезпечення для розробки двовимірних та тривимірних ігор з використанням мови програмування C#.

ВСТУП

Розвивається весь світ, особливо технології, а штучний інтелект зараз можна зустріти навіть у кавомашині. Сфера ігор теж не стоїть на місці, створюється нове програмне забезпечення для розробки, пишуться уроки для дорослих і дітей, адже тепер навіть з юного віку можна почати створювати власні ігри. Зростає і продуктивність, процесори стають більш технологічними та енергоефективними, завдяки чому потужність деяких смартфонів перевершує ноутбуки та комп'ютери, що дозволяє з легкістю виконувати складні завдання, наприклад – запускати 3D ігри з великою кількістю моделей та складними обчисленнями.

Багато людей використовують свої гаджети для відпочинку, зазвичай таким відпочинком є перегляд фото та відео або мобільні ігри. З початку пандемії почали грати навіть ті люди, які раніше використовували телефон виключно для дзвінків і будильника.

На майданчиках існують тисячі, а то й мільйони різних ігор, платні та безкоштовні, аркади та головоломки, 3D та 2D, а в останні роки активно розвивається сфера Hypercasual, вони дуже легкі для користувача, необхідно лише кілька секунд, щоб зрозуміти геймплей, а грати зручно тримаючи телефон однією рукою та використовуючи лише один палець. Відрізняються Hypercasual ігри і короткою довгою сесією, час проходження одного рівня може становити 20–30 секунд, проходження фази 200–300 секунд, завдяки цьому користувачі можуть грати під час очікування черги до лікаря або в магазині, поки їдуть у метро або чекають на подачу страви в ресторані, навіть між підходами у тренажерному залі. Однак, незважаючи на всю простоту, дані ігри можуть бути складні в розробці та складатися з десятків компонентів, за результатом можуть ховатися години роботи дизайнера, художника, розробника та на прикладі однієї такої я покажу процес розробки та використання штучного інтелекту.

Крім розважальних цілей, ігри бувають корисними. Існує безліч освітніх та розвиваючих ігор. Наприклад, існують шахи, шашки, морський бій – це приклад простих класичних ігор, де штучний інтелект може використовуватись як суперник для гравця. Для дітей батьки завантажують на телефон або планшет спеціально розроблені розважально-розвиваючі ігри та дають своїй дитині. Мені доводилося розробляти такий проект у бакалаврській роботі.

В ході цієї кваліфікаційної роботи мною будуть досліджені способи застосування штучного інтелекту в мобільних іграх, теоретичний матеріал алгоритмів, буде розглянуто та реалізовано практичну частину у різних програмних забезпеченнях. Підсумковим результатом стане розроблена та опублікована hypercasual мобільна гра з реалізованим у ній штучним інтелектом на основі здобутих навичок.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Поняття ШІ

Насамперед необхідно з'ясувати, що являє собою штучний інтелект. Штучний інтелект – це система, яка здатна виконувати не лише математичні операції, а й імітувати людську поведінку, а саме – виконувати інтелектуальні та творчі завдання. Для створення штучного інтелекту виділяють кілька підходів:

- логічний;
- агентно-орієнтований;
- машинне навчання;
- нейромережі.

За здібностями штучний інтелект поділяють на 3 групи:

- слабкий, коли система може справлятися лише з обмеженою задачею, а її налаштування повністю робить людина;
- загальний, він же спеціалізований, коли система може вирішувати певну задачу краще людини;
- сильний, коли система зможе розуміти інформацію як людина.

Наразі майже весь штучний інтелект можна назвати слабким або загальним, бо він працює за чітко заданим людиною алгоритмом. Від дійсно сильного ШІ нас можуть відділяти сотні років досліджень та спроб розробки. Отже, поки що сконцентруємось на тому, що маємо.

Логічний підхід полягає у моделюванні міркувань. Ми намагаємося імітувати систему логіки, але в дуже обмеженому вигляді. Хорошим прикладом є мова Пролог. У мові існують терми, факти та умови. Факти представляють конкретну інформацію, правила вказують на логічний висновок з умовою, а терми є деякими змінними.

В агентно-орієнтованому підході програма здатна виконувати наперед задані користувачем вказівки протягом тривалого часу. Зазвичай агенти

беруть інформацію про навколишній світ за допомогою датчиків або, як часто буває в іграх, дані про навколишній світ заздалегідь записані в агента. Завданнями може бути пошук шляху чи прийняття рішень.

Машинне навчання є одним із розділів штучного інтелекту (рисунок 1.1). Суть такого підходу полягає у самостійному отриманні інформації у процесі роботи системи. Для навчання системі необхідна величезна кількість вхідних даних і чим більше, тим краще кінцевий результат. Навчання можливо з учителем, у разі дані відзначаються людиною і вирішуються завдання регресії і класифікації. Можливо навчання без вчителя, у разі система сама знаходить залежності між об'єктами, можуть вирішуватися завдання кластеризації і пошуку асоціацій.

Рисунок 1.1 – Розділи штучного інтелекту

Нейронна мережа – це один із алгоритмів машинного навчання. Працює за принципом біологічних нейронних мереж у людини. Система складається з вхідного шару, на вхід якого подається інформація, n прихованих шарів, які займаються обробкою даних, та вихідного шару, який виводить підсумковий результат.

1.2 Штучний інтелект в іграх

В іграх штучний інтелект використовують все частіше, зазвичай у ролі керуючої системи об'єкта[1]. Система діє виходячи з прописаних умов і вказує на дії об'єкта, він же «інтелектуальний агент». В якості агентів можуть виступати ігрові персонажі, боти або навіть абстрактні сутності, як наприклад погода на сцені. Весь набір керування та дій називають циклом «Відчувати/Мислити/Діяти».

Агент отримує інформацію про навколишнє середовище, це може бути відстань до ворога, кількість здоров'я, сила шкоди або будь-які інші значущі показники – це процес «відчувати». Наступним кроком агент обробляє дані, наприклад, чи достатньо енергії для здійснення стрибка, після чого створює кроки своїх дій. У заключному кроці агент робить набір вибраних раніше інструкцій, наприклад, застосовує фатальний удар, якщо він доступний і ворог у зоні видимості. У реальному світі система може бути забезпечена десятками або навіть сотнями датчиків, щоб зчитувати стан зовнішнього світу, але в іграх все набагато простіше, адже більшість даних вже є частиною гри. Нам не потрібно реалізовувати алгоритми розпізнавання, щоб визначити, що об'єкт, що стоїть попереду, є машиною або ворогом, або об'єктом для збору. Розглянемо простий приклад, де в ролі агента буде персонаж, завдання якого збирати монети на карті, які створюються з деякою періодичністю в кількості 1–2 штуки.

Як писав раніше, в іграх агенту заздалегідь відомо, як саме виглядає монета, нам досить просто вказати посилання на необхідний об'єкт або зробити позначку на самому об'єкті. Таким чином завдання спрощується і на перший погляд цілком просте рішення – вказувати агенту точку для руху в напрямку монети, проте в такому випадку обиратиметься випадкова монета і шлях не завжди виявиться оптимальним, адже ми можемо пройти всю карту замість того, щоб слідувати до найближчої мети (рисунок 1.2).

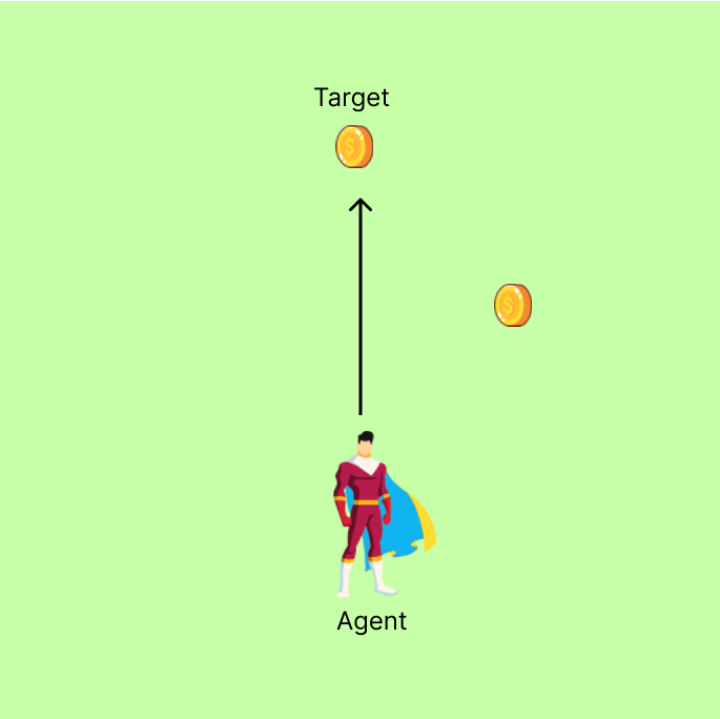

Рисунок 1.2 – Прийняття рішення агентом

Як ми бачимо, на шляху до верхньої монети у персонажа була ще одна справа, але наше рішення не враховує такої ситуації (рисунок 1.3).

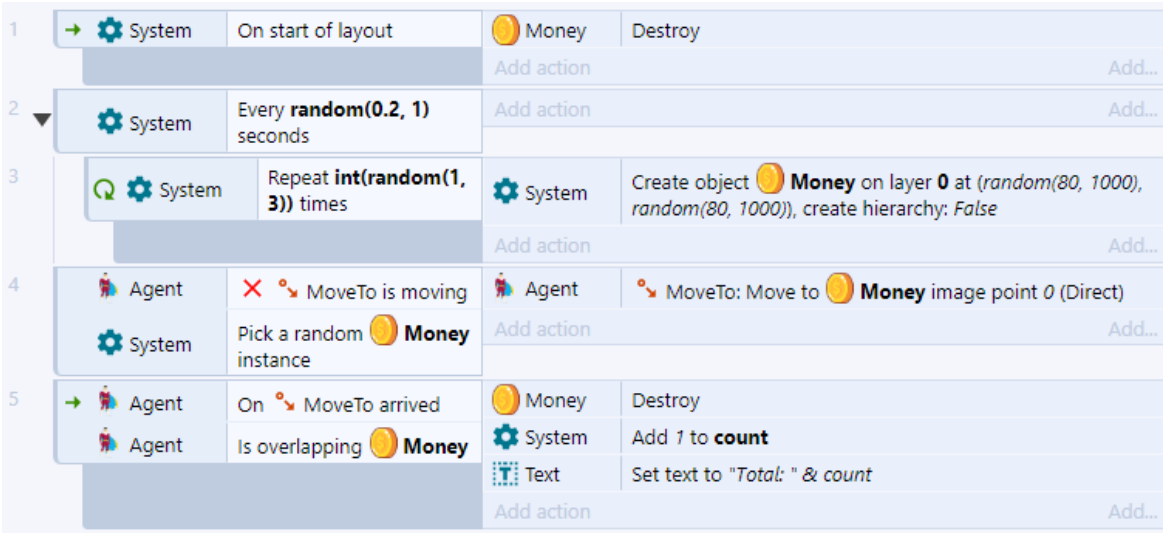

Рисунок 1.3 – Рішення простого пошуку об’єкта

Якщо у нас стоятиме завдання зібрати максимум монет за обмежений час, то цей агент явно програє своїм конкурентам. Щоб його удосконалити, достатньо шукати шлях до найближчої монети після підбору попередньої. Реалізувати це можна одним рядком, а коефіцієнт корисної дії зросте приблизно в півтора рази, як показав проведений експеримент. Можна провести ще кілька покращень – шукати найближчу монету в процесі руху або шукати найбільше скупчення монет під час руху, щоб разом підібрати їх усі. Допишемо кілька рядків і проведемо повторний експеримент, запустивши виконання обох програм на 30 секунд, наприкінці виведемо результати (рисунок 1.4).

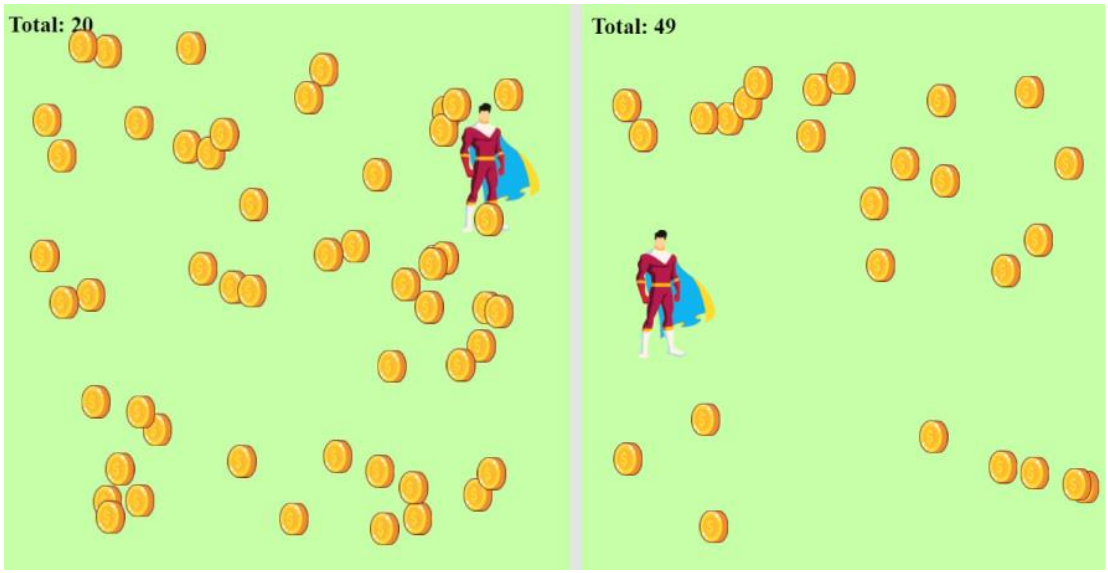

Рисунок 1.4 – Результати роботи двох агентів

Результат вражає, перший агент за 30 секунд встиг зібрати 20 монет, за цей час другий агент зібрав практично 50 монет, це в два з половиною рази більше. На такому простому прикладі було розібрано завдання прийняття базових рішень штучним інтелектом, у результаті якого ми побачили різницю у роботі алгоритмів двох ігрових агентів з однаковими вхідними даними.

Ігровий штучний інтелект часто має ряд обмежень:

- спрощення, щоб приносити гравцю розвагу від гри, а не домінування над ним;
- реалістичність, щоб у гравця складалося враження, що він грає проти реальної людини;
- швидкодія, щоб не створювати зайвого навантаження.

Деякі ШІ реалізуються за допомогою кінцевого автомата (Finite state machine) – це коли агент має чіткий поточний стан і конкретні можливості переходу в інше. Сама назва говорить нам про те, що кількість станів скінчена. Особливість цієї системи в тому, що в будь-який момент часу агент знаходиться в одному стані, з можливістю перейти в інше за наявності такого зв'язку.

Як приклад розглянемо агента ворога, який може патрулювати, атакувати, тікати – це його стану. Ворог патрулює свою зону, а якщо побачить гравця, то атакує його – це умова переходу з першого стану в друге. Якщо під час атаки у ворога залишається мало здоров'я, то він починає тікати. Якщо ворог тікає і втратив гравця з виду – знову патрулює. В даному випадку вийшли досить прості циклічні умови, які з легкістю можна описати, однак, щоб зрозуміти перевагу кінцевого автомата, необхідно змоделювати більш реальну ситуацію, адже часто поведінка ворогів куди складніше. Гравець може втекти від ворога в момент битви, таким чином ворог повинен почати його пошук, якщо по закінченню деякого часу гравець не буде виявлений – перейти до патрулювання. Процес патрулювання передбачає пересування по заданій траєкторії, але можуть бути контрольні точки, де ворог переходить до стану бездіяльності. Ворог може мати малу кількість здоров'я після битви, тоді при виявленні гравця зі стану патрулювання і бездіяльності необхідно перейти до втечі.

Станів і умов може бути велика кількість, через що використання конструкції if-else призведе до ускладнення читання коду, тому розробники

і використовують кінцевий автомат з відображенням станів і переходів між ними (рисунок 1.5).

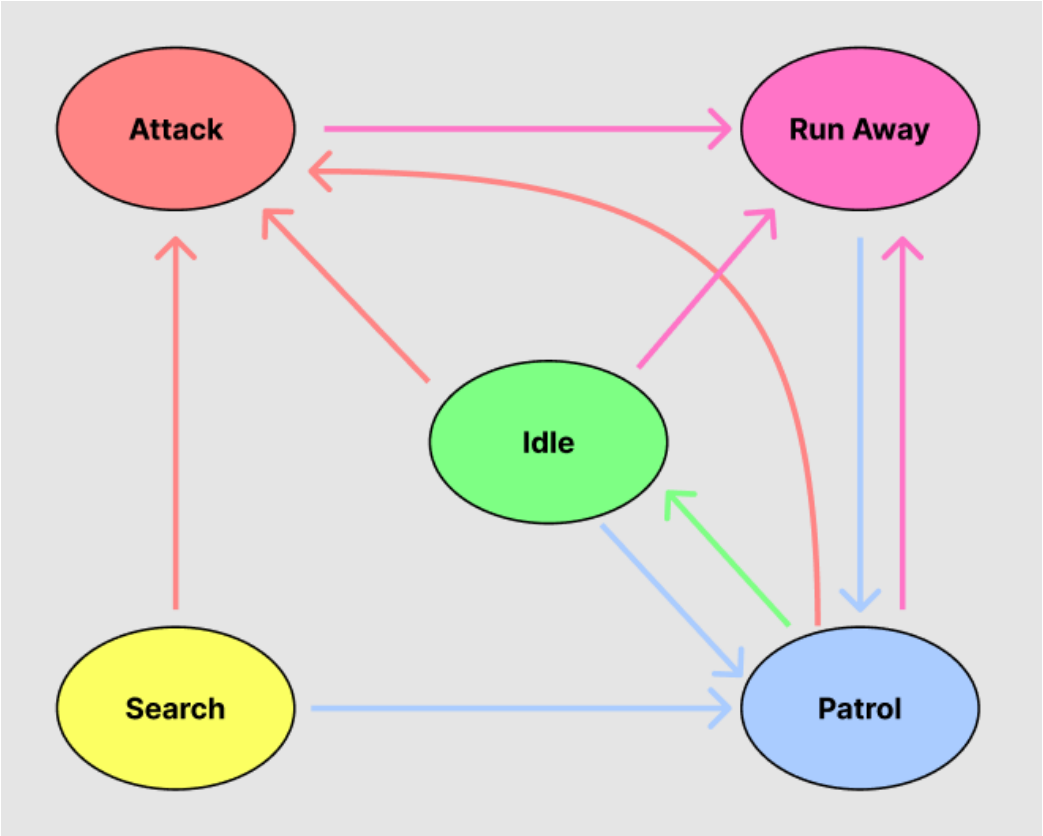

Рисунок 1.5 – Відображення схеми скінченного автомату

Якщо окремих станів велика кількість, то деякі схожі виносять і об'єднують в групу, таким чином позбавляючись від однотипних умов переходу. Кожна така група є теж кінцевим автоматом і в такому випадку ми маємо набір кінцевих автоматів з можливими переходами від одного до іншого – це називається ієрархічний кінцевий автомат. Тоді ми вже можемо переходити від одного автомата к іншому, в кожному з яких своя логіка вибору стану. У деяких іграх може використовуватися дерево прийняття рішень, проте вони складні при великій структурі через свою варіативності створення, тому їх доречно добавляти для невеличких задач.

1.3 Пошук шляху

Пошук шляху, мабуть, можна назвати найчастішим способом застосування алгоритмів для створення штучного інтелекту в сфері ігор[2]. Практично в будь-якій грі завжди є потреба знайти шлях від точки А до точки Б – переслідування ворогом персонажа, переміщення агентів по карті, відображення траєкторії руху для гравця. Все просто, коли цей шлях лінійний і рух відбувається по прямій, без перешкод на шляху. Однак, такі ідеальні умови зустрічаються хіба що у відкритому полі без ігрових об'єктів. Популярними алгоритмами є:

- алгоритм Дейкстри;
- алгоритм пошуку A*;
- хвильовий алгоритм;
- навігаційна сітка (navmesh).

Алгоритм Дейкстри можна назвати одним з найпростіших, його завдання – пошук оптимального шляху в графі. Однак, саме завдання є дуже поширеною, тому і застосування такого алгоритму виправдано. Працює алгоритм на орієнтованих графах (графи, у яких ребрам присвоєно напрямок).

$$G = (V, E), \tag{1.1}$$

де V – множина вершин графа;

E – множина ребер графа.

Для кожного ребра вказано невід'ємна вага і вершина, з якої здійснюється пошук оптимальних шляхів. Алгоритм може знайти найкоротший шлях між вершинами якщо існує хоча б один шлях, що зв'язує зазначені вершини, в іншому випадку алгоритм поверне нескінченно у вигляді відстані у вигляді відстані між вершинами. Розглянемо алгоритм роботи на зображеному графі (рисунок 1.6)

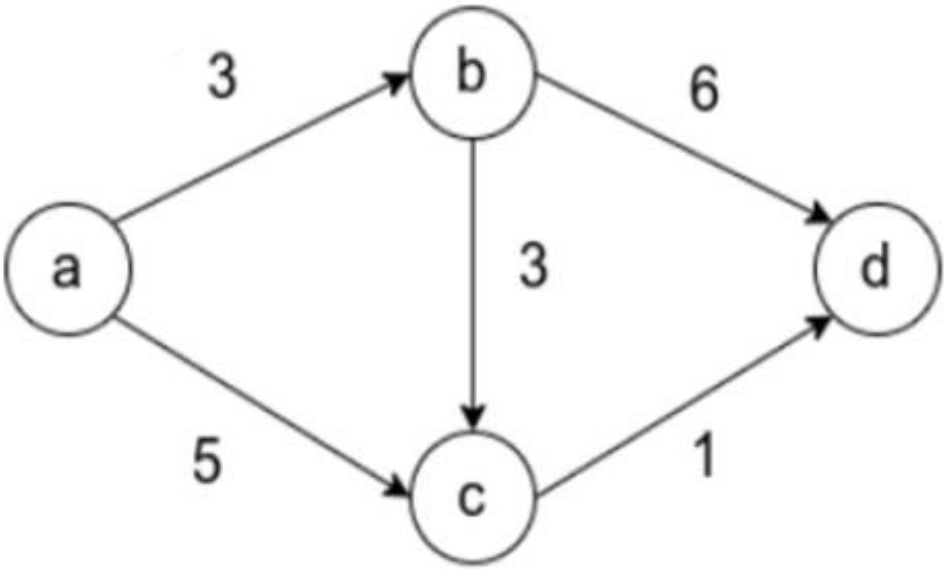

Рисунок 1.6 – Орієнтований граф з чотирма вершинами

Припустимо, нам потрібно знайти оптимальний шлях від «a» до «d». Кожній вершині вказується мінімальна відома відстань до стартової вершини, мітка самої вершини «a» дорівнює 0, а до решти вершин вказується нескінченність, щоб позначити, що відстань поки що невідома. На першому кроці вершина «a» має мінімальну мітку, її сусідами є «c» і «b». Відвідуємо вершину «b» і вказуємо їй мітку 3, так як вона менше поточної нескінченності, після чого відвідуємо вершину «c» і вказуємо їй мітку 5, так як вона теж менше поточної нескінченності. Бачимо, що на цьому всі сусіди вершини «a» перевірені, вершина вважається відвіданою, відзначаємо її.

Другим кроком необхідно вибрати знайти нову найближчу вершину, ми маємо «b» з міткою відстані 3 і вершину «c» з відстанню 5, вибираємо першу (рисунок 1.7).

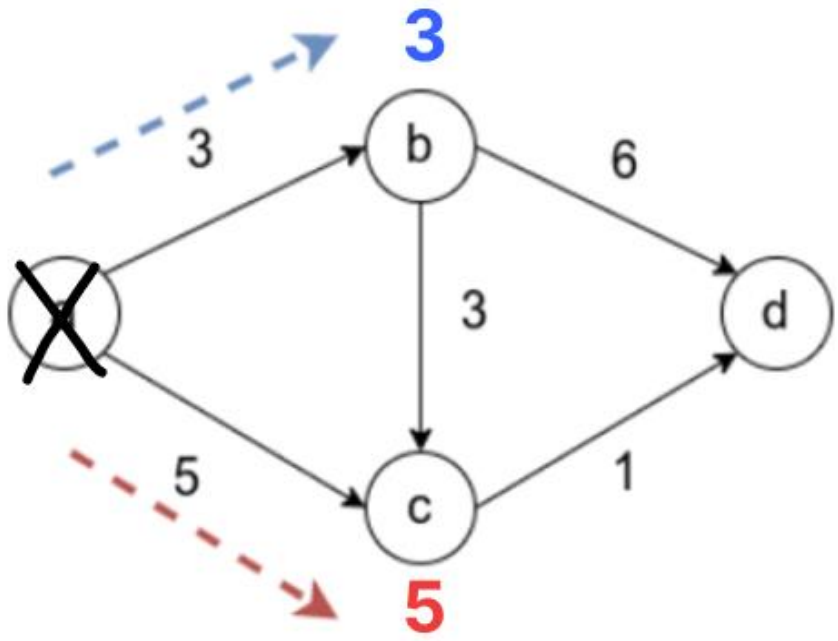

Рисунок 1.7 – Другий крок алгоритму Дейкстри

Сусідами вершини «b» є «a», «c» и «d». Першу вершину пропускаємо, так як вона відзначена відвіданої і в її бік не направлено ребро. Між «c» і «d» вибираємо першу, так як у неї стоїть мітка 5, що менше нескінченності. Якщо йти в вершину «c» через «b», то відстань шляху отримаємо 6, так як це сума 3 + 3, проте поточна мітка «c» дорівнює 5, а 5 менше 6, отже мітка залишається колишньою. Вибираємо наступну сусідню вершину – «d», відстань до неї є сумою відстаней між «a», «b» і «b», «d», тобто 3 + 6 = 9, а якщо 9 більше нескінченності, то перезаписуємо вершині «d» нову мітку. Переконуємося, що всі сусідні вершини перевірені і відзначаємо поточну «b» як відвідану.

На третьому етапі алгоритм знову вибирає невідвідану вершину з мінімальною міткою. З доступних «c» і «d» вибираємо першу, тому що її мітка 5 менша від мітки 9. Сусідами вершини «c» є «a», «b» і «d», однак перші дві вже відвідані і до них не направлені ребра , тому що з тих, що залишилися, вибираємо вершину «d» (рисунок 1.8).

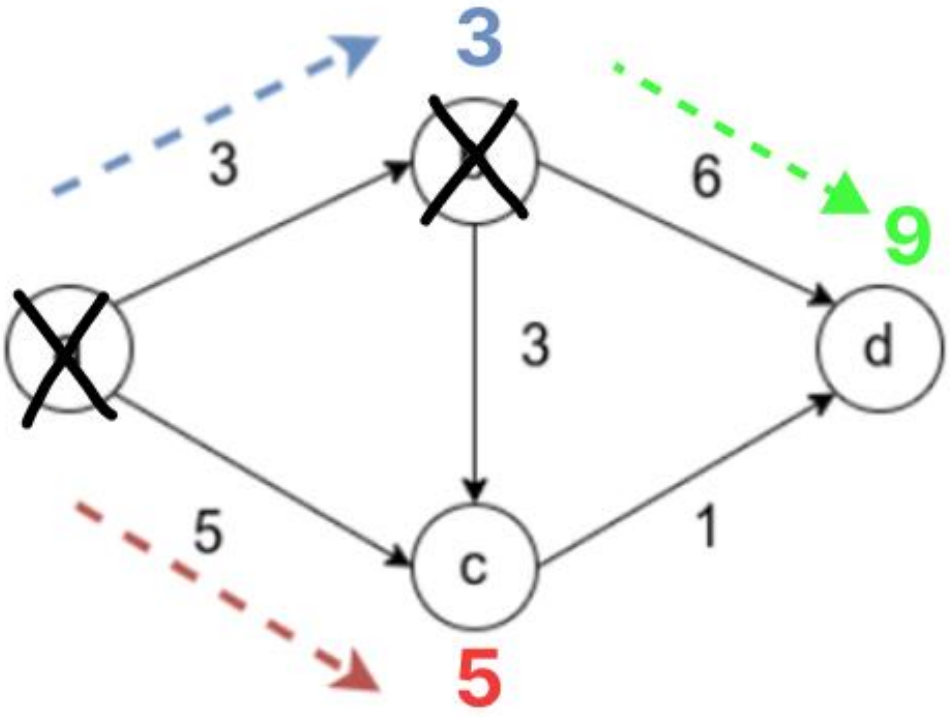

Рисунок 1.8 – Третій крок алгоритму Дейкстри

Якщо йти у вершину «d» через «c», то відстань дорівнює сумі «a», «c» і «c», «d», тобто 5 + 1, що дорівнює 6. Бачимо, що відстань 6 менша за поточну 9, значить перезаписуємо значення. Вершину «c» позначаємо як відвідану.

На останньому кроці у нас залишається єдина невідвідана вершина – «d», однак у цієї вершини всі ребра є вхідними, тому автоматично відзначаємо її як відвідану і алгоритм на цьому закінчує свою роботу. В результаті виконання алгоритму ми маємо мітки над кожною вершиною з відстанню до неї, якщо ця відстань позначена нескінченністю, отже, до вершини неможливо дістатися. Якщо в процесі кроків алгоритму записувати в масив історію пересування, то можна не тільки вивести найкращу відстань з вершини «a» до вершини «d», яка дорівнює 6, але й сам шлях – з «a» до «c», далі з « c» у «d» (рисунок 1.9).

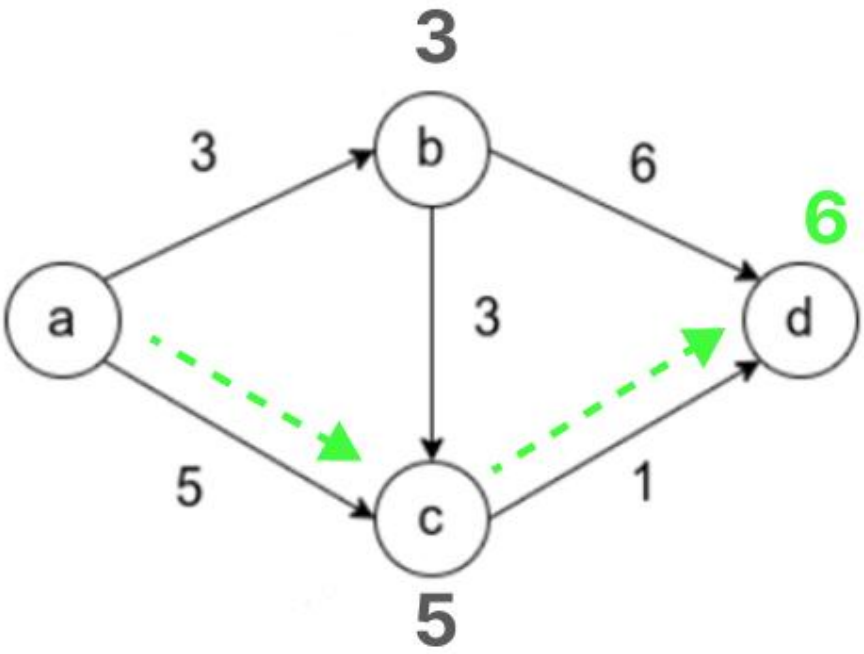

Рисунок 1.9 – Результат роботи алгоритму Дейкстри

Як підсумок, алгоритм дуже зрозумілий у використанні і неймовірно простий в написанні для розробників. Область застосування досить велика і в реальному житті, і в комп'ютерних/мобільних іграх. Якщо у вас в грі є невелике містечко з вулицями, то перехрестя можуть бути вершинами графа, а самі вулиці ребрами. В якості відстаней між вершинами дуже легко використовувати реальну довжину відрізка дороги. Таким чином, можливо з легкістю зробити штучний інтелект у вигляді таксі, яке везе пасажира по місту, або ж можна зробити навігацію в карті для гравця. Хоч ребра між вершинами є прямими відрізками, в грі це може бути звивиста дорога, спуск річкою і все, що завгодно, важливо лише вірно розставити ціну і тоді можна навіть проектувати сценарії проходження рівня від «легкого» шляху до «складного». Однак важливим пунктом варто виділити, що даний алгоритм не призначений для пошуку шляху в постійно мінливому ігровому світі, іншими словами, ви не зможете реалізувати обхід перешкод, які можуть зустрітись у агента на шляху.

Алгоритм А * за фактом є розширеним алгоритмом Дейкстри, який непоганий в пошуку найкоротшого шляху, проте у нього йде час на прорахунок всіх напрямків. У свою чергу алгоритм А * використовує в собі жадібний пошук і алгоритм Дейкстри, тобто А* враховує повну відстань від початку і оцінену відстань до мети, завдяки чому він є точним і досить швидким алгоритмом. На рисунку 1.10 можна побачити, як Дейкстри виконує зайві дослідження, в той час як А* уникає їх, так як вони знаходяться в протилежному напрямку від заданої мети. В результаті алгоритм А* витратив в 3 рази менше операцій.

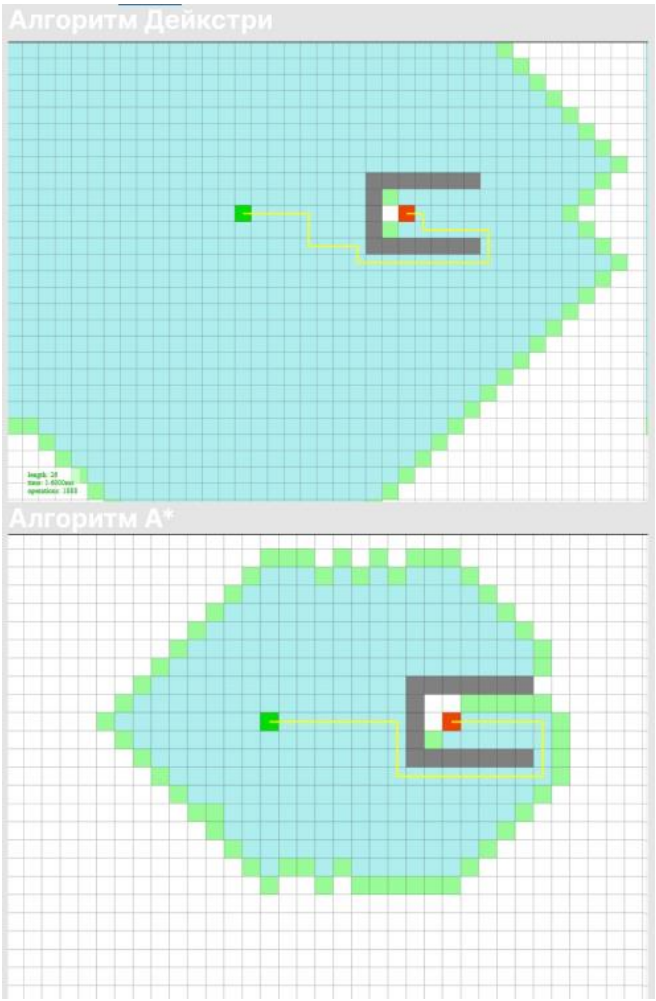

Рисунок 1.10 – Порівняння алгоритмів Дейкстри та А*

Ще одним простим алгоритмом є хвильовий алгоритм, він же алгоритм Лі. Виходячи з назви, пошук відбувається за принципом хвилі, тобто з стартової точки знаходяться, або околиці фон Неймана – сусідні клітини, які мають спільну сторону з обраної клітиною, або околиці Мура – сусідні клітини, які мають спільну вершини з обраної клітиною, таким чином ми або дозволяємо, або забороняємо діагональні переміщення.

Кожен крок алгоритму – це знаходження сусідів і запис в них числа, яке дорівнює кількості пройдених кроків від стартової позиції. З кожної нової клітини повторюються аналогічні дії, поки не буде виявлена фінішна. За підсумком найкоротший шлях будується в зворотному порядку від фінішної точки за мінімальними значеннями у клітинах (рисунок 1.11).

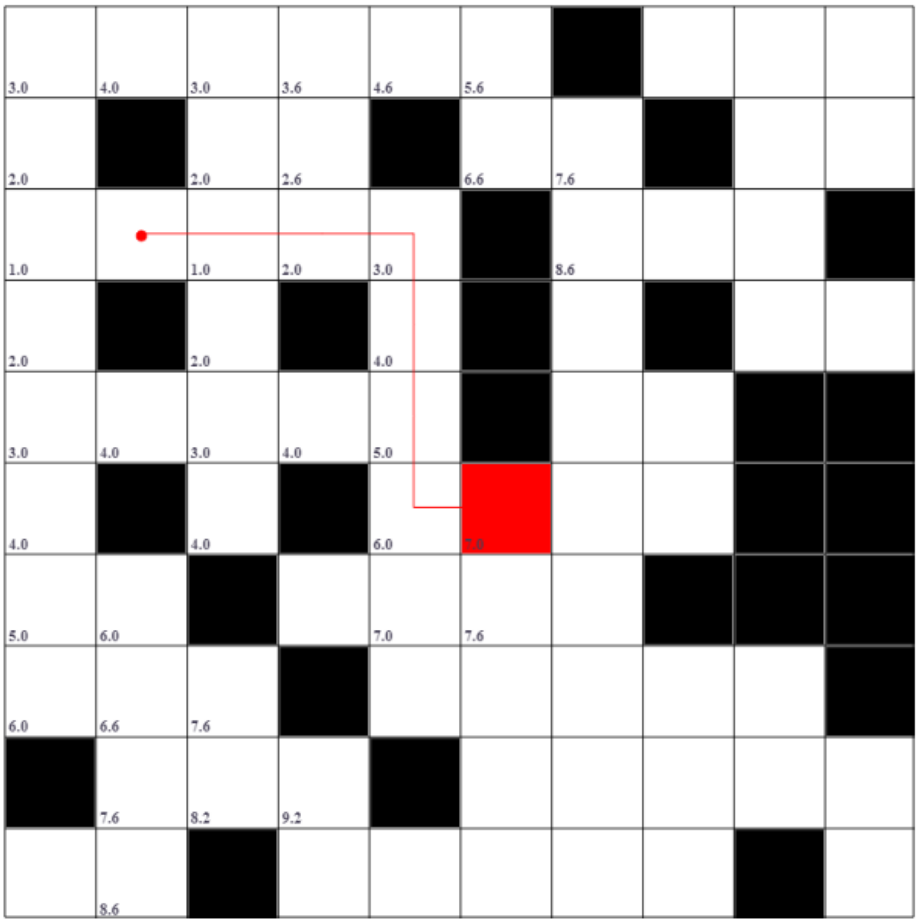

Рисунок 1.11 – Результат роботи хвильового алгоритму

1.4 Постановка задачі

Метою кваліфікаційної роботи є вивчення алгоритмів штучного інтелекту, їх наочне застосування в іграх, проектування і розробка програми під мобільні платформи Android і IOS, використовуючи програмне забезпечення Unity для створення 3D ігор і Construct 3 для створення 2D ігор.

Основними завданнями є:

- вивчення технічної літератури по створенню і способу застосування алгоритмів прийняття рішень і пошуку шляху;
- розробка мобільної розважальної гіперактивності із застосуванням штучного інтелекту;
- створення інтелектуальних штучних агентів, які будуть імітувати поведінку покупців в магазині;
- отримання знань з практичного використання програм для розробки мобільних ігор;
- публікація готових продуктів в Google Play і App Store.

Після успішного виконання всіх поставлених цілей, необхідно знайти навички самостійної розробки мобільних додатків, використання в них штучного інтелекту і особливостей застосування відповідних алгоритмів, після чого структурно викласти отриману інформацію в своїй роботі.

2 ПРОЕКТУВАННЯ ДОДАТКУ

2.1 Огляд програмного забезпечення для розробки Unity

Раніше розробникам доводилося писати ігри практично машинним кодом, але в наш час існують спеціальні програми, які значно спрощують і прискорюють процес створення. Під час своєї бакалаврської роботи мені довелося познайомитися з більшістю основних і популярних рушіїв для створення ігор, тому я заздалегідь знав, що мій вибір стане Unity, який я буду використовувати для створення кінцевих продуктів і Construct 3, який зручний в швидкому прототипировании і перевірці концепцій. Перш ніж перейти до безпосередньої розробки основних додатків, необхідно познайомитися з кожним програмним забезпеченням і навчитися реалізовувати описані в аналізі алгоритми.

Unity – це сучасне крос-платформне середовище для розробки 2D і 3D мобільних ігор, комп'ютерних і консольних проектів з зрозумілим і зручним інтерфейсом[3]. Підтримується написання скриптів на мові програмування C#[4].

Основні переваги даного рушія:

- безкоштовний тариф для приватних осіб з невеликим доходом;
- бібліотека ассетів, де опубліковані платні і безкоштовні матеріали для ігор, від звуків до повноцінних ігрових шаблонів;
- створення скриптів на одному з найпопулярніших мов програмування – C#;
- достаток навчальних матеріалів, як платних, так і безкоштовних від базового до просунутого рівня;
- багатомільйонна спільнота, так як Unity можна назвати номером один для розробки мобільних ігор;
- підтримка експорту під всі сучасні операційні системи.

З мінусів варто відзначити продуктивність і оптимізацію, досить важко буде розробити великий AAA проект, тому розробники часто для таких цілей використовують або Unreal Engine, або великі компанії створюють власні рушії під проекти.

Unity проект складається з набору сцен, які включають в себе елементи ігрового світу. Кожна сцена завантажується окремо і відображається у вікні сцени. Для перегляду гри існує вікно Game, від час розробки ми маємо змогу запустити проект и побачити, як він буде відображатись на пристрої, також є можливість зміни розміру екрана. На самих сценах знаходяться об'єкти, список яких розміщений у вікні ієрархії зліва. Об'єкти складаються з набору компонентів, з якими можна взаємодіяти. Для додавання або видалення компонентів об'єкта передбачено вікно «Inspector». Доступно вікно «Console» для можливості налагодження програми. Інтерфейс програми відображений на рисунку 2.1.

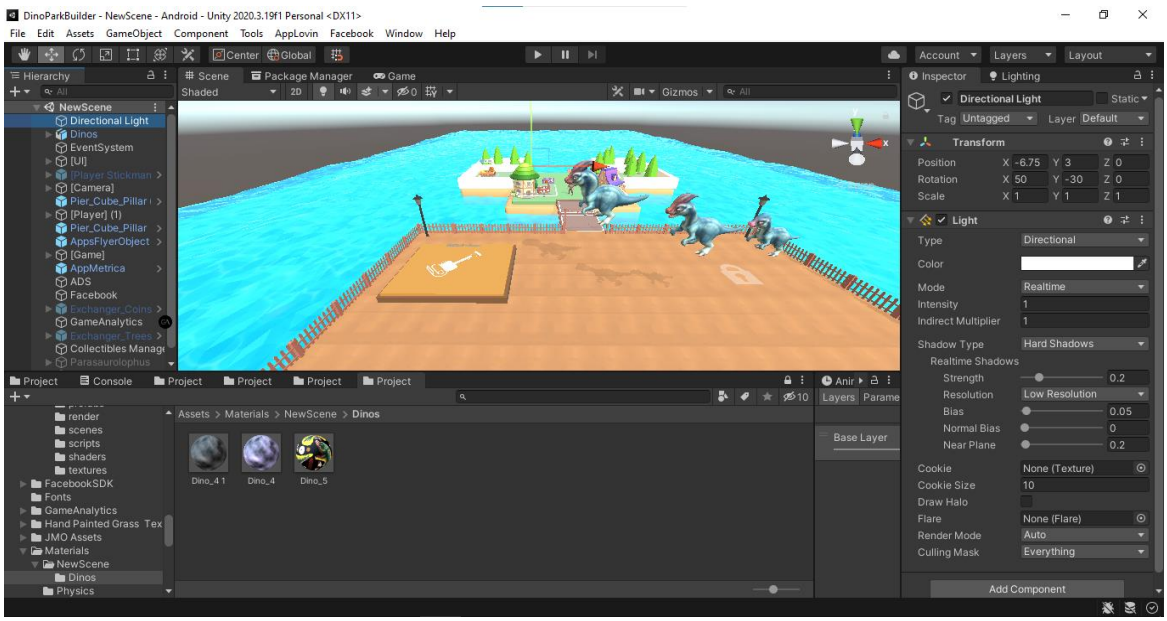

Рисунок 2.1 – Інтерфейс рушія Unity

Взаємодіяти з об'єктами дуже зручно, досить просто додати нові компоненти у вікні «Inspector». Єдиним обов'язковим компонентом є

«Transform», він містить інформацію про положення і поворот об'єкта в світових координатах щодо батьківського, а також значення його розміру. Об'єкт лише з одним компонентом «Transform» є пустушкою. Можна створювати складні групи об'єктів вкладаючи їх один в одного, тоді зовнішній об'єкт буде вважатися батьківським, а внутрішній – дочірнім. Для візуального відображення на об'єкті присутній компонент «Mesh Renderer», він включає в себе матеріал, яким потрібно відмалювати об'єкт і працює разом з компонентом «Mesh Filter» в якому вказується 3D модель самого об'єкта. Об'єкт може мати різні типи колайдерів – це спеціальні маски, які описують як саме варто обробляти зіткнення з об'єктом. Наприклад, компонент «Box Collider» говорить про те, що об'єкт має кубічну форму зіткнення з розмірами, зазначеними в полях «Size» і центральною позицією, зазначеної в полі «Center» (рисунок 2.2).

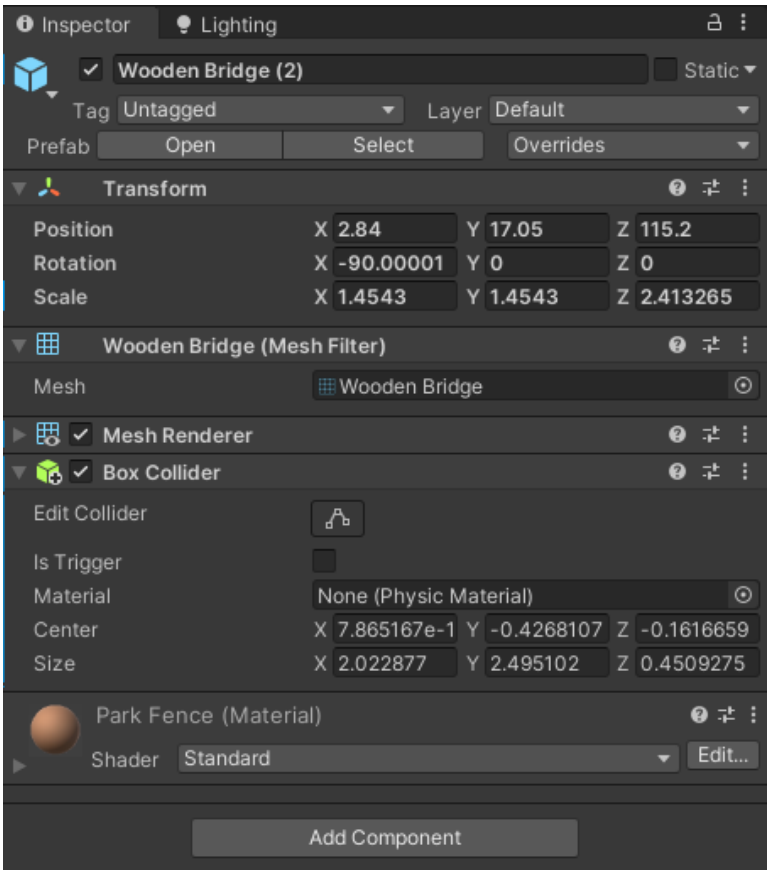

Рисунок 2.2 – Компоненти об’єкту bridge

2.2 Огляд програмного забезпечення для розробки Construct 3

Construct 3 – це програмне забезпечення, яке дозволяє створювати кроссплатформенні 2D ігри з використанням мови програмування JavaScript, або користуючись вбудованою блоковою системою, коли всі умови і дії можна просто перемістити мишкою в список подій[5].

Основні переваги даного рушія:

- легковажність як самої програми, так і експортованих проектів;
- можливість експортувати під більшість сучасних платформ, в тому числі під HTML5;
- віддалений передпоказ відразу на пристрій;
- простота освоєння навіть без колишніх навичок розробки.

З мінусів можна виділити:

- щорічна оплата підписки 100$;
- обмеження частоти FPS в 60 кадрів;
- обмеження можливої ваги програми в 150 мегабайт,
- обмежений набір сторонніх SDK.

Проект в Construct 3 складається зі сцен, вони ж «Layouts». Спочатку сцена являє собою білий аркуш, розмір якого можна налаштувати. Сцена розділяється на шари – таким чином ми можемо регулювати глибину в двовимірній грі, вказуючи порядок відтворення або налаштовувати паралакс ефект, вказуючи швидкість зміщення шару. Кожній сцені відповідає якийсь список подій, він же «Event Sheet», який необхідний для функціонування гри. У списку подій створюються умови і вказуються відповідні їм дії, сюди ж додаються глобальні змінні і власні функції. Весь список обробляється системою зі швидкістю 60 разів на секунду в порядку зверху вниз. Властивості проекту, сцен і об'єктів знаходяться зліва, а сам список сцен, об'єктів і шарів – справа. У центрі розташоване вікно сцени з можливістю перемикання на вікно списку подій (рисунок 2.3).

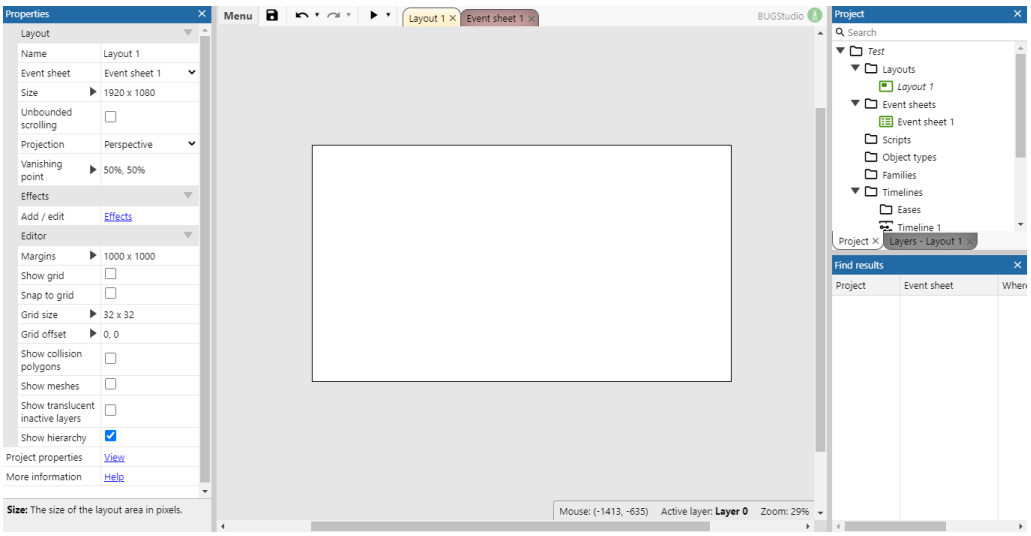

Рисунок 2.3 – Інтерфейс рушія Construct 3

Об'єкти заздалегідь розділені на типи, розробниками програмного забезпечення. Для додавання нового об’єкту потрібно натиснути правою кнопкою миші на макеті та вибрати із списку необхідний тип об’єкту. Перелік частини об'єктів відображений на рисунку 2.4.

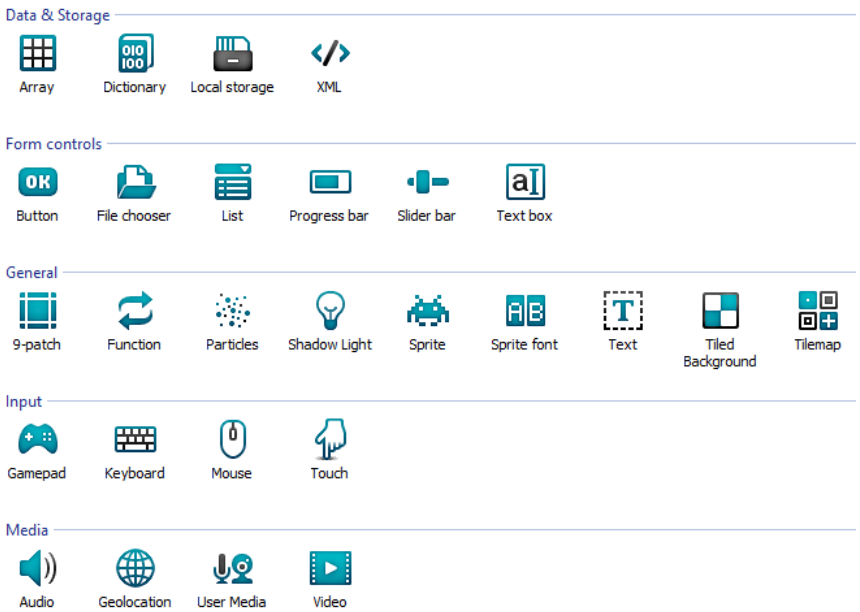

Рисунок 2.4 – Список можливих об’єктів у Construct 3

Кожен раз в подіях об'єкти фільтруються і вибираються відповідні за заданими умовами і дії виконуються лише для відібраних об'єктів. Наприклад, якщо у нас є умова «агент стикається з монетою», то програма вибирає з усіх копій лише 2 конкретних – одного агента і одну монету. Справа в події знаходяться дві дії – видалити об'єкт монети і додати одиницю в лічильник монет агента (рисунок 2.5), завдяки фільтрації, буде видалена обрана монета і лічильник збільшиться лише у одного агента, який підібрав монету, а не у всіх об'єктах.

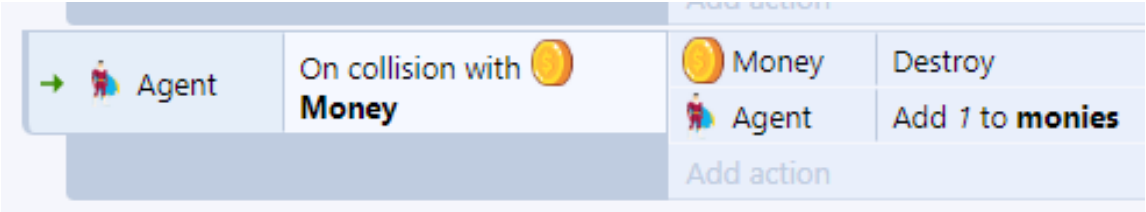

Рисунок 2.5 – Фільтрація об’єктів у подіях

Події можуть не мати фільтрів, в такому випадку дії спрацьовують для всіх об'єктів. Наприклад, на рисунку 2.6, подія має умову старту рівня і дії нищення монети та збільшення лічильника агента. Зліва в умові немає фільтруючих механізмів, отже при запуску рівня системою будуть видалені всі екземпляри об'єкта «Money», а усім об’єктам «Agent» буде додано одиницю до змінної «monies».

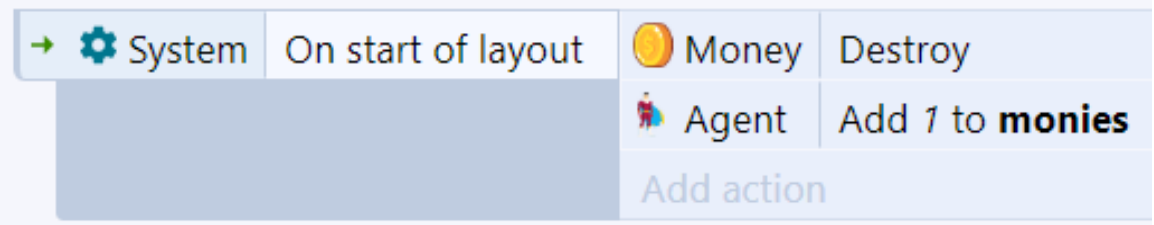

Рисунок 2.6 – Подія без фільтрації об’єктів

2.3 Хвильовий алгоритм у Construct 3

В ході аналізу було виявлено, що одне з найпопулярніших завдань штучного ігрового агента є рух від точки до точки, обходячи перешкоди. В якості ознайомлення можливостей Construct 3 і для отримання навичок роботи з алгоритмами розглянемо процес проектування і створення хвильового алгоритму[6].

Для початку необхідно створити новий проект і вказати розміри рівня 1000*1000 пікселів. Створити 4 об'єкти «Sprite», перший назвемо «Node» – це буде наш вузол, куди персонаж може дійти, другий назвемо «Wall» – це стіна, яку персонаж повинен уникати, третій об'єкт буде «FinishPoint» – це точка, до якої потрібно знайти шлях, а четвертим об'єктом буде наш персонаж. Розставимо об'єкти на карті (рисунок 2.7).

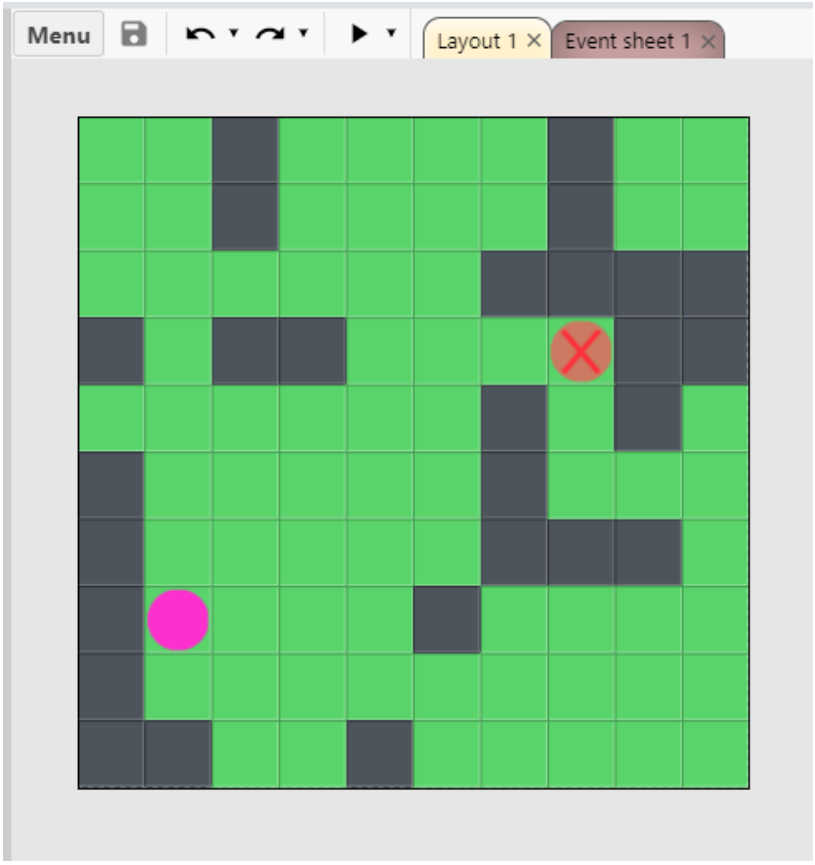

Рисунок 2.7 – Макет для хвильового алгоритму

Кожному вузлу додамо змінні «myStep», де будемо зберігати крок, на якому цей вузол був виявлений і «mainNode», яку будемо активувати, якщо даний вузол було виявлено останнім (рисунок 2.8).

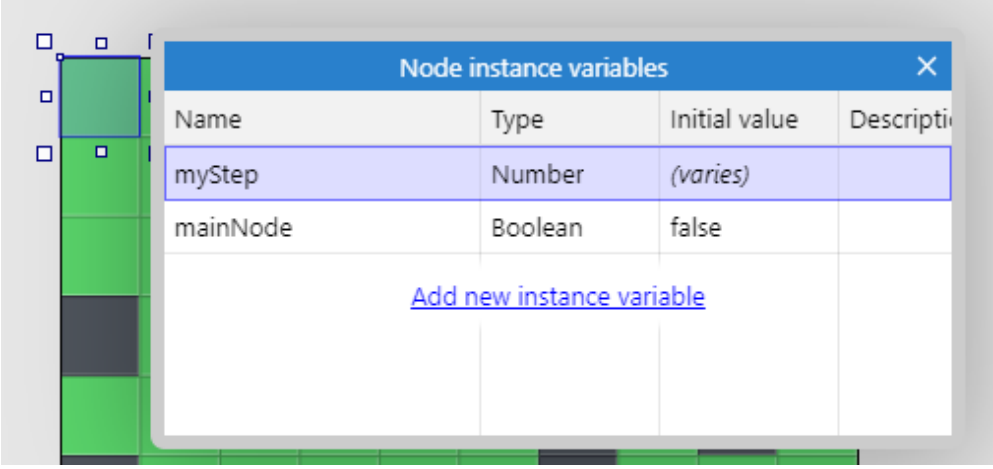

Рисунок 2.8 – Змінні для об’єкта «Node»

Створимо масив, в який записуватимемо позиції вузлів, коли виявимо шлях. Додамо текст, який будемо створювати при виявленні нового вузла, значення тексту дорівнюватиме кроку, на якому даний вузол був виявлений. Текст потрібен виключно для демонстраційних цілей. Для візуального відображення додамо ще 3 кольори в об'єкт вузол, вони відображатимуть стан – вузол пройдено, новий вузол, знайдений шлях. Додамо глобальні змінні «Step», у якій рахуватимемо кількість кроків від старту алгоритму і булеву змінну «PathFound», вона сигналізуватиме про необхідність припинення пошуку, у разі виявлення кінцевої точки. Напишемо функцію «resetWorld», яка буде служити для переключення змінних та об'єктів у робочі стани перед початком алгоритму – шлях не знайдений, кроків 0, масив точок порожній, вузли у стандартному стані, а вузол на якому знаходиться наш персонаж вказується як початковий, з якого починається пошук (рисунок 2.9).

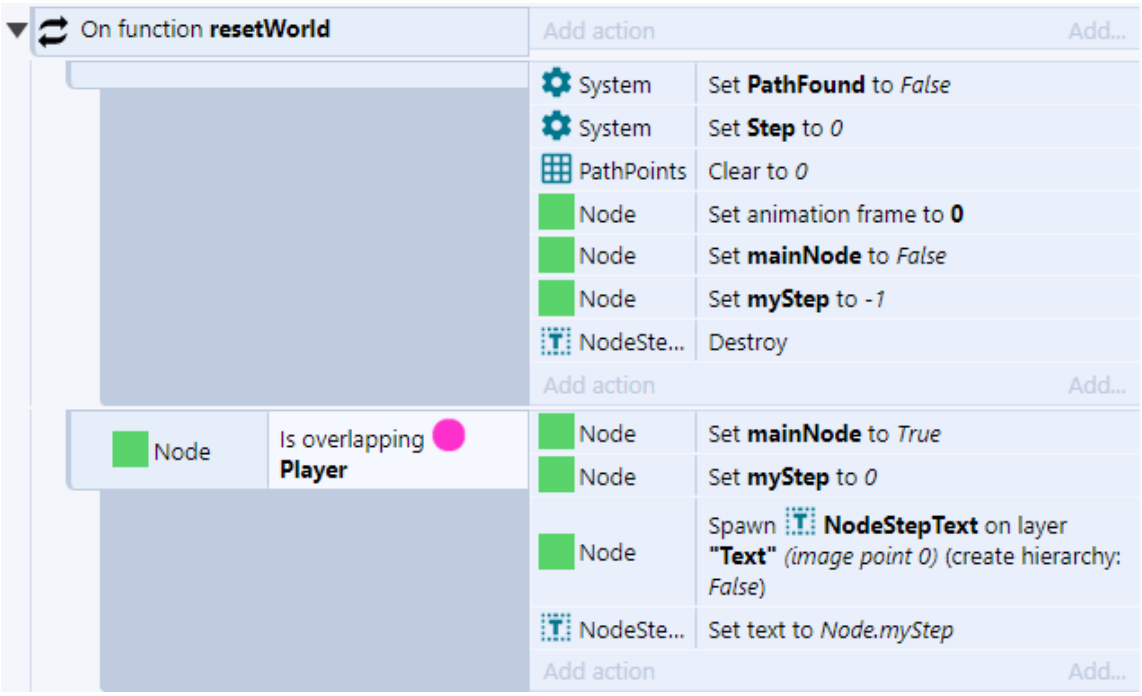

Рисунок 2.9 – Функція «resetWorld»

Алгоритм опишемо у функції «waveStep». Створимо об'єкт «Pointer», він буде необхідний для зручного відображення покажчика і легкого пошуку сусідніх вузлів. При вході в функцію будемо збільшувати глобальну змінну «Step» на 1, далі створимо цикл, в ньому перебираємо всі основні вузли, позначаємо їх як пройдені і шукаємо сусідні. Сусіднім вузлам записуємо в змінну екземпляра «myStep» глобальний крок, на якому вузол був знайдений, створюємо текст з відображенням цього значення і вказуємо логічної змінної «mainNode» істину, щоб при наступному кроці з цього вузла продовжити пошук. Всі нові вузли візуально відзначаємо більш світлим кольором, а пройдені більш темним. У разі виявлення фінішної точки зупиняємо цикл, відзначаємо глобальну логічну змінну «PathFound» як істину і переходимо до виконання функції для переміщення персонажа. Якщо по завершенню кроку фінішна точка не була знайдена, то викликаємо функцію «waveStep» повторно (рисунок 2.10).

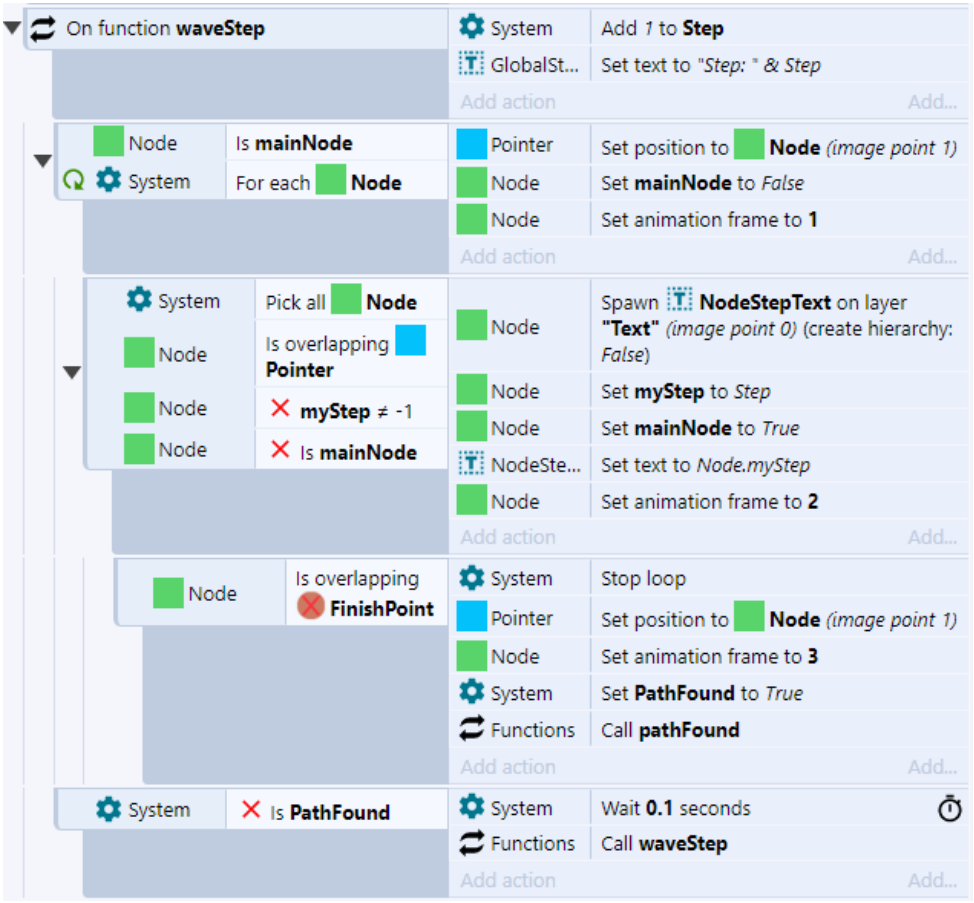

Рисунок 2.10 – Основна функція кроку хвильового алгоритму

При знаходженні фінішної точки нам залишається лише занести всі дані в масив позицій, для цього спочатку вказуємо розмірність шириною рівній кількості кроків і висотою 2, так як необхідно зберігати дані по двох осях. В останній індекс заносимо позицію фінішної точки, потім створюємо цикл з кількістю повторень рівним кількості пройдених кроків. У кожній ітерації циклу знаходимо найближчий до об'єкта «Pointer» вузол, значення кроку якого збігається з шуканим, якщо таких вузлів кілька, то вибираємо випадковий. Позицію знайденого вузла заносимо в масив, зміщуємо покажчик і візуально вузол виділяємо іншим кольором, якщо крок був останнім – активуємо початок руху персонажа. Ці кроки повністю описують вміст функції «pathFound» зображеної на рисунку 2.11.

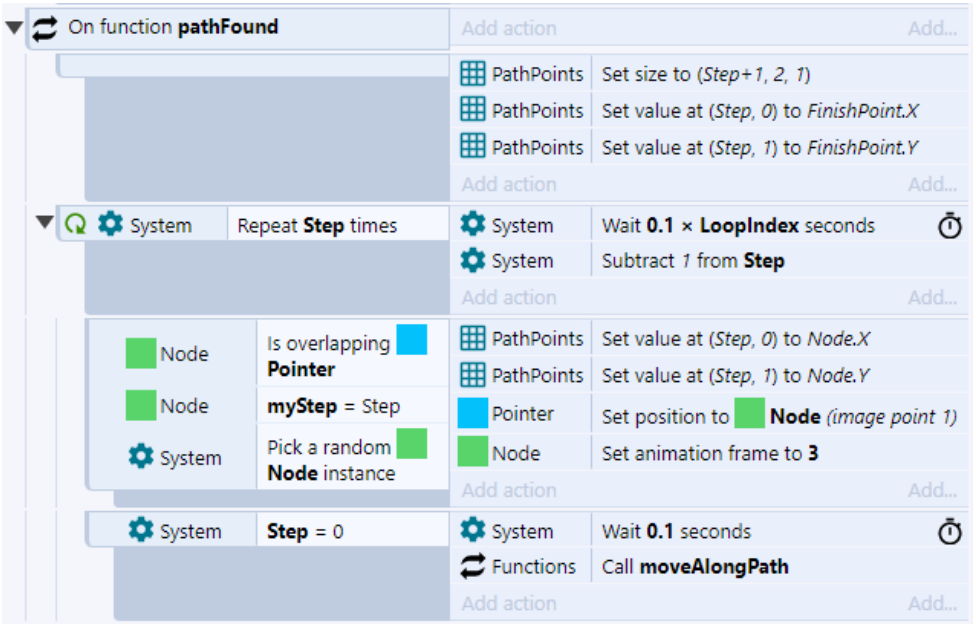

Рисунок 2.11 – Тіло функції «pathFound»

Підсумковий результат є повноцінним алгоритмом, який повністю справляється з поставленим завданням. Має можливість пошуку шляху з урахуванням діагональних переміщень і без них (рисунок 2.12 та рисунок 2.13).

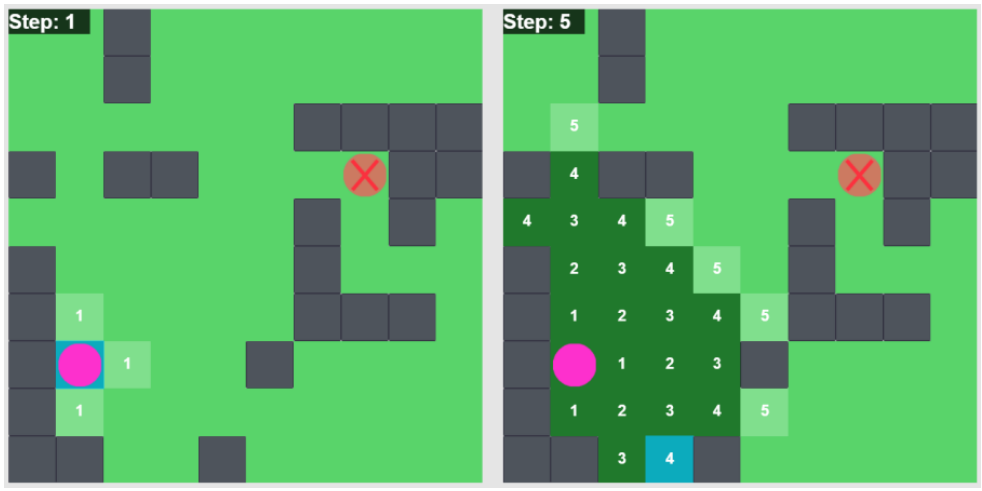

Рисунок 2.12 – Кроки 1 та 5 хвильового алгоритму

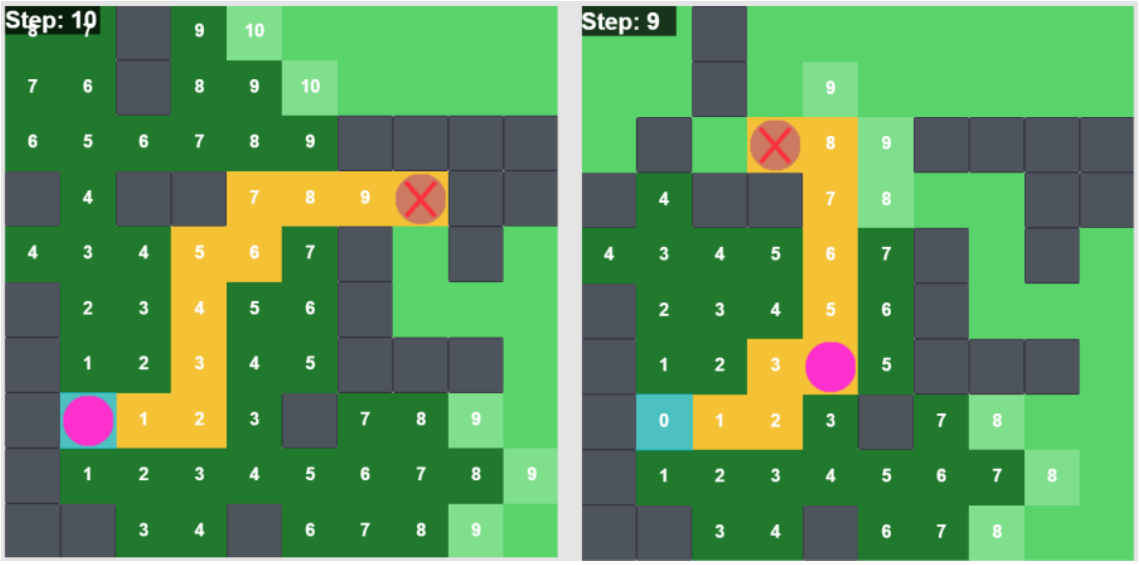

Рисунок 2.13 – Результати роботи хвильового алгоритму

Додатково була створена можливість вибору кінцевої точки натисканням клавіші мишки, додані анімації плавного відображення пошуку і переміщення персонажа, застосовано згладжування ключових точок. Можна додати криві Безьє в спробах зробити рух ще більш акуратним, наприклад, для квадратичної кривої першою опорною точкою брати поточний елемент масиву -1, другою опорною точкою буде поточною елемент +1, а третій – середнє значення елементів. Таким чином вийде згладити рух при поворотах на 90 градусів. Можна відстежувати патерни зміни напрямків і замінювати позиції групами, проте переді мною стоїть завдання дослідити і зрозуміти принцип дії алгоритмів, а не розробити власний ідеальний з нуля. Хоч дані напрацювання цілком можна використовувати в іграх, де потрібно переміщення будь-яких об'єктів з можливістю обходу перешкод, причому не тільки двовимірних, але і 3D, якщо уявити третю вісь константою, але на ринку існує безліч вже готових добре оптимізованих і зручних рішень, тому часто простіше взяти написане і витратити час на більш унікальні для проекту механіки.

2.4 Використання NavMesh у Unity

NavMesh – це вбудована система навігації в Unity. Вона складається з декількох компонентів:

- navMesh, який за фактом є набором даних описують ігрову карту;
- navMeshAgent, який вказується на всіх агентів, яким потрібно зробити пересування;
- offmeshLink, який необхідний для вказівки особливих шляхів руху, наприклад підйомів і спусків;
- navMesh Obstacle, який вказується на всіх рухомих перешкодах, щоб агенти могли динамічно будувати шлях з можливістю обходу таких об'єктів.

Агенти navMesh описуються у вигляді циліндрів, а шлях у вигляді багатокутників. Для пошуку шляху всередині Unity використовується алгоритм A*. Щоб створити навігаційну сітку navMesh, необхідно запекти рівень. Для цього необхідно перейти у вкладку «Window – AI – Navigation». Відкриється вікно налаштування, в якій нас цікавить вкладка «Object» (рисунок 2.14).

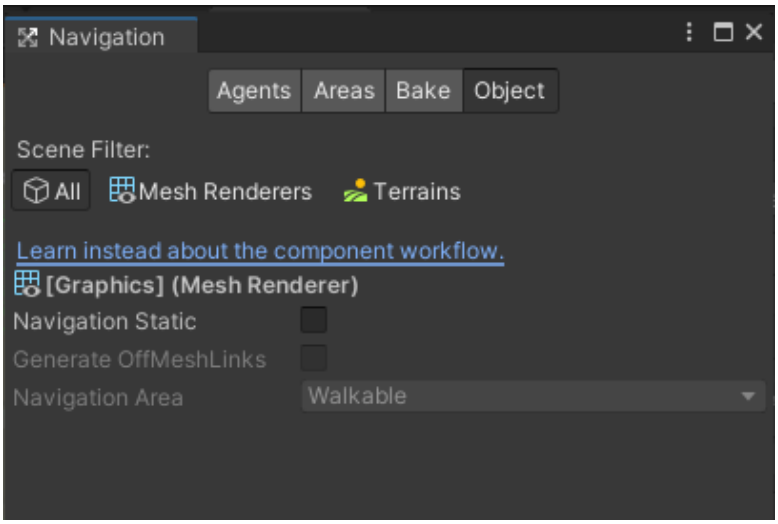

Рисунок 2.14 – Запікання рівня для створення NavMesh

Для коректного запікання рівня необхідно виставити настройки у вкладці «Bake». Тут вказується радіус і висота агента, щоб врахувати відстань від стін і від стелі для проходу під перешкодами. Вказується максимально можливий кут підйому і максимально можлива висота, на яку агент здатний зайти. Тут же вказується мінімальні розмір області, яка генерується, є можливість вказати точність результуючої сітки, проте з великою точністю сильно зростає навантаження, що позначається на продуктивності (рисунок 2.15).

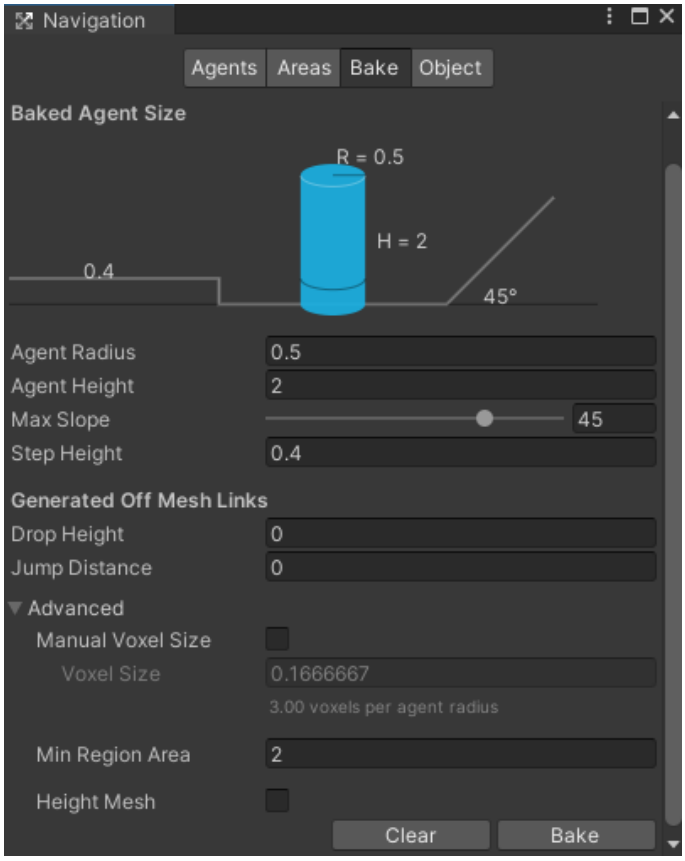

Рисунок 2.15 – Налаштування параметрів запікання

Після натискання кнопки «Bake» рівень буде запечений і навігаційні параметри будуть збережені в файл з таким же ім'ям, як і ім'я сцени. У вкладці «Scene» в редакторі буде відображатися запечений шлях (рисунок 2.16)

Рисунок 2.16 – Відображення запеченого шляху

У вкладці «Areas» є можливість вказати типи зон і вартість їх проходу. Вартість важлива в разі пошуку оптимального шляху. Самі зони дають можливість більш тонко налаштовувати навігаційну сітку, враховуючи різницю місцевості. Наприклад, ми можемо вказати зони «асфальт», «земля», «вода» і вказати вартості рівними 1, 2, 5 відповідно. Таким чином, в момент пошуку оптимального шляху, прохід через воду буде оцінений в 5 разів дорожче, можна вважати довшим, ніж прохід по асфальту.

Додатково зони дозволяють створювати окремі області для кожного агента. Наприклад, якщо у вас є машина, то вона не здатна переміщатися по воді, але добре їде по асфальту і може по дорозі, а катер здатний переміщатися лише по воді, суша не підходить, тому ми можемо вказати в першому випадку всі зони і виключити лише непрохідні, а для катера простіше вказати лише одну зону води для прорахунку навігації (рисунок 2.17).

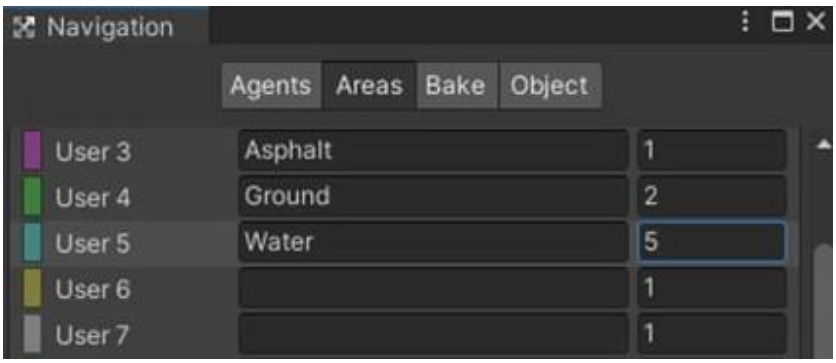

Рисунок 2.17 – Налаштування зон та їх вартостей

В кінці можна створити агента, якому додамо компонент NavMeshAgent (рисунок 2.18).

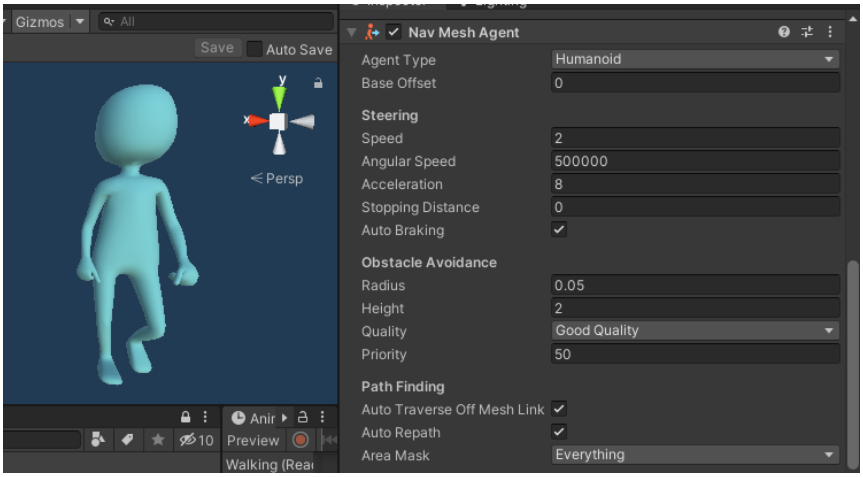

Рисунок 2.18 – Налаштування агенту navMesh

Даний компонент вказуватиме, що об'єкт є агентом і надає нам можливість налаштування радіуса, висоти, швидкості його руху і прискорення, відстань перед цільовою точкою для зупинки і можливість автоматичного уповільнення перед зупинкою. Винесені налаштування якості обходу перешкод, автоматичного повторного пошуку шляху, в разі не виявленні такого в перший раз і доступний список масок, де вказується які типи областей повинні враховуватися для даного агента при побудові шляху.

3 РОЗРОБКА ТА ОПИС ДОДАТКА

3.1 Створення проекту та імпорт асетів

Перед початком роботи необхідно виділити окрему папку для проекту на комп'ютері, відкрити програму «Unity Hub» і створити новий проект (рисунок 3.1).

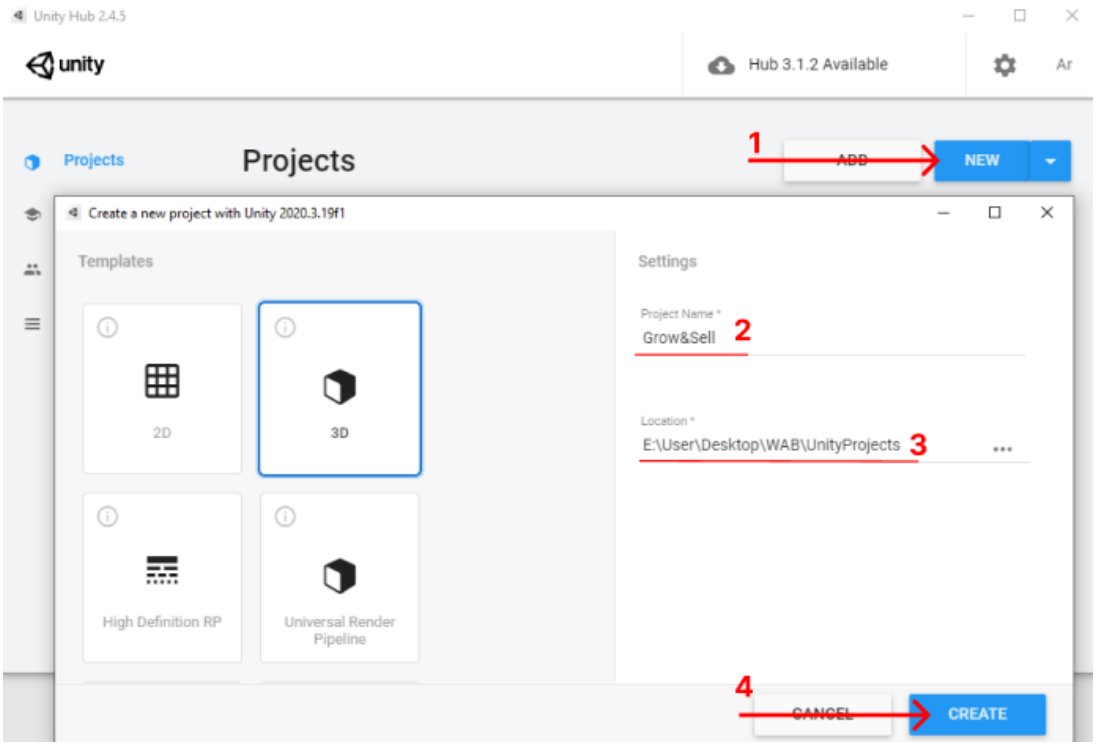

Рисунок 3.1 – Створення проекту Unity

Далі очікуємо кілька секунд або хвилин, поки проект створюється. Для початку роботи необхідно задати структуру папок, щоб було зручно орієнтуватися в проекті. Натискаємо правою клавішею миші у вікні «Project», далі вибираємо «Create – Folder» і створюємо такі папки:

- scripts, де будемо зберігати створені скрипти;
- models, де будемо розміщувати моделі;
- animations, в якій будуть анімації об'єктів;

- material, в ній будуть знаходиться матеріали об'єктів;
- prefabs, тут заготовки об'єктів;
- sprites, в яку завантажимо необхідні зображення;
- externalAssets, тут будуть знаходитися всі сторонні асети.

Завантажимо необхідні моделі персонажів, вітрин, квітів, об'єктів. Створимо примітив кілька примітивів кубів – це буде земля карти, підлогу і стіни магазину. Розставимо схематично всі об'єкти, додамо персонажа і налаштуємо камеру[7]. Підсумковий вид зображений на рисунку 3.2.

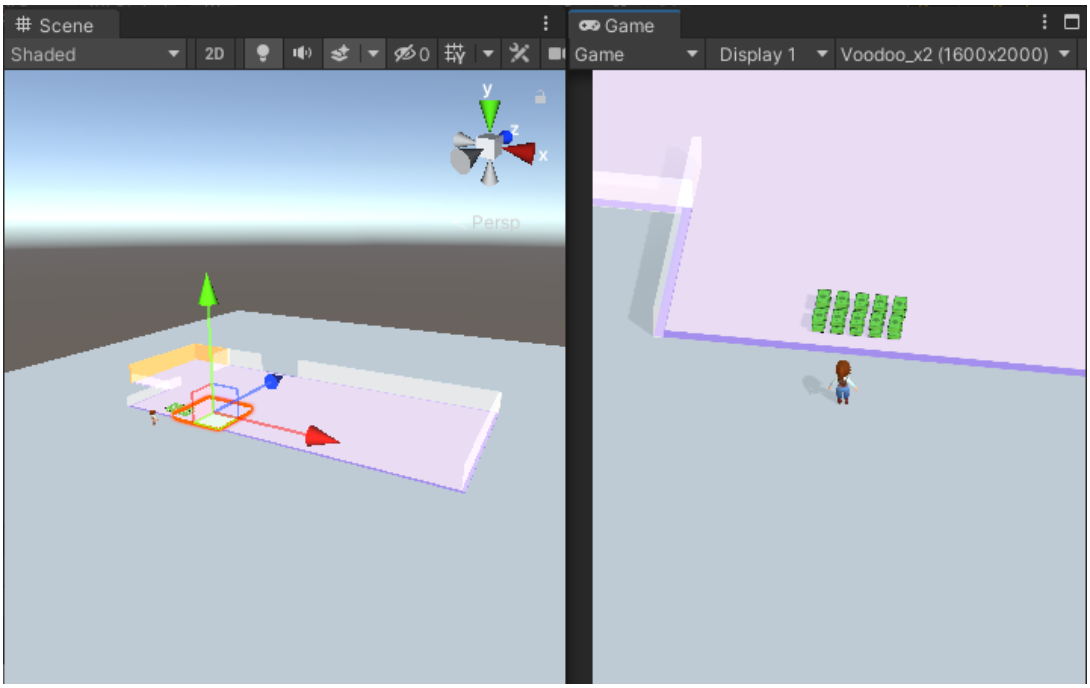

Рисунок 3.2 – Побудований рівень та відображення камери

Для обробки зіткнень додамо компонент «Box Collider» на землю і стіни. Персонажу додамо компоненти «Capsule Collider» і «Rigidbody», перший буде відповідати за маску колізії, а другий вкаже, що дане тіло є фізичним. Для запобігання перевертань зайдемо в пункт «Constraints» і заблокуємо повороти по осях «X» і «Z». В кінцевому підсумку набір компонентів буде виглядати як на рисунку 3.3.

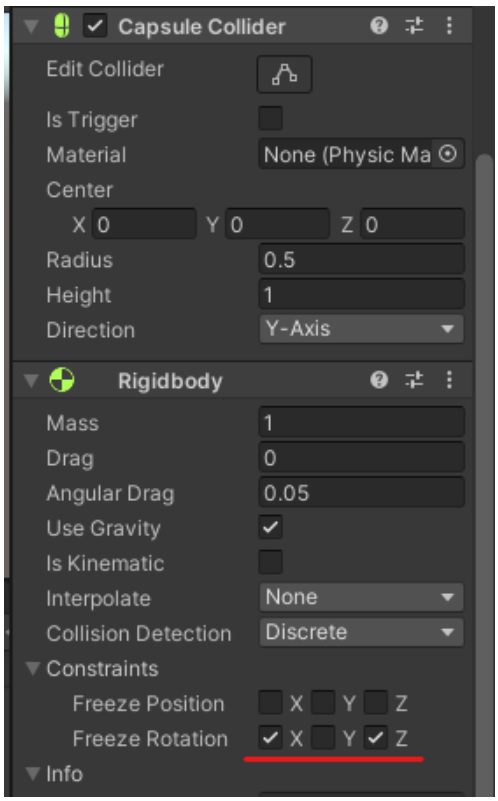

Рисунок 3.3 – Компоненти об’єкту персонаж

Наступним кроком імпортуємо асет «Joystick», він дозволить легко додати управління джойстиком в гру і не витрачати на це час. У папці «Scripts» створимо новий C# скрипт, назвемо його «PlayerController»[8]. Створимо 3 приватних поля, в першому будемо зберігати джойстик, у другому посилання на компонент «Rigidbody» і третій компонент буде вектором для зберігання початкового повороту. Додатково зробимо булеву змінну «_blockInput», яка буде блокувати управління і виведемо в публічному полі швидкість руху персонажа. Додамо метод «Awake», який системою Unity викликається перед запуском гри, в ньому отримаємо посилання на компонент фізичного тіла. Напишемо метод «Move», в ньому ми будемо отримувати значення з джойстика, які запишемо в якість нормалізованого вектора напрямку. Якщо вектор не дорівнює нулю, то зміщуємо і повертаємо персонажа (рисунок 3.4).

```
private void Move()
{
    var horizontal :float = _joystick.Horizontal();
    var vertical :float = _joystick.Vertical();

    var direction :Vector3 = new Vector3( x: horizontal,  y: 0,  z: vertical).normalized;


    if (direction.magnitude > 0)
    {
        var targetAngle :float = Mathf.Atan2( y: direction.x,  x: direction.z) * Mathf.Rad2Deg;
        transform.localEulerAngles = _initialRotation + new Vector3( x: 0f,  y: targetAngle,  z: 0f);

        var positionToMove :Vector3 = _rigidbody.position +
                              transform.forward * (playerSpeed * Time.fixedDeltaTime);
        _rigidbody.MovePosition(positionToMove);
    }
}
```

Рисунок 3.4 – Функція переміщення персонажа

Створимо об'єкт «Canvas» зліва в ієрархії, додамо в нього префаб «JoystickController», який був в імпортованому ассеті, він необхідний для управління. Виділимо об'єкт «Player» і натиснемо «Add Component», зі списку компонентів виберемо створений скрипт «PlayerController». Особливості підключення гуманоїдних анімацій пропущені і не будуть описані в рамках даної роботи. При запуску проекту маємо оточення і персонажа, яким можемо управляти переміщаючи джойстик (рисунок 3.5)..

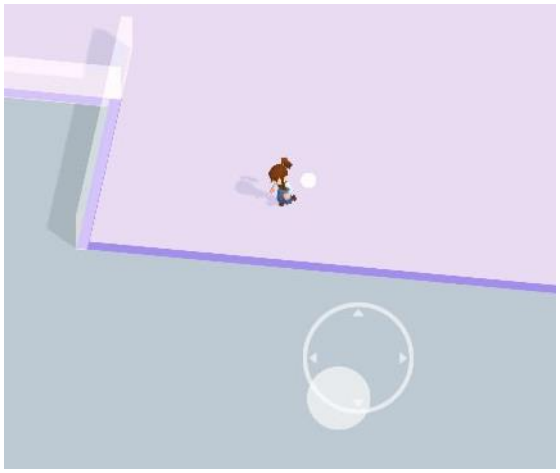

Рисунок 3.5 – Можливість управління джойстиком

Ідея гри створити квітковий магазин, з можливістю поливу, збору квітів, розміщення їх на вітринах і подальшим продажем. Штучний інтелект полягатиме в створенні співробітників для кожного процесу і в створенні відвідувачів, який будуть генерувати список покупок і слідувати по магазину в пошуку товару.

Вирощувати будемо ромашки, тюльпани і троянди. Для кожного типу квітки буде по дві грядки і по одному насосу, який буде активувати полив через крани. У кожній грядці буде по 3 квітки, між грядками розміщені крани поливу, один кран поливає відразу 2 квітки. Для кожного насоса створимо скрипт «Pomp Object». Реалізуємо метод «OnPlayerEnter», який буде обробляти вхід персонажа в тригерну зону. При вході будемо відзначати булеву змінну, що персонаж знаходиться в зоні насоса і запускати метод ініціалізації доступних для поливу кранів (рисунок 3.6).

```
0+3 usages  Nobody !
public override void OnPlayerEnter(PlayerController controller)
{
    InitializeWateringTaps();
    _enterTrigger = true;
    _playVibration = true;
}
```

Рисунок 3.6 – Метод «OnPlayerEnter» в об’єкті «Pomp Object»

Реалізуємо метод «OnPlayerExit», в ньому будемо відзначати булеву змінну, що персонаж більше не знаходиться в зоні насоса, це потрібно для припинення поливу. У методі «InitializeWateringTaps» будемо знаходити всі доступні крани для поливу і заносити в список.

Далі створимо метод «FillProgressBar», виклик якого розмістимо в метод «Update», за умови, що персонаж знаходиться в зоні насоса. Даний

метод призначений для заповнення радіального прогресу в інтерфейсі і для активації кранів поливу квітів (рисунок 3.7).

```
private void FillProgressBar()
{
    radialProgressBar.fillAmount += _fillAmount;
    if (_playVibration)
    {
        VibrationController.Instance.PlayVibration( _key: "Watering_Vibration");
    }

    if (radialProgressBar.fillAmount >= 1)
    {
        EnableWateringTap();
        radialProgressBar.fillAmount = 0;
    }
}
```

Рисунок 3.7 – Метод «FillProgressBar» у насосі

Як бачимо, ми заповнюємо прогрес, а коли той стає рівним або більше одиниці, то активуємо подачу води через кран, таким чином поливаючи і вирощуючи квіти.

Додамо можливість збору, для цього нам знадобиться реалізувати методи «OnPlayerEnter» і «OnPlayerExit» в об'єктах кранах. Якщо персонаж входить в зону – позначаємо булеву змінну як «true» і в методі апдейт по черзі віддаємо доступні квіти персонажу. Якщо персонаж виходить із зони – позначаємо булеву змінну як «false».

Для можливості отримання ресурсів додамо гравцеві поле «_objectsHolder», в яке вкажемо посилання на дочірній об'єкт, куди повинні розміщуватися зібрані квіти. Оголосимо метод «TakeResource», в ньому ми будемо викликати додавання нового квітки в об'єкт «ObjectsHolder», який свого часу має свою реалізацію додавання. Метод «AddToList» об'єкта «ObjectsHolder» зображений на рисунку 3.8.

```
public bool AddToList(CollectableObject collectableObject, Vector3 localScale)
{
    if (HandsIsFull())
    {
        return false;
    }

    _objects.Add(collectableObject);
    collectableObject.transform.SetParent(listBeginPosition);
    collectableObject.SetScale(localScale);
    collectableObject.RotateToBaseValue();
    SortCollection();

    return true;
}
```

Рисунок 3.8 – Метод додавання квітів у список

Тепер, коли ми маємо можливість переміщатися персонажем, поливати квіти і збирати їх, можемо додати 3 вітрини для розміщення квітів в магазині і подальшого їх продажу. У першій вітрині будуть знаходитися ромашки, у другій тюльпани і в третій троянди. Так як логіка вітрин однакова – зберігати в собі квіти, то можна зробити префаб об'єкта і після скопіювати його на сцену 3 рази, вказавши в якості мети різні типи квітів, для перщої – ромашки, для другої – тюльпани, для третьої – троянди.

Створимо модель вітрини і перемістимо її зі сцени в папку «Prefabs», таким чином створимо префаб об'єкта. Зайдемо в префаб та додамо два компоненти «Box Collider», в одному з яких відзначимо галочку «Is trigger» – це буде значити, що даний колайдер необхідний буде для обробки входження об'єктів в його зону.

Створимо в папці «Scripts» новий скрипт «ObjectVitrine», зайдемо в префаб вітрини і додамо створений скрипт в якості нового компонента. У скрипті вкажемо поле «_currentCount», яке буде зберігати в собі кількість поточних квітів, вкажемо масив позицій, куди повинні розміщуватися квіти. Додамо 2 методи на обробку входу і виходу персонажа в зону коллайдера.

Створимо метод «GetResourcesFromPlayer», який буде брати квіти від гравця і розміщувати в точки спавна (рисунок 3.9).

```
private void GetResourcesFromPlayer(PlayerController controller)
{
    if (gettingResourcesProcess != null)
    {
        StopCoroutine(gettingResourcesProcess);
        gettingResourcesProcess = null;
    }

    var resourcesCount :int = controller.ObjectsCountInHand();
    gettingResourcesProcess = StartCoroutine(
        routine: GetResourcesFromPlayerWithDelay(resourcesCount, controller, workerBehaviour: null));
}
```

Рисунок 3.9 – Метод «GetResourcesFromPlayer» у вітрині

Перевіряю працездатність системи. Активую насос, поливаю квіти, збираю їх і відношу до вітрини. Результати на рисунку 3.10.

Рисунок 3.10 – Функціонування основного ігрового циклу

Основний ігровий цикл повністю працездатний и має чітку структуру[9]. Наступним кроком можна створювати інтелектуальних агентів, які будуть виконувати роль покупців в магазині.

3.2 Розробка інтелектуальних агентів покупців

Для початку потрібно написати генератор клієнтів. Для цього створимо пустушку «[CustomerSpawner]» на сцені. Створимо новий скрипт «CustomerGenerator», додамо його як компонент до пустушки. Додамо поле «spawnDelay» для можливості налаштування затримки між створенням клієнтів, поле «maxCustomers», де будемо вказувати максимально можливу кількість відвідувачів і додамо список «_availableResources», куди будуть заноситися доступні в магазині вітрини. Наступним кроком скористаємося патерном «Фабричний метод» і створимо клас «CustomerGenerationFabrique», який буде мати в собі дві булеві змінні для можливості включити або виключити вибір випадкового кольору покупця і вибору випадкового головного убору. Основним і єдиним методом буде «GenerateRandomCustomer», він на вході приймає позицію для створення у вигляді вектора і список покупок, а на виході створює клієнта з зазначеними настройками (рисунок 3.11).

```
public Customer GenerateRandomCustomer(Vector3 position, List<WantedResource> order,
{
    Color randomColor = Color.black;
    GameObject randomHead = null;

    bool colorChange = false;
    bool headChange = false;

    if (customerBodyMaterialColor.Length > 0){...}

    if (customerHead.Length > 0){...}

    var customer = Instantiate(customerPrefab);

    if (colorChange && headChange){...}
    else if (colorChange){...}
    else if (headChange){...}
    else{...}

    customer.CheckForWrapping(wrappingChance);

    return customer;
}
}
```

Рисунок 3.11 – Реалізація фабрики створення клієнтів

Клас «Customer» має в собі поле «_beginPosition», яке зберігає вектор стартової позиції, поле «_currentOrder», в якому вказано індекс поточного номера замовлення, поле «_orderedObjects», в якому знаходиться список всіх замовлень і поле «_aiPath», в якому зберігається згенерована карта можливих проходів. Однак, цю карту ще належить створити.

Для можливості переміщення по сцені необхідно імпортувати сторонній ассет пошуку шляху. Ассет називається «Pathfinder» від автора Aron Granberg. Це алгоритм пошуку A*. Детальну роботу алгоритму розбирав у аналізі предметної області, сьогодні залишається лише застосувати готове рішення. Після імпорту створюємо об'єкт на сцені та додаємо йому компонент «Pathfinder» (рисунок 3.12).

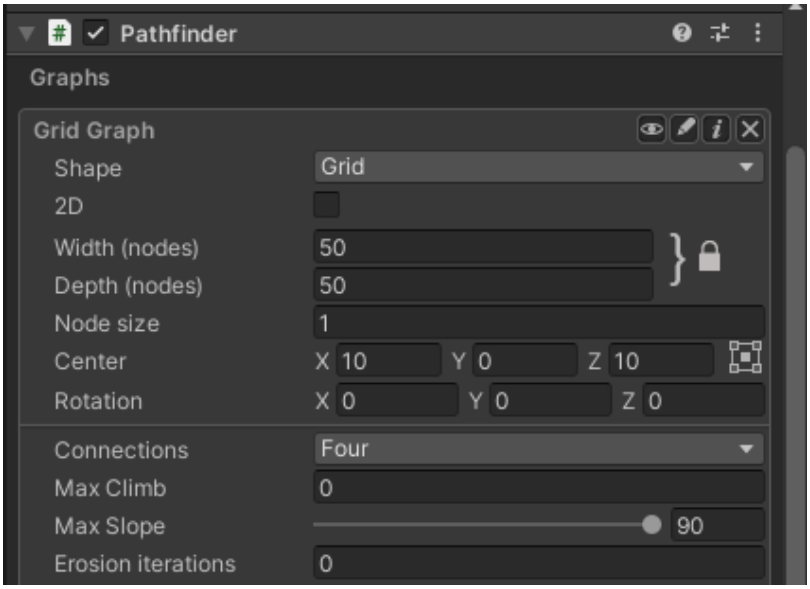

Рисунок 3.12 – Компонент «Pathfinder»

Доступні налаштування розміру сітки, її центральна позиція та перевірка сусідніх клітин, або на околицях фон Неймана, або на околиці Мура. Також успадковані налаштування від navMesh, також доступні налаштування типу колайдера у агента, його радіус і висота. Для створення необхідно натиснути кнопку «Scan», після чого отримаємо згенеровану карту (рисунок 3.13).

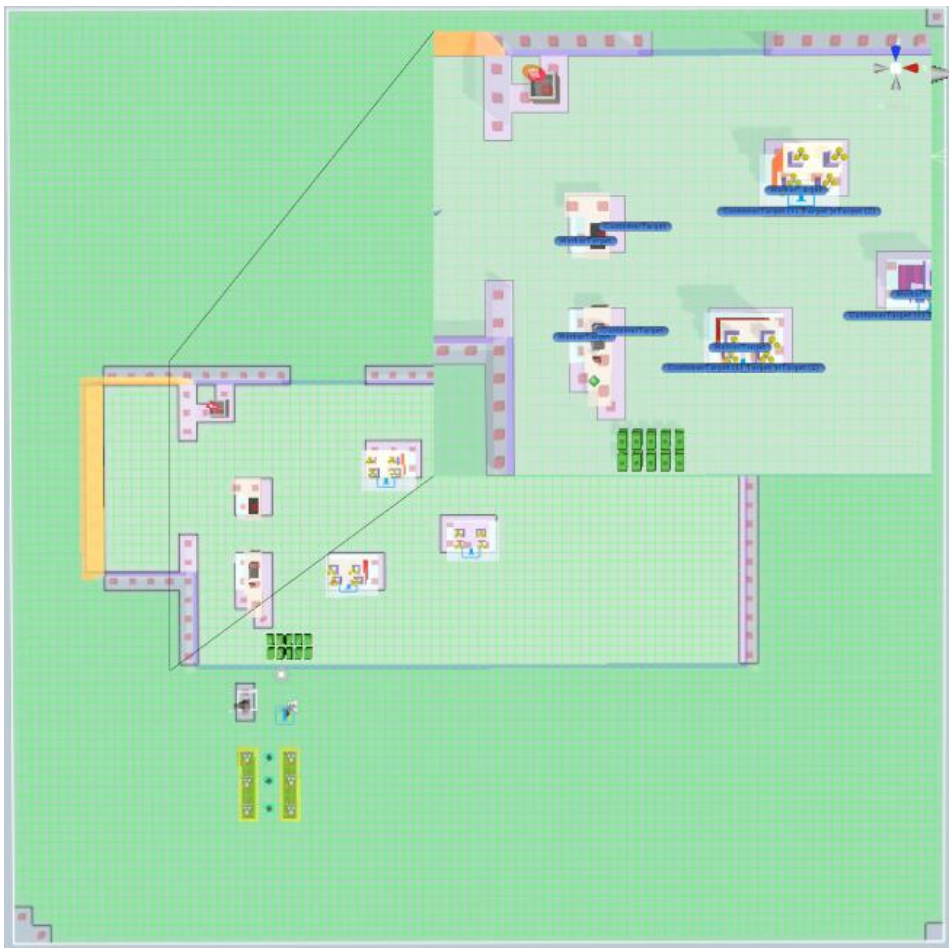

Рисунок 3.13 – Згенерована карта шляхів

Після генерації на сцені створюється сітка із зазначеними розмірами. Алгоритм обійшов перешкоди і створив навколо них сітку, за якою агенти зможуть переміщатися. Так як сітка у нас має розмірність 50 на 50, не всі моделі обробилися коректно, рішенням проблеми може стати декілька варіантів – збільшення дозволу сітки, що призведе до більшого споживання ресурсів або зміщення позицій об'єктів під сітку.

Повернемося до створення клієнтів, додамо основний метод «SetupCustomer», в якому і будемо встановлювати початкову позицію, задамо список необхідних покупок і передамо в змінну «_aiPath» посилання на згенеровану навігаційну карту. Тіло даного методу представлено на рисунку 3.14.

```
public void SetupCustomer(Vector3 position, List<WantedResource> order, MagazineManager manager)
{
    position.y = transform.position.y;
    _beginPosition = position;
    transform.position = position;

    _order = order;
    _aiPath = GetComponent<AIPath>();
    _magazineManager = manager;

    _currentOrder = 0;

    SetCustomerDestination();
    curentCheck = StartCoroutine( routine: CheckCustomersState());

    ui = MainInfoUI.Instance.GetOrderUI();
    ui.Initialize(_order[_currentOrder], transform);
}
```

Рисунок 3.14 – Тіло методу «SetupCustomer»

Клієнту необхідно вказати первинну мету, для цього викликається метод «SetCustomerDestination», паралельно з цим запускається курутина «CheckCustomersState» для відстеження стану клієнта[10]. У методі «SetCustomerDestination» клієнту вказується поточна мета зі списку, активується булева змінна «canMove», яка дозволяє рух і задається позиція мети для пошуку шляху. Наприкінці записується новий стан – «CustomerState.GoToTarget» (рисунок 3.15).

```
private void SetCustomerDestination()
{
    var target :Transform  = GetTargetForOrder();

    if (target == null)
    {
        return;
    }

    _aiPath.canMove = true;
    _aiPath.SetPath(null);
    _aiPath.destination = target.position;
    customerAnimator.SetBool(Movement,  value: true);
    _currentState = CustomerState.GoToTarget;
}
```

Рисунок 3.15 – Тіло методу «SetCustomerDestination»

У момент, коли змінюється стан, спрацьовує логіка в курутині «CheckCustomersState» (рисунок 3.16).

```
private IEnumerator CheckCustomersState()
{
    while (_currentState != CustomerState.WaitingForBuy)
    {
        switch (_currentState)
        {
            case CustomerState.GoToTarget:
                CheckTargetPlace();
                break;
            case CustomerState.TakeProducts:
                CheckResourceCollection();
                break;
        }

        yield return new WaitForSeconds(checkDelay);
    }

    yield return null;
}
```

Рисунок 3.16 – Реалізація курутини «CheckCustomersState»

Викликається метод «CheckTargetPlace», у якому перевіряється умова досягнення кінцевої точки, якщо умови правильні, рух зупиняється і клієнту вказується новий стан – «TakeProducts». Через зміну стану знову спрацьовує умова в курутині та викликається метод «CheckResourceCollection».

Значна частина логіка покупця зосереджена у методі «CheckResourceCollection». Першим рядком перевіряється чи всі товари взяв клієнт, для цього необхідно виконання умови, що поточне замовлення більше або дорівнює загальній кількості замовлень, у такому разі виходимо з поточної курутини, змінюємо стан на «WaitingForBuy» і стартуємо нову курутину «CheckCustomerSellingState» (рисунок 3.17) .

```
if (_currentOrder >= _order.Count && _currentState != CustomerState.WaitingForBuy)
{
    SetCustomerDestination();
    _currentState = CustomerState.WaitingForBuy;
    StopCoroutine(curentCheck);
    StartCoroutine( routine: CheckCustomerSellingState());

    return;
}
```

Рисунок 3.17 – Завершення усіх заказів

Наступна умова перевіряє, чи зібрано повністю поточне замовлення і якщо кількість необхідного товару більша або дорівнює необхідному в замовленні, то відзначаємо замовлення як завершене, додаємо одиницю до лічильника завершених замовлень та обнуляємо дані про кількість товарів. Звертаємось до методу «SetCustomerDestination» для пошуку нової мети та переводимо стан знову на «GoToTarget» (рисунок 3.18).

```
if (_currentOrderItemCount >= _order[_currentOrder].count)
{
    _order[_currentOrder].orderCompleted = true;
    _currentOrder += 1;
    _currentOrderItemCount = 0;

    SetCustomerDestination();

    _currentState = CustomerState.GoToTarget;
}
```

Рисунок 3.18 – Перевірка завершення поточного замовлення

Якщо поточне замовлення не укомплектовано, тоді отримуємо посилання на поточну вітрину, потім створюємо поле «resource», якому присвоюємо результат роботи методу «GiveResource» (рисунок 3.19).

```
else
{
    var magazine = Magazines.Instance.GetMagazineByType(_order[_currentOrder].type);
    var resource :CollectableObject  = magazine.GiveResource();

    if (resource != null)
    {
        resource.transform.SetParent(resourcesHolder);
        resource.SetGlobalPosition(resourcesHolder.position);
        resource.RotateToBaseValue();
        _orderedObjects.Add(resource);

        _currentOrderItemCount += 1;
        CurrentItemsCount += 1;
        _order[_currentOrder].currentCount = _currentOrderItemCount;
        customerAnimator.SetBool(Holder,  value: true);
    }
}
```

Рисунок 3.19 – Отримання ресурсу з вітрини

Далі нам необхідно реалізувати логіку здійснення покупки клієнтами. Для цього спочатку потрібно створити касу. Перейдемо на сцену, створимо об'єкт, назвемо його Cashbox, перемістимо в папку Prefabs, щоб створити префаб. У папці «Scripts» створимо новий скрипт «CashboxObject» і додамо його касі як компонент. За аналогією з вітринами створимо два «Box Collider», один з яких виконуватиме роль тригера. Створимо основне поле _customersQueue, тип даних черга. У це поле додаватимемо клієнтів для обслуговування. Додамо булеву змінну «needcheck», вона сигналізуватиме про наявність клієнтів у черзі. Напишемо метод «AddCustomerToQueue», в якому реалізуємо додавання клієнта в чергу та переклад поля «needcheck» у положення істини (рисунок 3.20).

```
public void AddCustomerToQueue(Customer customer)
{
    _customersQueue.Enqueue(customer);
    needcheck = true;
}
```

Рисунок 3.20 – Метод «AddCustomerToQueue» в касі

Додамо методи «OnPlayerEnter» і «OnPlayerExit», вони будуть обробляти вхід і вихід персонажа в зону кассии переводити бульову змінну «_exitTrigger» в стан "false" або «true». Створимо курутину «SellingProcess» (рисунок 3.21), в ній буде зосереджено основний код процесу обслуговування клієнтів. Насамперед переведемо «needcheck» у стан «false», що означає, що ми почали обслуговування клієнтів. У методі створимо цикл «while», який буде працювати доти, доки гравець не вийде з каси або закінчиться черга. У циклі створюємо поле «customer», в яке записуємо посилання на клієнта зі стандартного методу «_customersQueue.Dequeue». Викликаємо клієнту процес заповнення прогресу обслуговування, який реалізований у методі «FillProgress», створюємо очікування часу обслуговування, після чого викликаємо метод «GetMoneyFromCustome»r, який додає гроші гравцеві і викликаємо у клієнта метод «GoToExit».

```
private IEnumerator SellingProcess()
{
    coroutineStarted = true;
    needcheck = false;

    _fillAmount = 1 / customerServiceTime * Time.deltaTime;

    while ((_exitTrigger == false || _casierOntherPosition)
           && _customersQueue.Count > 0)
    {
        var customer = _customersQueue.Dequeue();
        customer.FillProgress(_fillAmount);
        yield return new WaitForSeconds(customerServiceTime);

        GetMoneyFromCustomer(customer);
        customer.GoToExit();
        customer.StopFillProgress();

        yield return new WaitForSeconds(0.1f);
    }


    coroutineStarted = false;
    yield return null;
}
```

Рисунок 3.21 – Курутина «SellingProcess» обслуговування клієнтів

Створимо метод «GoToExit» у класі «Customer», у ньому викликатимемо курутину «CheckCustomerExitState». У курутині відключаємо малювання інтерфейсу користувача, вказуємо поточний стан «GoToExit», вказуємо пошук до точки «_beginPosition». Створюємо цикл «while», в якому перевіряємо, чи дійшов клієнт до точки виходу, якщо так, то відключаємо рух і знищуємо клієнта (рисунок 3.22).

```
private IEnumerator CheckCustomerExitState()
{
    ui.DisableAll();
    _currentState = CustomerState.GoToExit;
    _aiPath.canMove = true;
    _aiPath.SetPath(null);
    var destination :Vector3 = _beginPosition;
    destination.x -= 3f;
    _aiPath.destination = destination;
    customerAnimator.SetBool(Movement, value: true);

    while (_currentState == CustomerState.GoToExit)
    {
        if (_aiPath.reachedEndOfPath && !_aiPath.pathPending)
        {
            StopMoving();
            Destroy(ui.gameObject);
            CustomerRemoveEvent?.Invoke(this, _magazineManager);
            Destroy(gameObject);
        }

        yield return new WaitForSeconds(checkDelay);
    }

    yield return null;
}
```

Рисунок 3.22 – Курутина «CheckCustomerExitState»

На цьому клас «Customer» можна важати завершеним, залишається лише підключити включення анімацій і перевірити працездатність. Для завершення проекту необхідно ще реалізувати штучних інтелектуальних агентів у вигляді працівників магазину.

3.3 Розробка інтелектуальних агентів працівників

Усі дії пошуку шляху повністю збігаються з описаними у покупцях. Перейдемо в папку «Scripts» і створимо новий клас «WorkerBehaviour». Додамо структуру перелік «WorkerType», де вказуватиметься тип співробітника, додамо структуру «WorkerState», де вказуватиметься стан співробітника. Тип співробітників будемо вказувати у панелі «Inspector». Додамо метод «FindWorkerTarget», у якому напишемо конструкцію switch-case, де в залежності від типу співробітника вибиратимемо його призначення (рисунок 3.23).

```
private void FindWorkerTarget()
{
    switch (workerType)
    {
        case WorkerType.Placer:
            FindPlacerTarget();
            break;
        case WorkerType.CottonPlacer:
            FindPlacerTarget();
            break;
        case WorkerType.Casier:
            currentTarget = _manager.Cassa.workerTarget;
            break;
        case WorkerType.Filler:
            FindFillerTarget();
            break;
        case WorkerType.CottonFiller:
            FindFillerTarget();
            break;
        case WorkerType.Wrapper:
            currentTarget = _manager.Wrapper.workerTarget;
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }

    SetTargetDestination(currentTarget.position);
}
```

Рисунок 3.23 – Метод «FindWorkerTarget»

Для касира досить проста мета – знайти касу в момент створення та перейти у її позицію. Щоб каса працювала належним чином, необхідно додати методи «OnWorkerEnter» і «OnWorkerExit», які будуть обробляти вхід і вихід співробітника в зону каси. Таким чином, нам достатньо лише додати булеву змінну «_casierOntherPosition», в якій ми зберігатимемо наявність працівника за касою і далі просто використовуватимемо методи обслуговування клієнтів з урахуванням цієї змінної.

Зі співробітниками, які допомагатимуть збирати квіти трохи складніше, проте і тут ми можемо використовувати методи, написані для персонажа, для можливості підбору квітів та їх розміщення на вітринах. Залишається лише реалізувати систему переходу станів (рисунок 3.24), для цього напишемо метод «FindPlacerTarget», в якому задамо умову «HandsIsFul», вона буде означати, що у співробітника в руках є квіти і йому необхідно віднести їх у вітрину, умову «HandsIsE» , вона буде означати, що у співробітника немає в руках квітів і йому необхідно піти збирати квіти, умову «ObjectsCountInHand», вона буде означати, що якщо на грядці немає квітів, то необхідно підійти до насоса, щоб полити та виростити квіти.

```
private void FindPlacerTarget()
{
    if (objectsHolder.ObjectsCountInHand() > 0 && _workerState == WorkerState.FillVitrine){...}

    if (objectsHolder.HandsIsEmpty() && _workerState != WorkerState.CollectObject){...}

    if (objectsHolder.HandsIsFull() && _workerState == WorkerState.CollectObject){...}

    if (objectsHolder.ObjectsCountInHand() < _data.InventoryCapacity
        && _workerState == WorkerState.CollectObject){...}

    if (_workerState == WorkerState.Waiting)
    {
        currentTarget = waitingPosition;
    }
}
```

Рисунок 3.24 – Структура умов зміни станів співробітників магазину

Останнім кроком залишається протестувати розроблену гру, перевірити як відпрацьовує штучний інтелект наших співробітників та покупців. Результати роботи зображені на рисунку 3.25 та рисунку 3.26.

Рисунок 3.25 – Демонстрація роботи ШІ покупців

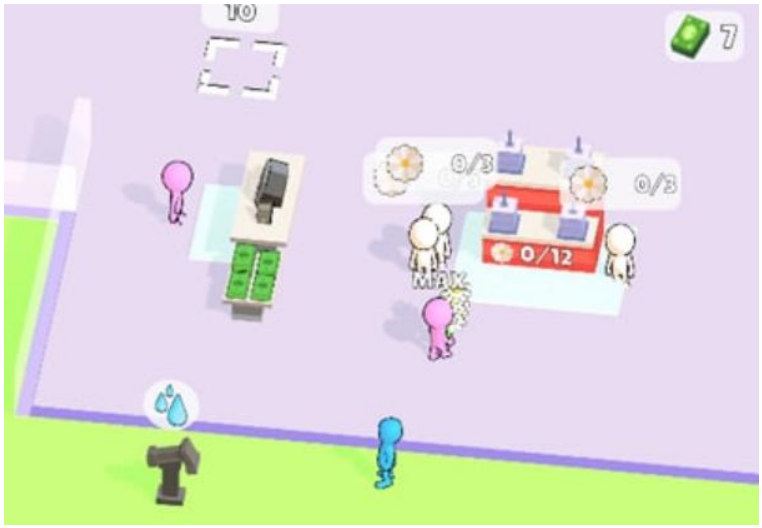

Рисунок 3.26 – Демонстрація роботи ШІ співробітників

Як бачимо, покупці становяться біля вітрин, очікують квіти, йдуть до каси, а співробітники обслуговують клієнтів та допомагають збирати квіти.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи мною було розібрано основні принципи роботи та реалізації штучного інтелекту при розробці ігрових мобільних додатків. Завдяки детальному аналізу теоретичного матеріалу було розібрано такі алгоритми пошуку шляху:

- алгоритм Дейкстри;
- алгоритм пошуку A*;
- хвильовий алгоритм;
- навігаційна сітка (navmesh);

Було проведено порівняння кожного алгоритму за швидкістю роботи, покроково описаний процес створення хвильового алгоритму в середовищі розробки Construct 3. Були описані підходи до створення штучних інтелектуальних агентів на примітивній конструкції if-else, на основі дерева рішень та на основі кінцевого автомата, принцип використання та розробки якого докладно показав з прикладу патрулюючого ворога.

Після вивчення матеріалу було описано два середовища розробки ігор. Зроблено висновки, що Construct 3 добре підходить для швидкого прототипування для перевірки гіпотез, а Unity чудово підходить для повноцінної розробки готових продуктів. Середовище Unity підтримує створення 3D проектів, а скрипти пишуться мовою програмування C#, що є величезним плюсом, адже під цю мову створено тисячі бібліотек із готовими рішеннями базових завдань. Було описано процес створення проекту в Unity, розглянуто можливості додавання об'єктів та застосування до них як базових компонентів, так і власних.

Була придумана ідея розробки гри симуляції квіткового магазину, в якій повинна бути можливість вирощування квітів, процес їх збирання та розміщення на вітринах для подальшого продажу. Для першої версії проекту було вирішено зробити 3 типи кольорів.

У якості штучних інтелектуальних агентів було виділено 2 сутності:

- покупці;
- співробітники.

Обидва типи агентів повинні мати можливість переміщення картою, для цього був застосований алгоритм пошуку шляху A*. Обидва типи агентів працюють з урахуванням кінцевого автомата, тобто, мають заздалегідь задані стани. Для покупців був створений алгоритм, що дозволяє генерувати їх випадковим чином у точці створення, після чого їм випадково вказується список покупок. Весь цикл переходу станів виглядає так:

- визначити вітрину з необхідним елементом;
- перевірити, чи є у вітрині наявність товару;
- якщо товар є, то взяти, інакше чекати;
- викреслити товар зі списку;
- перевірити, чи є у списку покупок елемент;
- якщо елемент є, то перейти на перший крок, інакше – піти на касу;
- якщо каса завершила обслуговування, то вирушатиме до виходу.

Покупці повністю справляються з поставленим завданням і поводяться подібно до звичайної людини, яка купує товар у магазині.

Для співробітників було розроблено алгоритм, який дозволяє делегувати завдання персонажа. Завдяки заздалегідь продуманій структурі вдалося застосувати методи персонажа для кожного завдання. Співробітники повністю справляються зі своїм завданням – обслуговують клієнтів на касі, поливають квіти на грядці, збирають квіти та відносять їх на вітрину.

Проект був розроблений з усіма умовами оптимізації під мобільні пристрої, з продуманою структурою для можливості розширення. По завершенню роботи проект було експортовано під операційну систему Android та опубліковано на майданчику Google Play. На даний момент проект підтримується та має понад 10000 установок.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Стрельченя Д. Ю. Використання штучного інтелекту в ігрових додатках: тези доповідей дванадцатої міжнародної науково-технічної конференції, (м. Харків, 27 – 28 квітня, 2022 р.). Харків, 2022. Том 2: секція 5, с. 19.

2. Перехрест Р.Г. Знаходження найкоротшого шляху в лабіринті до рухомих об'єктів: збірник наукових праць за матеріалами всеукраїнської науково-практичної конференції студентів і аспірантів, (м. Київ, 29 березня, 2018 р.). Київ, 2018. Секція 1, с. 41–42.

3. Unity Documentation. URL: https://docs.unity.com (дата звернення 13.04.2022).

4. Герберт Шилдт. C# 4.0: повне керівництво = C# 4.0 The Complete Reference. Київ «Вільямс», 2017. 1056 с.

5. Scirra Construct 3. URL: https://www.construct.net/en (дата звернення 15.04.2022).

6. Хвильовий алгоритм. URL: https://uk.wikipedia.org/wiki/Хвильовий_алгоритм (дата звернення 16.04.2022).

7. John Sharp Games, Design and Play – Addison-Wesley, 2016, 288 p.

8. Metanit.com. URL: https://metanit.com (дата звернення: 20.04.2022).

9. Дібрівний, О. А., Гребенюк, В. В., Кравчук, П. О., Сеньков, О. В., Кільменінов, О. А. Використання принципів solid. при розробці відеоігор на основі ігрового двигуна unity. Київ: Телекомунікаційні та інформаційні технології, Київ. 9 с.

10. Робота з курутинами. URL: https://habr.com/ru/post/216185 (дата звернення 21.04.2022).