Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Розробка симулятора робота-прибиральника у 3D середовищі з використанням паралельних обчислень

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
к.ф.-м.н., ст.викл. *Гречко А.В.*

_____
(підпис)
“____” _____ 2020 р.

Виконав студент *ФІ-4*
Нестеров М.С.

“____” _____ 2020 р.

Київ 2020

ЗМІСТ

ВСТУП

Актуальність, наукове та практичне значення обраної теми

З початку винайдення парового двигуна, а згодом і електроенергії, життя людей почало стрімко змінюватись у бік автоматизації праці та зменшення кількості робочих годин. Однією з дуже важливих сфер, яка зазнала змін, є повсякденні домашні обов'язки. Так, наприклад, поява пральних машин, посудомийок, пилососів дозволила людям економити купу часу.

«Автоматичний» означає не лише те, що робот має батарею, хорошу силу обчислювання та гарні «поведінкові» звички, а набагато більше. Як зазначає Steels, автономність – це не просто поєднання змістовних правил, це можливість створювати свої власні правила. Пересуватися, прокласти шлях або поприбирати – це усе дуже просто для нас, людей, та багатьох тварин. Різниця між роботами та тваринами полягає в тому, що тварини використовують мозок для того, щоб вирішити задачу, і ми починаємо розуміти складність цих «простих» завдань, коли аналізуємо складність мозку [1].

За останні кілька десятків років дуже активно почали застосовуватись різні підходи зі сфери комп'ютерних наук, особливо це актуально для сфери робототехніки. Ця сфера потребує розумних рішень, для того, щоб адаптуватись до зовнішніх умов, за рахунок форми, сенсорів, датчиків та ефективних алгоритмів. Домашнє середовище є дуже неструктурованим та різноманітним, на відміну від індустріального середовища, де роботи-прибиральники давно використовуються, ще у кінці XX століття. Тому до поставлених задач розробників роботів входили невеликий розмір, незалежність та автономність робота [1].

Техніка виконує таку роботу, яка для людей є дуже енергозатратна, важка, неприємна та набридлива. До недавнього часу людям необхідно було прибирати підлогу за допомогою пилососа, що також витрачає певний час та енергію. Але зараз стають усе більш популярними та доступними у ціні роботи-пилососи, які прибирають дім самостійно.

Однак, чим кращі характеристики робота, тим, відповідно, і дорожче він коштує. Це є наслідком того, що розробка робота є багатоетапним процесом, який потребує застосування знань з багатьох сфер діяльності, зокрема навігації, сканування середовища, алгоритмів пересування, інженерної частини тощо. Для того, щоб перевірити ефективність роботи певних алгоритмів, необхідно створювати симулятори робота-прибиральника.

Мета та завдання курсової роботи

Метою цієї курсової роботи є дослідження та впровадження ефективних алгоритмів пересування віртуального робота-прибиральника, його навігації в реальному часі у симульованому середовищі. Реалізувати побудову карти місцевості за допомогою влаштованих в роботі сенсорів для подальшого використання в алгоритмах пошуку шляху. Дослідити, чи впровадження паралельних обчислень в певні процеси допоможе більш доцільно розподілити ресурси та підвищити оптимізацію.

Завданням цієї курсової роботи є створення симуляції пересування, навігації, виявлення перешкод та поведінки моделі робота у 3D середовищі з обмеженнями, які діють у реальному світі, для отримання максимально достовірних результатів.

Об’єкт дослідження

Різні алгоритми, за якими пересуваються роботи-прибиральники, принципи їх роботи, можливі методи для їх покращення. Дослідження різних моделей роботів, їх відмінностей, переваг та недоліків.

Предмет дослідження

Безпосереднє створення симулятора, відтворення роботи основних складових робота, їх правильне поєднання.

Використане програмне забезпечення

Для виконання поставлених завдань було прийнято рішення використовувати ігровий рушій Unity3D, який підходить не тільки для задач, пов’язаних зі створенням комп’ютерних ігор, а також широко використовується для створення 3D симуляторів в абсолютно різних галузях, починаючи з симулювання робота для подальшого створення реальних моделей на основі отриманих даних [2], закінчуючи створенням симуляторів самокерованого автомобіля для дослідження в роботі про штучний інтелект [3].

Скриптова система рушія зроблена на Mono — вільний відкритий проект з реалізації .NET Framework. Для написання логіки було обрано мову C# - це стандартна мова для розробки проектів у середовищі Unity3D, в якості IDE (Integrated Development Environment) – Rider від JetBrains.

Структура роботи

Текстова частина цієї роботи складається зі вступу, трьох основних розділів та їх підрозділів, висновків, списку використаних джерел та додатків.

Перший розділ присвячено аналізу предметної області, а саме стану питання у сфері автономних роботів-прибиральників, їх особливостей, тонкощів та пов’язаних проблем, а також буде постановлено завдання цієї роботи.

Другий розділ присвячено опису методів, алгоритмів та підходів та вирішення поставлених завдань.

Третій розділ описує розробку моделі робота та середовища, імплементацію алгоритмів, структуру програми та обрані засоби для її написання. Також надаються скріншоти розробленого програмного інтерфейсу та опис роботи симулятора.

РОЗДІЛ 1

1.1. Аналіз сучасного стану у сфері роботів-пилососів

Робот-пилосос (або робот-прибиральник) - це електромеханічний прилад, який використовується для прибирання підлоги, меблі та килимів. Електричний мотор всередині приладу запускає вентилятор, який створює частковий вакуум, за рахунок чого повітря ззовні разом із пилом та брудом засмоктується у пилосос. Робот також оснащений спеціальними щіточками, що обертаються, які виходять за периметр пилососа, тим самим допомагають дістатись до важкодоступних місць кімнати (кути, простір під меблями). Сміття накопичується у контейнері всередині або ззовні пилососа (рис. 1.1). На роботі-пилососі встановлені сенсори, які можуть варіюватись від моделі до моделі. Так, це можуть бути інфрачервоні, ультразвукові, тактильні сенсори тощо [4].

За рахунок роботи цих сенсорів робот маневрує серед великих предметів, може заїжджати та виїжджати кутів, рухатись уздовж стіни, не пошкоджуючи меблі при цьому [5].

Однак, більш дешеві роботи-пилососи мають не ефективний паттерн очистки підлоги, і чим кращі характеристики, тим, відповідно, дорожча і ціна на ці прилади. Тому серед цілей у галузі подібної робототехніки є покращення ефективності роботи та зниження ціни за рахунок нових рішень (заміна сенсорів на більш дешеві, покращення алгоритмів прибирання) [4].

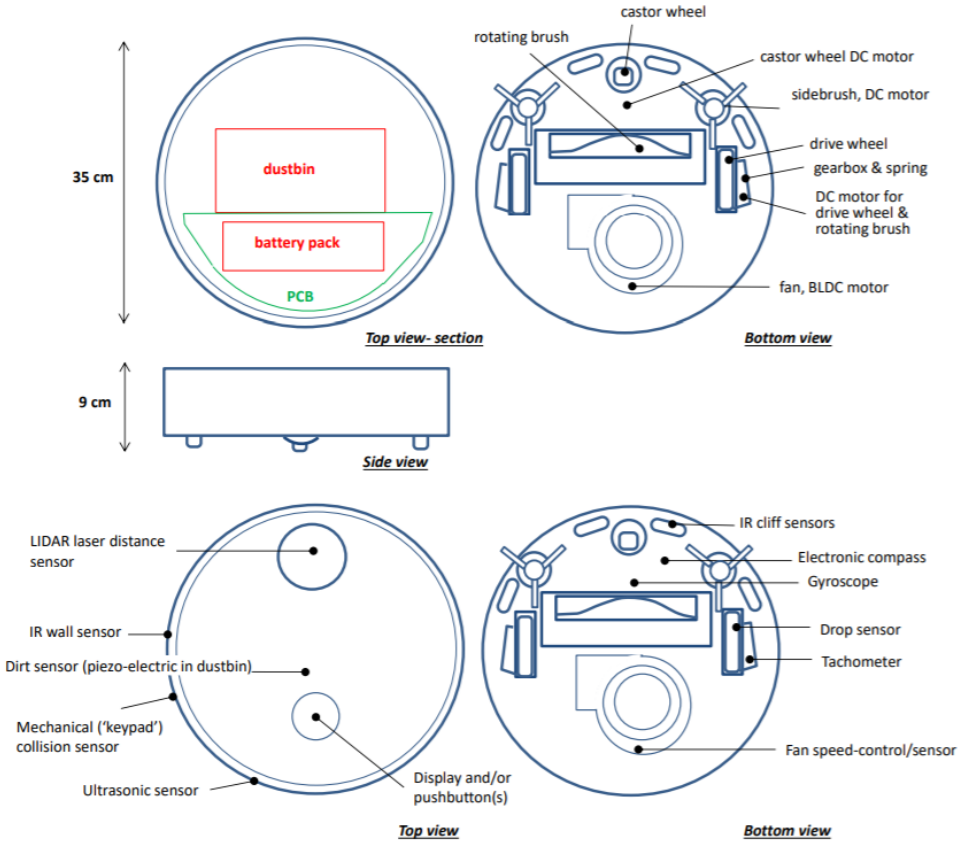

Рис. 1.1. Схематичне зображення робота-пилососа. Адаптовано з [6].

1.2. Огляд існуючих аналогів розробки роботів-пилососів

1.2.1. Характеристика принципу роботи роботів iRobot Roomba та Neato – роботи, що мають тактильні та інфрачервоні сенсори.

Дві найбільш популярні моделі роботів-пилососів – це iRobot Roomba та Neato. Робот Roomba прибирає дуже ефективно, оскільки він може охопити важкодоступні місця у квартирі. Roomba виявляє перешкоди, а також здатен до уникання «обривів» ('cliff detecting', від англ. 'cliff ' - обрив), такі як сходи та пороги. iRobot Roomba рухається у випадковому напрямі і не будує мапу місцевості приміщення. Він рухається від стіни до стіни, обертається, коли стикається з перешкодою, і покриття всієї площі займає у нього досить багато часу для прибирання [6].

Робот Neato детектує перешкоди за допомогою лазерів 360° навколо себе, а також використовує SLAM (**S**imultaneous **L**ocalization **A**nd **M**apping) алгоритм,

що дозволяє побудувати мапу кімнати для навігації і їздити по паттерну з прямими лініями, що частково перекриваються (рис. 1.2) [6].

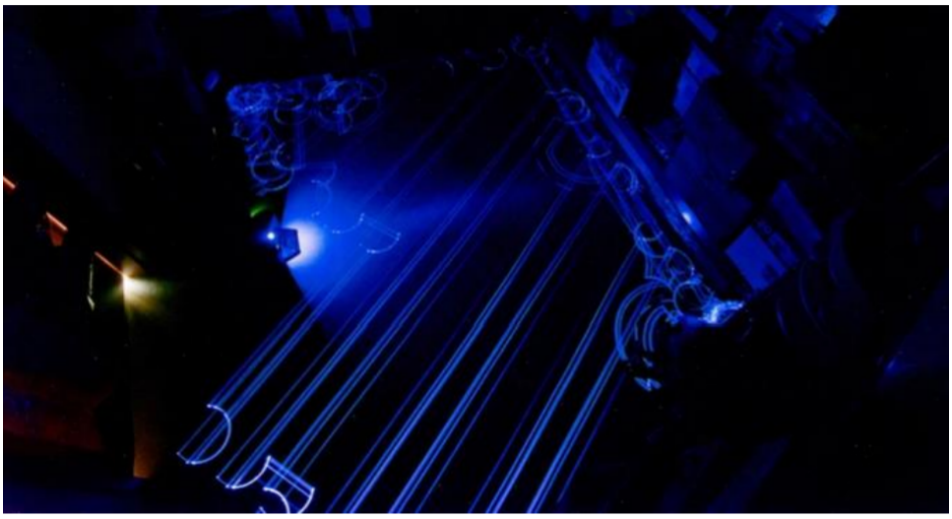

Рис. 1.2. Використання SLAM алгоритму для побудування мапи місцевості роботом Neato. Адаптовано з [6].

Тактильні сенсори знаходяться у передній частині робота – бампері. За рахунок стикання бамперу з перешкодами робот від'їжджає назад, повертається та їде далі. Цей вид сенсорів має перевагу над інфрачервоними, оскільки останні погано детектують тонкі предмети (наприклад, ніжки стільців та столів) [7].

Сенсори «обриву» працюють за рахунок роботи інфрачервоних сенсорів та під корпусом робота. Якщо промінь відбивається – це означає, що робот на підлозі і він продовжує їхати, якщо ні – то робот розвертається [7].

Алгоритм. Робот має два модулі роботи – автономний та мануальний. Під час автономного режиму робот може притримуватись наступних алгоритмів: випадкове, спіральне, «S»-подібне прокладання шляху та рух уздовж стіни (рис. 1.3) [7].

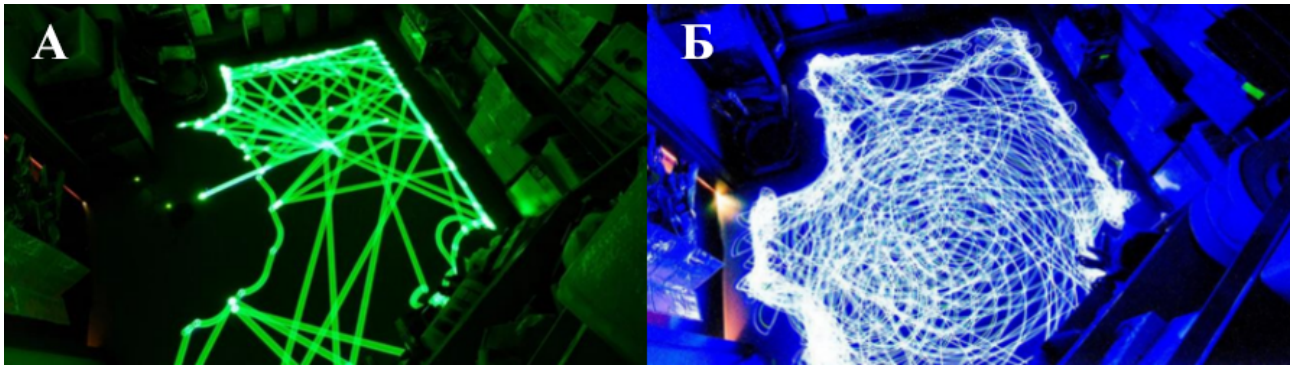

Рис. 1.3. Використання різних паттернів прибирання роботом iRobot Roomba: А - випадковий; Б – випадковий + спіральний. Адаптовано з [6].

1.2.2. Характеристика принципу роботи роботів-пилососів, що мають ультразвукові сенсори. Компактний автономний бездротовий робот використовує ультразвукові сенсори для того, щоб уникати та об'їжджати перешкоди, а також щоб побудувати найбільш ефективний маршрут кімнати для прибирання. Робот-пилосос прибирає периметр кімнати біля стін та меблів, а потім – простір підлоги, що залишився. Робот маневрує в кутах та навколо великих об'єктів за рахунок роботи акустичної радіолокаційної системи [5].

Ультразвукові сенсори допомагають роботу отримувати інформацію про зовнішнє середовище, приймати рішення та діяти відповідно. За рахунок роботи сенсорів можна безконтактно виявляти об'єкти на відстані 2-400 см з великою точністю. Перевагами ультразвукових сенсорів є те, що вони не сприймають природнє або штучне світло у кімнаті і, відповідно, ультразвук не поглинається чорними об'єктами, що може бути проблемою для сенсорів, що використовують, наприклад, інфрачервоне випромінювання. Однак, серед недоліків варто зазначити, що цей тип сенсорів погано виявляє акустично м'які матеріали, такі як тканина, наприклад [4].

Навігація. Більшість роботів уникають перешкод за рахунок випадкового стикання з предметами (наприклад, моделі, які мають тактильні сенсори). Тоді ж як роботи з ультразвуковими сенсорами можуть будувати карту місцевості навіть у середовищі, що постійно змінюється. Робот не запам'ятовує однаковий

маршрут, а щоразу адаптується до умов і прокладає новий шлях. Протокол побудований на швидкості та ефективності. Перед безпосереднім прибиранням робот-пилосос будує карту місцевості. Якщо поставити робота посередині кімнати, то процес прибирання закінчиться тоді, коли робот приїде у точку старту. При ввімкнені робота він перевіряє середовище і вирішує у якому напрямку краще поїхати. Цей процес забезпечується за рахунок перевірки відстані до найближчої перешкоди лівим чи правим сенсором. Там, де відстань буде більша, туди і поїде робот. Після цього він перевіряє, чи робот знаходиться біля кута за рахунок роботи заднього сенсору. Робот рухається назад, доки відстань не буде менше безпечної (15 см). Після закінчення цього набору дій, у робота вмикається вентилятор і він починає власне пилососити і прибирати кімнату. Потім, робот починає рухатись по прямій лінії, доки не виявить перешкоду на безпечній відстані. Якщо робот застрягне, то мотор вимикається, задля збереження енергії. Однак, він повторно запускає процес ініціалізації, щоб зареєструвати зміну положення або нові перешкоди [4].

1.2.3. Характеристика принципу роботи роботів-пилососів, що мають LIDAR сенсори. Роботи, принцип роботи яких базується на LIDAR (**L**ight **I**maging, **D**etection, **A**nd **R**anging) сенсорах, будують мапу місцевості приміщення, за допомогою випускання лазерів певної довжини. LIDAR – це метод детектування об’єктів. Принцип роботи метода полягає у тому, щоб виміряти відстань до об’єкта за рахунок випромінення пучка світла і, у відповідь, отримання відображеного світла, що реєструється сенсорами. За рахунок різниці у часі, куті пучка випроміненого світла та розміщення системи, сенсор зможе сприйняти інформацію про місцевість (рис. 1.4) [8].

Довжини хвиль лазеру варіюються від 10 мікрометрів (інфрачервоне світло) до 250 нанометрів (ультрафіолет). Для не наукових цілей використовуються такі довжини хвиль, які будуть не травматичні для ока людини. LIDAR широко використовуються у робототехніці для будування детальної мапи місцевості (наприклад, супутниках та планетоходах). Перевагами цих сенсорів є те, що вони здатні виявити не лише великі тверді об’єкти, як

ультразвукові, тактильні та інфрачервоні сенсори, а навіть аерозолі у повітрі. Тому LIDAR сенсори має найбільші переваги у детектуванні різноманітних об’єктів з різних матеріалів, якими зазвичай збагачені жилі приміщення. Серед моделей, які мають ці сенсори є Neato Botvac Connected та Xiaomi Mi LIDAR Vacuum cleaner.

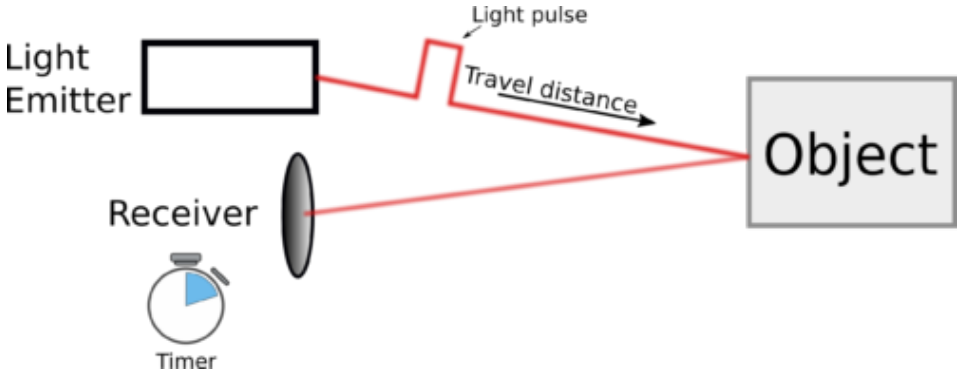

Рис. 1.4. Принцип роботи LIDAR сенсору.

1.3. Постановка завдання

В результаті вищезазначеного огляду аналогів роботів-прибиральників, їх структури та поведінки, а також розглянутих особливостей пов’язаних з різними реалізаціями тонкощів та проблем, завданням цієї курсової роботи є:

- Створити симульоване 3D середовище з діючими на об’єкти силами гравітації, інерції та правильним розрахунком колізій між предметами. Необов’язковою умовою є створення сцени з освітленням та додавання текстур на оточуючі предмети.
- Створити модель робота, що буде виглядати як типовий представник робота-прибиральника (у формі приплюснутого конусу) та рухатись за допомогою двох рушійних колес.
- Орієнтування робота у просторі має здійснюватися за допомогою сенсорів закріплених на корпусі, що основані на LiDAR технології. При

реалізації сенсорів має бути використано підхід з паралельними обчисленнями.

- Під час їзди, робот має створювати карту місцевості на основі даних, отриманих з сенсорів, за допомогою яких він зможе прокладати оптимальний маршрут до довільної точки на місцевості, наприклад, до місця з зарядкою.
- Розробити та використати оптимальні алгоритми для переміщення робота.
- Створення мінімального графічного інтерфейсу для виведення загальної інформації про поточний стан роботу та керування деякими процесами.

РОЗДІЛ 2

2.1. Алгоритм пошуку шляху

Для симуляції роботи робота-пилососа необхідно реалізувати засоби, за допомогою яких він зможе самостійно дістатися до місця зарядки при завершені прибирання або при низькому заряді батареї, зокрема реалізувати систему пошуку шляху, яка б вираховувала найоптимальніший маршрут від поточного місцезнаходження робота до необхідної позиції на карті.

Існують різні методи планування руху та знаходження шляху на карті. Одними з найбільш часто використовуваних є алгоритм Дейкстри, A* алгоритм, пошук в ширину (Breadth-First Search or BFS), пошук в глибину (Depth-First search or DFS). Але серед усіх цих алгоритмів, було обрано A*, бо він є одним з найшвидших та простих [9].

A* (з англійської A-Star - A зірочка) – є одним з найкращих встановлених алгоритмів для загального пошуку оптимальних шляхів [2], який є дуже конкурентоспроможним в порівнянні з іншими алгоритмами пошуку шляху. Алгоритми пошуку використовуються в надзвичайно різноманітних програмах та мають широкий спектр застосування, особливо у застосунках з штучним інтелектом. Алгоритм A* є відносно простим для розуміння та для імплементації. Він був розроблений у 1968 році, як покращення вже існуючого алгоритму Дейкстри, шляхом використання евристик при пошуку.

A* знаходить шлях з найменшою відносною вартістю з заданого стартового сектору до цільового. По мірі проходження карти, він прямує за шляхом з найменшою евристичною оцінкою, запам’ятовуючи впорядковану чергу з послідовних секторів шляху. Цей алгоритм має високу точність, адже він враховує вже обраховані клітини і знаходить найкоротший шлях до кінцевої точки. Базовий принцип роботи цього алгоритму базується на формулі $F(x) = G(x) + H(x)$, де x – це поточний сектор на карті, $G(x)$ – дистанція зі стартової клітини до x, а $H(x)$ – евристична функція, що оцінює ціну найдешевшого шляху

з x до кінцевої клітини (рис. 2.1). Кожен сектор отримує оцінку F, за якою і буде будуватися найоптимальніший маршрут.

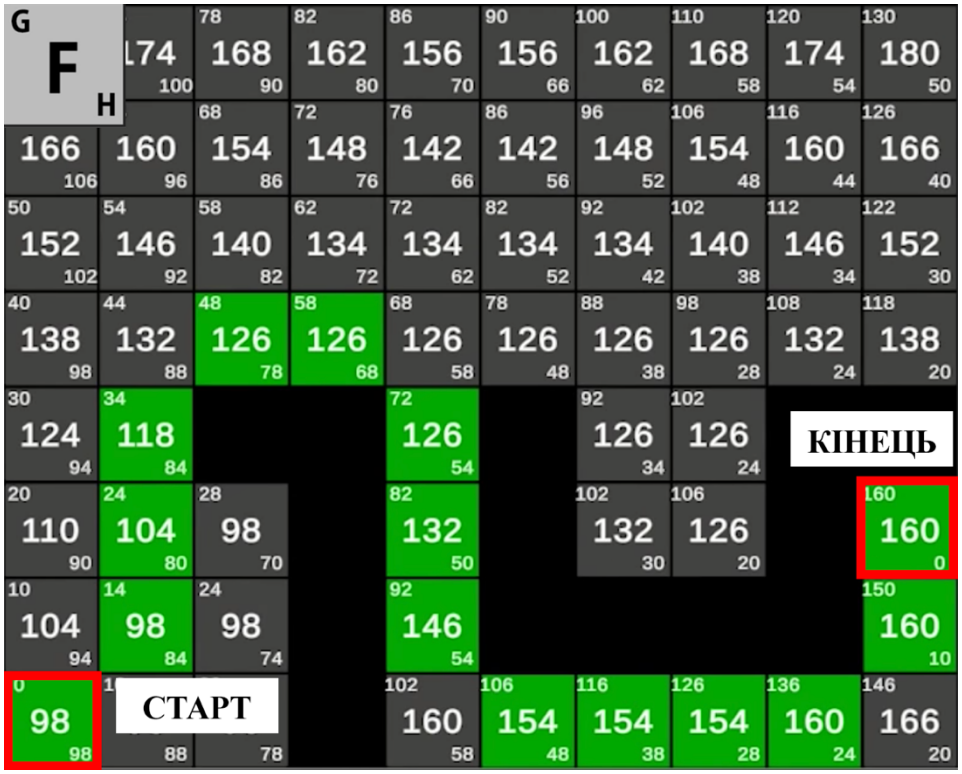

Рис. 2.1. Ілюстрація роботи алгоритму A* з відповідними позначеннями обрахованих значень F, G, H

Рухатися можна в усіх 8 напрямах (рис. 2.2) на сусідні клітини, при чому рух по горизонталі чи вертикалі оцінюється в 10 умовних одиниць, а по діагоналі – в 14. Числа 10 і 14 були обрані не просто так, в звичайній реалізації алгоритму ці значення дорівнюють 1 та 1.414 (діагональ клітини, тобто корінь квадратний з 2) відповідно для спрощення та пришвидшення обрахунків.

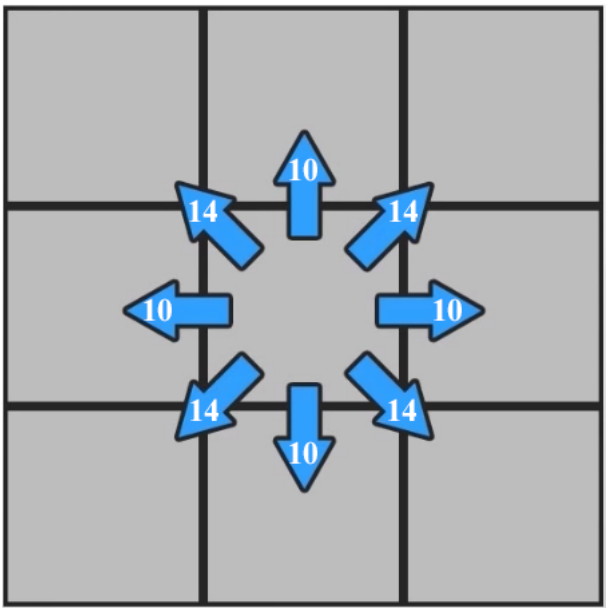

Рис 2.2. A* - напрями та вартість переміщення на сусідні клітини

2.2. Паралельні обчислення та написання багатопоточного коду в Unity

В однопоточній обчислювальній системі, одночасно надходить одна інструкція, і одночасно виходить один результат. Час для завантаження та виконання програми залежить від обсягу роботи, яку процесору необхідно виконати.

Багатопоточність - це стиль програмування, який використовує можливості процесора для оброблення декількох потоків одночасно на декількох ядрах. Інструкції та задачі виконуються одночасно, замість виконання одне за одним.

Один потік, який є головним (main thread), працює на початку програми за замовчуванням. Цей основний потік створює нові потоки для обробки завдань. Ці нові потоки виконуються паралельно і зазвичай синхронізують свої результати з основним потоком після завершення.

Для написання багатопоточного коду в Unity існує спеціальний *C# Job System*, який повністю інтегрований з усім рушієм і робить написання правильного коду набагато простішим.

Впровадження паралельних обчислень забезпечує високоефективні переваги, що включають в себе менший час на рендер кадру, а як наслідок, і їх підвищену частоту (FPS – frames per second). Використання Burst Compiler з C# Jobs покращує швидкість компілювання коду та його якість, що знижує використання акумулятору на мобільних пристроях [10].

Значною перевагою C# Jobs System є його повна інтеграція з job system, що використовується нативно в Unity. Це означає, що написаний розробником код буде ділити робочі потоки з тими, що вже використовуються рушієм для виконання внутрішніх процесів. Ця кооперація допоможе запобігти створенню нових потоків на процесорі, що знизить навантаження на обчислювальне ядро та підвищить оптимізацію.

Звісно, можна писати багатопоточний код за допомогою засобів C#, але тоді розробнику доведеться вирішувати виникаючі проблеми і обходити підводне каміння, які точно виникнуть при такому підході. Більш того, такі речі, як створення потоків, їх закриття, пули та синхронізація клалися на плечі девелопера. Але з новою системою Jobs, яка стала доступна в Unity версії 2018 року, всі ці моменти переклалися на рушій, а від розробника вимагається лише створення задач та їх виконання.

2.3. Принцип роботи C# Job System

Для керування багатонитковим кодом, job system створює *jobs (задачі)* замість потоків, також ця система керує робочими потоками на декількох ядрах процесору. Зазвичай створюється один робочий потік на одне логічне ядро процесора, щоб уникнути переключення контексту. Job system додає кожну задачу в чергу для виконання. Робочі потоки, що керуються системою, беруть елементи з черги і виконують їх. Job system слідкує за тим, щоб задачі виконувались в правильному відповідному порядку, а також керує залежностями.

Job (задача) – це маленьких одиниця роботи, яка виконує одне конкретне завдання. Загалом, щоб виконати будь-які обчислення в системі jobs, необхідно використовувати ці задачі, які являють собою об’єкти, що складаються з методів і даних для їх обрахування. Завдання можуть бути самостійними, або вони можуть залежати від виконання інших завдань, перш ніж вони зможуть почати свою роботу. Це дуже типова ситуація, адже зазвичай один job підготовлює дані для іншої наступної job. Дуже важливо зберігати порядок залежностей для правильної роботи.

Кожен об’єкт job – це структура, що наслідує один з трьох інтерфейсів Job System.

РОЗДІЛ 3

3.1. Аналіз технічного завдання

Як вже було сказано, в симуляції мають діяти обмеження на оточуючий світ, об’єкти у ньому та самого робота, які максимально наближають реалізацію до реального світу. Такі речі, як гравітація та колізії між предметами реалізовані в Unity на програмному рівні. Наприклад, шляхом додавання скрипту Rigidbody на ігровий об’єкт, на нього починають діяти сила гравітації, імпульси, що були передані від інших ігрових об’єктів, також налаштовується його вага та інші параметри фізичного предмету. Для визначення фізичних границь ігрового об’єкту, достатньо додати один із колайдерів (з англ. collide - зіткнення): Box Collider, Capsule Collider, Mesh Collider або інші.

Для реалізації руху робота в симульованому середовищі були проаналізовані реальні аналоги, в абсолютній більшості з яких є 2 рушійних колеса по боках і одне маленьке колесо спереду, яке обертається на 360 градусів (рис. 3.1). Тому в симуляції було вирішено реалізувати саме таку модель, в якій використовуються вбудовані в ігровий рушій Wheel Colliders для задніх колес, які обертаються незалежно одне від одного, а подача енергії на них контролюється скриптом руху робота. Переднє ж колесо було вирішено зробити звичайною сферою з дуже маленьким коефіцієнтом тертя.

Рис. 3.1. Модель рушійної системи робота. Адаптовано з [4].

Для орієнтування у просторі та виявлення перешкод робот використовує спеціальні сенсори, які повідомляють про те, чи є якісь об’єкти в зоні їх дії, яка відстань до найближчого з них та в якому напрямі його було зафіксовано. За своєю дією це нагадує принцип роботи лазерних радарів, які випромінюють короткі пульсуючі промені в межах певного куту дії (рис. 3.2). Для їх симуляції була використана Physics.Raycast функція. Для підвищення ефективності та оптимізації швидкості оброблення інформації сенсорами, було прийнято рішення розпаралелити цей процес.

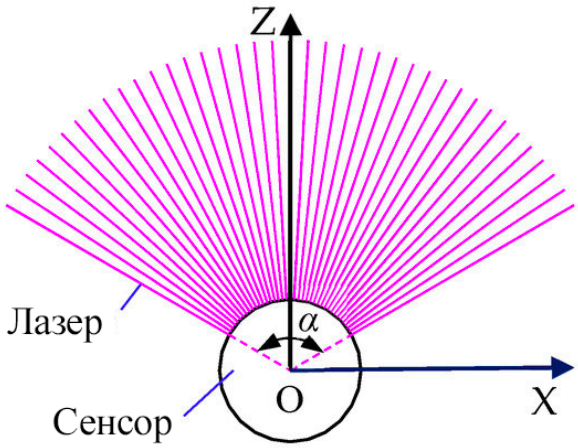

Рис. 3.2. Діаграма принципу роботи сенсору. Адаптовано з [11].

Надзвичайно важливою частиною цієї роботи є безперебійний аналіз оточуючого середовища та власного стану. Необхідно передбачити, що може виникнути ситуація, коли робот заїхав у кут і не може звідти виїхати, або він врізався у щось і застряг. В таких випадках робот має автоматично від’їжджати назад і повертати або застосувати інші маневри для уникання таких перешкод.

В роботі також має бути реалізована симуляція акумулятору та зарядної станції. Якщо робот працює, то кожну секунду рівень заряду акумулятору має трохи зменшуватись, коли ж рівень заряду падає до нуля, всі компоненти, такі як сенсори та колеса, мають вимикатись і все зупиняється. Має бути можливість увімкнути та вручну виставити рівень заряду. Однак, щоб не допускати ситуацій, коли робот вимкнувся посеред прибирання, має бути реалізований алгоритм пошуку шляху до зарядної станції до якої робот прямує за найкоротшим маршрутом, коли заряд батареї-акумулятору складає менше 20%. В якості

алгоритму пошуку шляху було обрано A*, для реалізації якого необхідно мати представлення навколишнього середовища на основі сітки, кожна клітина якого класифікується як вільний простір або перешкода. Диференціювання їх типу буде проводитись за допомогою сенсорів на роботі, які будуть динамічно заповнювати клітини сітки інформацією.

3.2. Обґрунтування алгоритму й структури програми

Умовно всю програму можна розподілити на такі компоненти: навігаційний модуль, модуль сенсорів та комплексний об'єднуючий модуль, що відповідає за переміщення, аналіз поточного стану, вибір програми, за якою робот буде їхати.

3.2.1. Сенсор. Кожен сенсор – це окремий незалежний компонент, єдина задача якого – виявити перешкоду в зоні його дії, визначити відстань до неї та вектор напряму, зберегти ці значення та повідомити про виявлену перешкоду інші компоненти через зручний і зрозумілий інтерфейс.

Працює сенсор таким чином: зі своєї основи він випромінює промені з обмеженою дальністю дії в площині перпендикулярній осі Y (паралельній підлозі) в діапазоні кута α. Довжина променю, їх кількість та кут α можна налаштувати (рис. 3.3).

Всього по периметру корпусу було вирішено розмістити 3 сенсори: зліва, справа та по центру, під кутами -70, 70 та 0 градусів відповідно (рис. 3.3). Таким чином робот орієнтується, з якої сторони знаходиться перешкода та зможе оминати її.

Випускання променів, їх обробка та аналіз має відбуватися швидко і з великою частотою оновлень: мною було обрано періодичність в 10 оновлень на секунду, щоб у системи був час зреагувати на появу перешкоди та на вибір інструкції для дій.

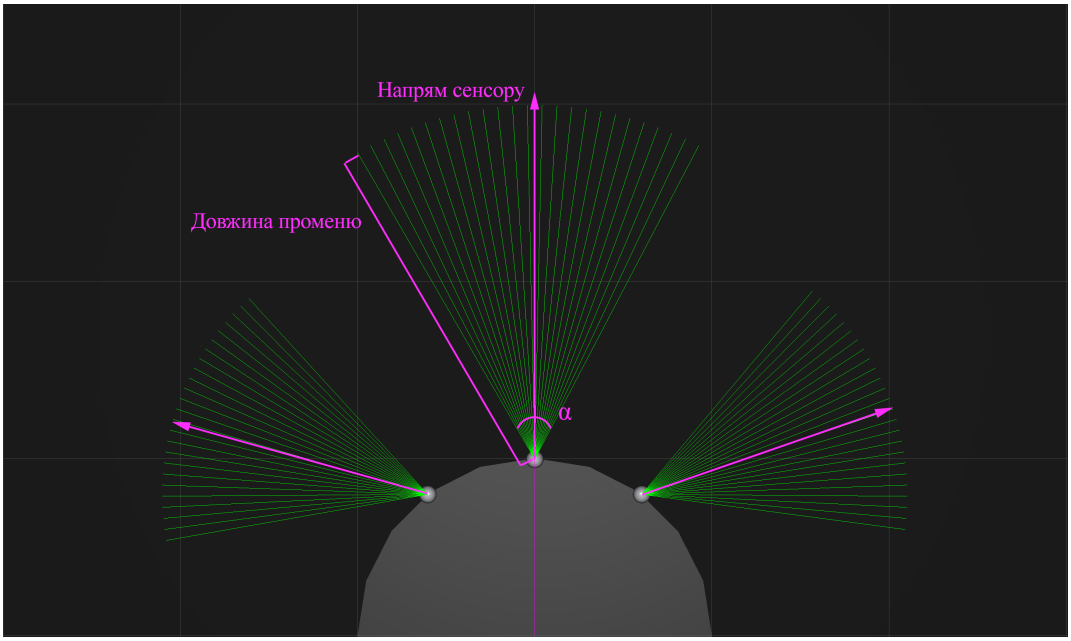

Рис. 3.3. Розташування сенсорів на корпусі; зображення параметрів, що налаштовуються

3.2.2. Навігаційний модуль. Для орієнтування в просторі одних тільки сенсорів недостатньо: робот може оминати перешкоди, але також потрібно мати інформацію про своє місцезнаходження в глобальних координатах, мати карту з перешкодами, щоб можна було пересуватись ефективно між різними кімнатами квартири. Наприклад, це необхідно для побудування маршруту до зарядної станції, адже просто їхати до неї напряму майже завжди буде неможливо.

Для вирішення цієї проблеми було обрано евристичний алгоритм глобальної оптимізації пошуку A*. Необхідною частиною цього алгоритму є представлення середовища як сітки, яка буде складатися з маленьких «нод» (клітин). Кожна з них містить інформацію про те, чи є в ній перешкода чи вона вільна для пересування. За замовченням, всі клітини позначаються як вільні, а вже під час проходження та сканування сенсорами робота навколишнього середовища, в ноди заноситься інформація про їх зайнятість перешкодою. Блок-схема алгоритму приведена нижче на рис. 3.4.

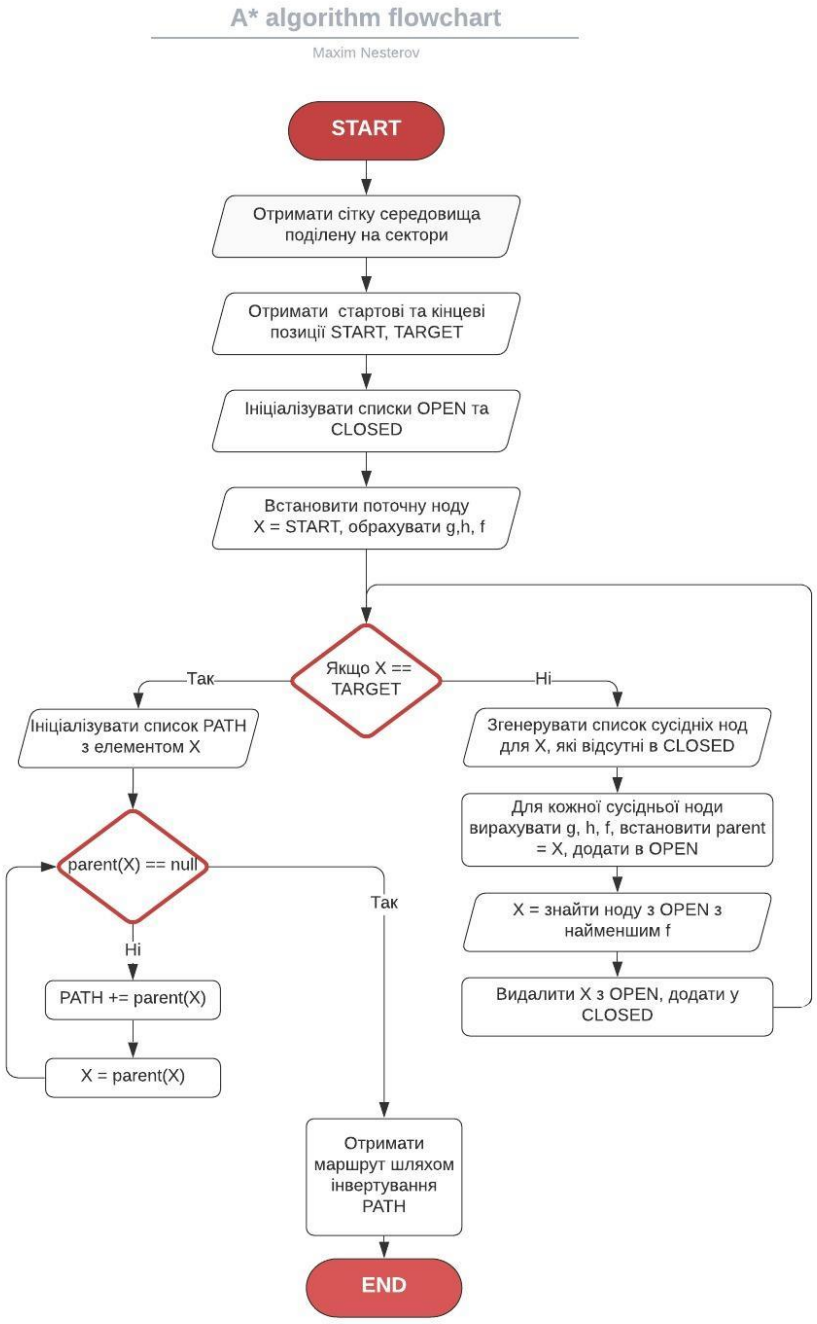

Рис. 3.4. Блок-схема алгоритму пошуку шляху

3.2.3. Комплексний модуль, що відповідає за рух. Комплексний модуль, що відповідає за переміщення робота. Він керує колесами, отримує дані з сенсорів на основі яких приймає рішення щодо подальших рухів. Налаштований таким чином, щоб рухатися по кордону кімнати, притримуючись стін для того, щоб скласти карту середовища і прибрати зони біля стін та кутів. Рухається робот на прямолінійно під стіною, а постійно під'їжджаючи та від'їжджаючи від

неї, що дозволяє йому прибрати сміття біля стін, а також більш ефективно покрити всю область. Траєкторія руху зображена на рисунку 3.5.

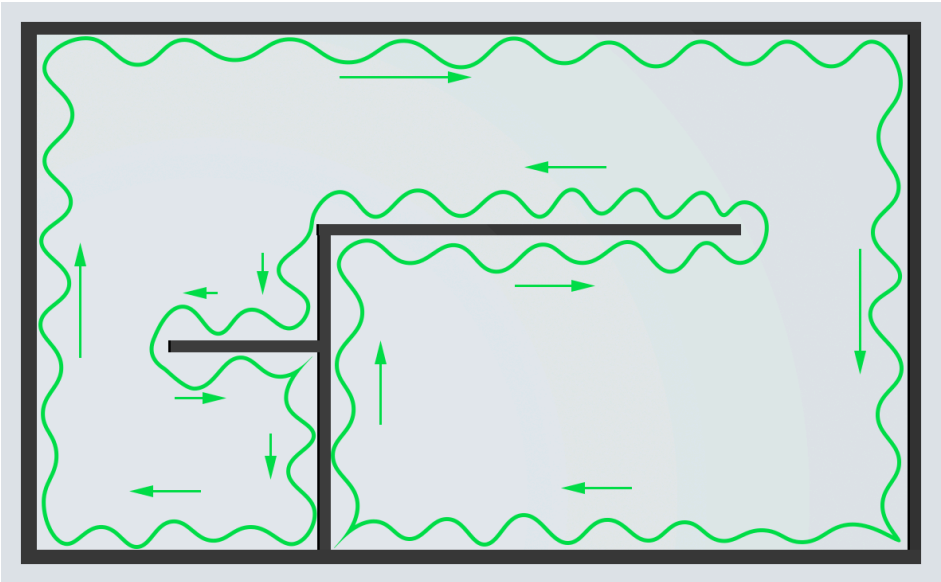

Рис. 3.5. Траєкторія руху роботу вздовж стін

3.3. Обґрунтування вибору засобів розробки

Основою для цього проекту є ігровий рушій Unity3D 2019.3.4, в якому об’єднані різні програмні засоби для зручного створення проектів. Редактор має зручний інтерфейс з підтримкою Drag & Drop різних вікон інтерфейсу, завдяки чому його легко налаштувати під свої потреби та проводити налагодження програми прямо в редакторі. В юніті проект складається з однієї чи більше сцен, які під час гри можуть перемикатися між собою. В кожній сцені є ігрові об’єкти, які можуть бути 3Д моделями або просто порожніми об’єктами в просторі. В кожного об’єкту є набір компонентів, які можна додавати на нього або видаляти, і з цими компонентами можуть взаємодіяти скрипти. Є влаштовані в рушій компоненти, такі як Rigidbody – для додавання об’єкту фізичних властивостей, але звісно компоненти можна створювати і самому для імплементації власної логіки та поведінки об’єктів – достатньо створити файл з класом, що буде успадковувати клас MonoBehaviour. Окрім компонентів у кожного об’єкту є назва, яка не є індивідуальною, тег і спеціальний шар для відображення моделі.

В кожного об’єкту на сцені, навіть порожнього, обов’язково є компонент Transform, в якому зберігається позиція, поворот і розміри по всіх 3 осях [12].

Починаючи з Unity 1.0, рушій підтримує тільки 1 мову програмування для написання скриптів, а саме C#. Звісно, можна написати окремі модулі програми на інших .NET мовах і скомпілювати в окрему DLL бібліотеку, яку можна імпортувати в Unity, але для програмування і компілювання проекту в реальному часі єдиною можливістю є використання мови C# [12].

В Unity є влаштоване повнофункціональне рішення для контроля версій для всіх ігрових скриптів та ресурсів під назвою Unity Asset Server. Він ідеально підходить для розробки в команді, для великих проектів з тисячами файлів, зміни в яких можна переглядати прямо всередині редактора Unity, ресурси зберігаються в базі даних PostgreSQL, що забезпечує високий рівень оптимізації. Але для цієї роботи весь цей функціонал не використовувався б і на половину, тому для VCS було обрано звичайний git з прив’язкою на GitHub.

3.4. Опис розробки програми

Перш за все необхідно було створити модель робота та оточуючого середовища. Спочатку за модель було обрано звичайний циліндр, але потім його було замінено на більш реалістичну версію, яку було взято з [13]. Для створення середовища були додані стіни, що формують умови квартири та підлогу.

Наступним кроком необхідно було реалізувати найпростіше пересування робота по горизонтальній поверхні в будь-яку сторону, яким би можна було б керувати задавши вектор руху відносно напряму робота. Як вже було сказано, для відтворення поведінки робота, необхідно реалізувати пересування за допомогою двох рушійних колес та одного вільно-обертаючого колеса спереду. За керування подачі потужності на колеса відповідає модуль MovementController. Також в цьому модулі реалізовані такі функціональні можливості, як різні режими їзди, їх автоматичне переключення режимів, перевірка на унеможливлення пересування в заданому напрямку при

непередбачених обставинах, наприклад коли робот врізався в кут і застряг, включається програма від'їзду назад. Також в цьому модулі оброблюються дані з сенсорів, на основі яких і можна приймати рішення щодо подальшого руху.

За навігацію, зберігання сітки карти з інформацією про перешкоди, побудову шляху та його перерахування у разі виявлення нової перепони відповідає модуль навігації. Для поділення карти на сітку, вона розбивається на клітини, в кожної з яких є інформація щодо своїх координат на сітці, розміру та позиції сітки. На основі цих даних можна вирахувати глобальні координати клітини.

Об'єкт класу сітки зберігає в собі двомірний масив клітин, відповідає за правильну ініціалізацію сітки в глобальних координатах, має функціональність для знаходження найближчої клітини до заданої точки, а також повідомляє підписників про зміни в сітці та елементах.

3.5. Створення об'єктів і розробка головної програми

Для створення моделі робота його було поділено на складові:

- Візуальна модель
- Колайдери (використовуються для визначення границь об'єкта, обрахування зіткнень, тощо)
- Колеса
- Об'єкти сенсорів

Візуалізацію ієрархії об'єкту в Unity наведено на рисунку 3.6.

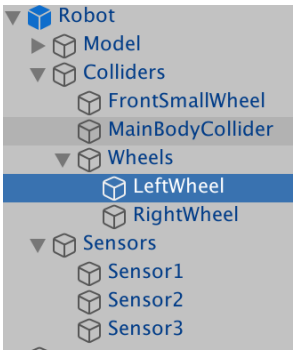

Рис. 3.6. Модель ієрархії складових робота

На модель робота було додано 3 сенсори, кожен з яких – це пустий GameObject, з моделлю маленької сфери для візуалізації, і скриптом RobotSensor. Сенсор зліва стоїть під кутом -70 градусів по осі Y, а відповідно справа – під кутом 70 градусів. Параметри сенсорів представлено нижче на рисунку 3.7.

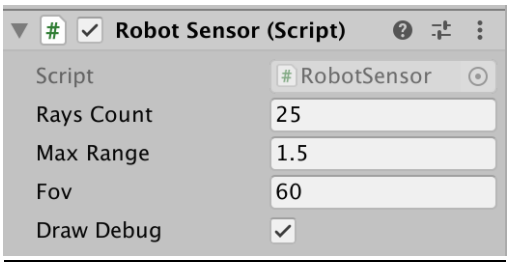

Рис. 3.7. Параметри сенсора

На самого робота було додано 3 скрипти: RobotController, RobotMovementController та PathNavigation з параметрами, що зображені на рисунку 3.8. Параметр HomePoint у RobotController та OriginPoint у PathNavigation – це посилання на позицію підзарядки. GridCellSize дорівнює 2.1, адже розмір робота дорівнює 2 у системах координат симулятора.

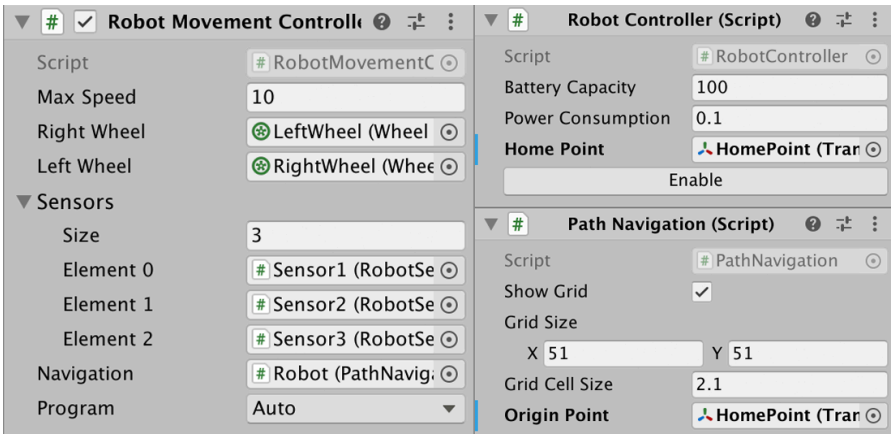

Рис. 3.8. Параметри скриптів, що додані на об’єкт робота

Для створення графічного інтерфейсу користувача було використано UI систему Unity. Всі елементи додавались на спеціальний об’єкт Canvas. Були використані компоненти Text, Image, Slider (для скейлінгу часу, візуалізації та керування зарядом батареї, а також для візуалізації роботи колес), Button та

Toggle. Створено скрипт UIController для з’єднання візуальної складової з даними програми.

3.6. Опис файлів даних та інтерфейсу програми

Інтерфейс програми складається з зображення виду згори поверхні підлоги, по якій робот пересувається у просторі певного приміщення. Камера розташована так, щоб робот знаходився в центрі екрану, і рухається разом з ним.

На екран виводяться поточний вектор напряму руху робота та візуалізація променів сенсорів, які зафарбовуються в різні кольори в залежності від виявлення перешкоди перед собою (рис. 3.6).

Також на підлогу проектується сітка місцевості, що використовується роботом для навігації та прокладання маршрутів в довільні точки. Вона поділена на клітини, кожна з яких зафарбовується в білий або червоний колір в залежності від того, чи сенсори виявили перешкоду в її площині (рис. 3.6).

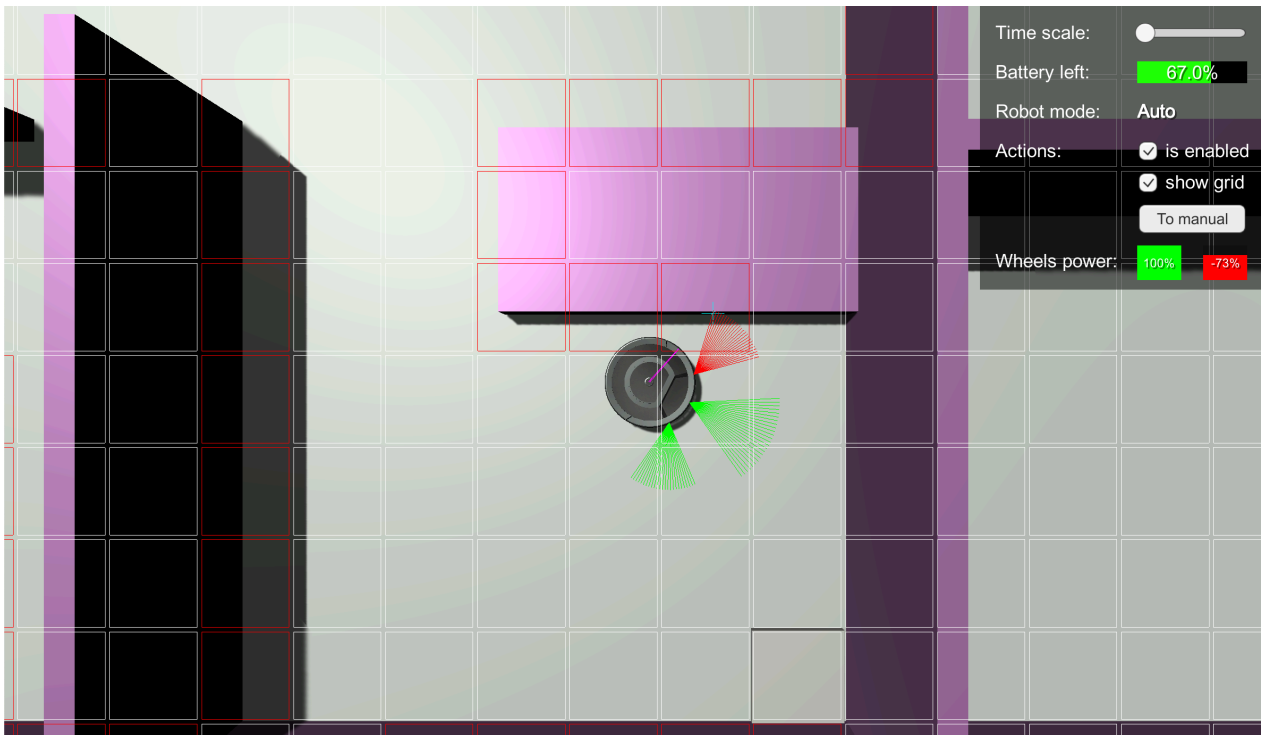

Рис. 3.6. Інтерфейс програми під час роботи, візуалізація променів сенсорів та сітки середовища, вид на робота зверху

Увімкнути показ сітки можна у спеціальній панелі у правому верхньому куті вікна. В цій панелі відображаються елементи інтерфейсу користувача, за допомогою яких можна взаємодіяти з програмою та які показують корисну інформацію. Опис кожного елементу наведено нижче, а UI продемонстровано на рисунку 3.7:

- Поточний стан робота-прибиральника, а саме один з:
    - Auto – звичайний автоматичний режим, в якому робот застосовує необхідні алгоритми пересування, сканує поверхні, об'їжджає перешкоди.
    - Stuck – включається, коли робот застряг, а саме у випадках, коли він за деякий час дельта проїхав дуже малу відстань відносно своєї максимальної швидкості, або ж у випадках, коли всі 3 сенсори виявили перешкоду на дуже малій відстані від себе.
    - Navigating – режим, коли робот їде по прокладеному маршруту до визначеної позиції. В цьому режимі він притримується заданої траєкторії руху, але при цьому реагує на отримані дані з сенсорів, у випадках, коли вони зафіксували перешкоду занадто близько, і вживає необхідні міри для уникання колізій.
    - Manual – в цьому режимі робот повністю керується користувачем з клавіатури за допомогою клавіш вперед, вниз, вліво, вправо.
- Поточний заряд батареї у відсотках, який можна за необхідності відредагувати вручну. Адже при заряді менше 20% робот включає режим Navigating і їде на точку підзарядки.
- Слайдер для регулювання швидкості часу у симуляції. Максимальне прискорення – у 5 разів. Досягається це не за рахунок збільшення швидкості робота, адже в такому випадку його буде більше заносити і він не буде встигати маневрувати між перешкодами. Замість цього просто змінюється глобальна змінна Unity Time.timeScale, яка відповідає за відносну швидкість всіх процесів в рушії, таких як фізичні показники, ефекти, переміщення об'єктів і т.д.

- Візуалізація потужності на ліве та праве колесо робота. Виводиться потужність у відсотках, а також у вигляді зафарбованих контейнерів, зелений колір означає, що колесо обертається вперед, а червоний – що назад.
- Кнопки та перемикачі для керування роботом і елементами інтерфейсу, а саме:
    - Перемикач, що вмикає або вимикає робота.
    - Кнопка, що переводить робота в режим Manual та назад.
    - Перемикач, що відповідає за показ сітки карти, поділену на клітини, що зафарбовані в білий або червоний колір (вільний простір або перешкода відповідно).

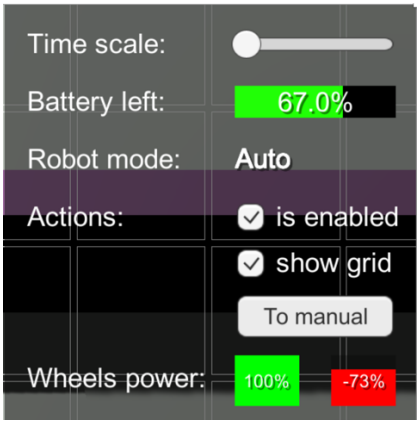

Рис. 3.7. Користувацький інтерфейс для взаємодії з програмою

3.7. Тестування програми і результати її виконання

Важливою частиною цієї роботи є навігація робота, побудова карти місцевості та правильна побудова маршруту від поточного місцезнаходження до довільної точки. Під час прибирання робот постійно сканує оточуюче середовище сенсорами і при виявленні перешкоди, шукає клітину на сітці до якої виявлена точка знаходиться найближче і помічає цю клітину як перешкоду. Під час тестування робот правильно детектував всі клітини, поряд з якими було виявлено перешкоду.

Кожну секунду рівень заряду батареї робота поступово зменшується і при рівні меншому за 20% автоматично включається режим навігації до точки підзарядки. Робот будує маршрут базуючись на карті, що у нього є на поточний момент. Якщо всі перешкоди було просканово, то маршрут будується правильно з першого разу (рис. 3.9).

Але звісно не завжди всі об’єкти були просканові і додані на сітку, тому маршрут може будуватися через стіни і інші перешкоди (рис. 3.8, А). При виявленні нової перешкоди, маршрут автоматично перебудовується базуючись на нових даних і робот продовжує рух вже по новій траєкторії (рис. 3.8, Б).

В режимі навігації робот прямує за заданим маршрутом, але при русі по діагоналі можуть траплятися колізії з перешкодами, тому при виявленні сенсорами об’єктів занадто близько, перевага віддається на маневри, що допоможуть уникнути зіткнення.

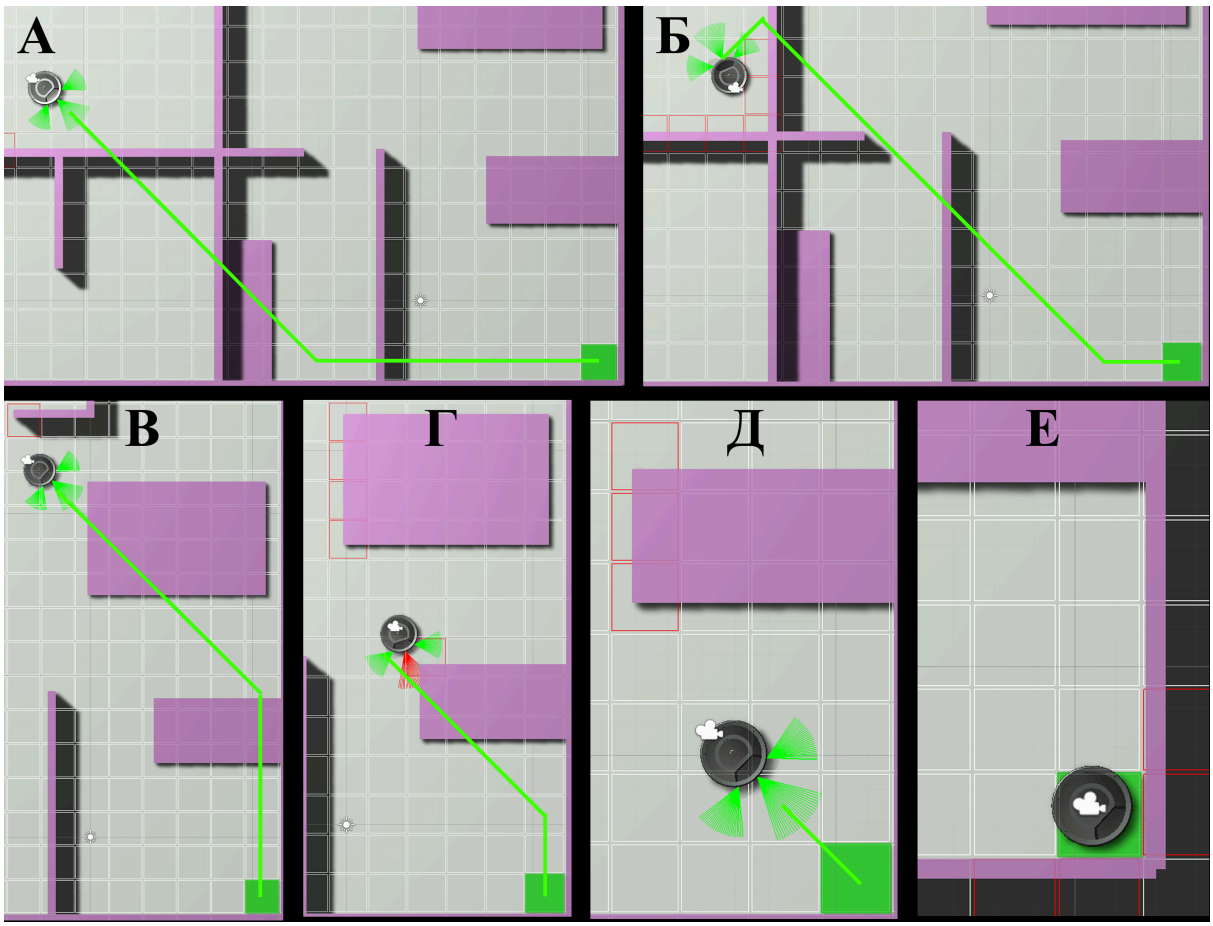

Рис. 3.8. Візуалізація роботи алгоритму побудови маршруту до точки, з динамічним виявленням перешкод та перерахунком шляху

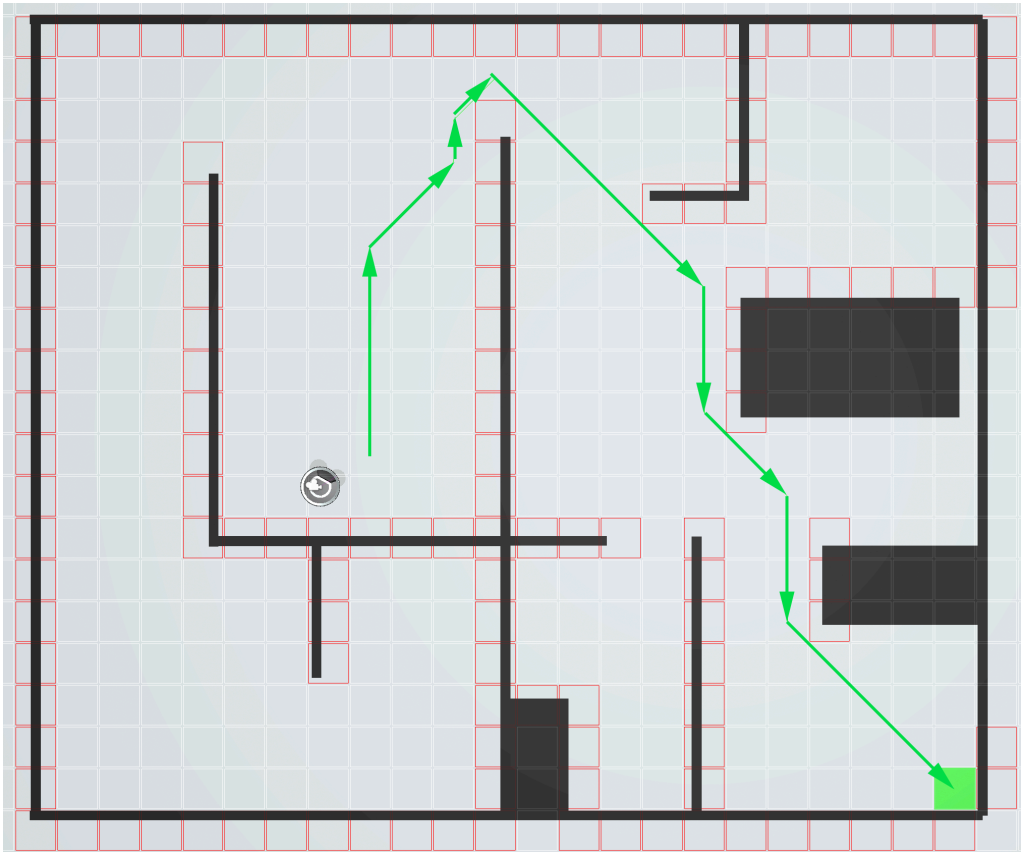

Рис. 3.9. Побудова маршруту при повністю відомій карті місцевості

ВИСНОВКИ

1. В результаті проведених в курсовій роботі досліджень, встановлено, що ігровий рушій Unity підходить для створення 3D симуляторів, адже він відносно простий у використанні, для написання скриптів використовується мова C#, має широкий вбудований функціонал і є дуже пластичною платформою.
2. Створена модель робота використовує LiDAR сенсори, що успішно продемонстрували свою роботу для орієнтування у просторі, а також для побудови сітко-подібної двомірної карти середовища.
3. В сенсорах було імплементовано C# Job System - систему Unity для написання багатопоточного коду і паралельних обчислень. Через те, що на корпусі робота встановлено всього 3 сенсори, які разом використовують незначну частину обчислювальної потужності, значного приросту в швидкодії програми виявлено не було.
4. За допомогою алгоритму пошуку шляху A*, робот без проблем прокладає маршрут до будь-якої позиції на площині місцевості і рухається по ньому, а також динамічно перебудовує маршрут при надходженні даних з сенсорів про виявлену перешкоду.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

[1] Ulrich I, Mondada F, Nicoud JD. Autonomous vacuum cleaner. *Rob Auton Syst* 1997; 19: 233–245.

[2] Víctor H. Andaluz , Jorge S. Sánchez , Jonnathan I. Chamba PPR, 121 Fernando A. Chicaiza , Jose Varela WXQ, Cepeda1 CG and LF. Unity3D Virtual Animation of Robots with Coupled and Uncoupled Mechanism. 2016; 1: 267–280.

[3] Al-ahmad AS, Kahtan H. *CAIAS Simulator: Self-driving Vehicle Simulator for AI Research*. Springer International Publishing. Epub ahead of print 2019. DOI: 10.1007/978-3-030-00979-3.

[4] Asafa TB, Afonja TM, Olaniyan EA, et al. Development of a vacuum cleaner robot. *Alexandria Eng J* 2018; 57: 2911–2920.

[5] Lee HJ, Park HJ, Kim S. A study on authentication mechanism using robot vacuum cleaner. *Lect Notes Comput Sci* 2005; 3483: 122–127.

[6] Viegand Maagøe A, Van Holsteijn en Kemna BV. Review study on vacuum cleaners - Final report.

[7] Hasan KM, Abdullah-Al-Nahid, Reza KJ. Path planning algorithm development for autonomous vacuum cleaner robots. *2014 Int Conf Informatics, Electron Vision, ICIEV 2014*. Epub ahead of print 2014. DOI: 10.1109/ICIEV.2014.6850799.

[8] Starek MJ. Light detection and ranging (Lidar). *Encycl Earth Sci Ser* 2016; 383.

[9] Handy Permana SD, Yogha Bintoro KB, Arifitama B, et al. Comparative Analysis of Pathfinding Algorithms A *, Dijkstra, and BFS on Maze Runner Game. *IJISTECH (International J Inf Syst Technol* 2018; 1: 1.

[10] C# Job System, https://docs.unity3d.com/Manual/JobSystemOverview.html.

[11] Li Y, Dai S, Shi Y, et al. Navigation simulation of a mecanum wheel mobile robot based on an improved A* Algorithm in unity3D. *Sensors (Switzerland)*; 19. Epub ahead of print 2019. DOI: 10.3390/s19132976.

[12] Scripting in Unity, https://unity.com/how-to/programming-unity.

[13] Robot cleaner 3D model, https://sketchfab.com/3d-models/doomba-4ad5546749f54041a382d6313cedc0fc.

ДОДАТОК А. СКРИПТ СЕНСОРУ

```
using System.Collections;
using System.Collections.Generic;
using DebugDrawingExtension;
using Unity.Collections;
using Unity.Jobs;
using UnityEngine;

namespace Robot.Sensors
{
    public class RobotSensor : MonoBehaviour
    {
        public delegate void SensorScanCb(RobotSensor sensor);

        public int raysCount = 100;
        public float maxRange = 5f;
        public float fov = 30f;
        public bool drawDebug = true;

        public int Id { get; private set; }
        public bool Enabled { get; private set; }
        public RayData? SensorHitRayData { get; private set; }

        private readonly List<SensorScanCb> _callbacks = new List<SensorScanCb>();
        private NativeArray<RayData> _raysData;

        public struct RayData
        {
            public Vector3 From { get; private set; }
            public Vector3 ForwardDir { get; private set; }
            public Vector3 RayDirection { get; private set; }
            public RaycastHit Hit;
            public readonly float Angle;
            public readonly float MaxRange;
            public readonly int SensorId;

            public RayData(float angle, float maxRange, int sensorId)
            {
                Angle = angle;
                MaxRange = maxRange;
                SensorId = sensorId;
                From = default;
                ForwardDir = default;
                RayDirection = default;
                Hit = default;
            }

            public void SetFrom(Transform transform)
            {
                From = transform.position;
                ForwardDir = transform.forward;
            }

            public void CalculateDirection()
            {
                var rotation = Quaternion.Euler(new Vector3(0, Angle, 0));
                RayDirection = rotation * ForwardDir;
            }
        }

        // Set-up job: Set up raycast commands in parallel.
        [Unity.Burst.BurstCompile]
        private struct SetupRaycasts : IJobParallelFor
        {
            // Output
            public NativeArray<RaycastCommand> Raycasts;
            public NativeArray<RayData> Data;

            public void Execute(int i)
            {
                var data = Data[i];
                data.CalculateDirection();
                Raycasts[i] = new RaycastCommand(data.From, data.RayDirection, data.MaxRange);
                Data[i] = data;
            }
        }
```

```
        private void Awake()
        {
            // init ray data
            _raysData = new NativeArray<RayData>(raysCount, Allocator.Persistent);
            Id = GetInstanceID();
            for (var i = 0; i < raysCount; i++)
            {
                var angle = fov / raysCount * i - fov / 2;
                _raysData[i] = new RayData(angle, maxRange, Id);
            }
        }

        private IEnumerator SensorUpdateData()
        {
            while (Enabled)
            {
                var rayCommands = new NativeArray<RaycastCommand>(raysCount, Allocator.TempJob);
                var rayHits = new NativeArray<RaycastHit>(raysCount, Allocator.TempJob);
                for (var i = 0; i < raysCount; i++)
                {
                    var data = _raysData[i];
                    data.SetFrom(transform);
                    _raysData[i] = data;
                }

                var setupRaycastsJob = new SetupRaycasts
                {
                    Raycasts = rayCommands,
                    Data = _raysData
                };

                var setupDependency = setupRaycastsJob.Schedule(raysCount, 1);
                var raycastDependency = RaycastCommand.ScheduleBatch(rayCommands, rayHits, 1,
setupDependency);

                raycastDependency.Complete();

                var prevDataHasValue = SensorHitRayData.HasValue;
                // find best ray
                SensorHitRayData = null;
                for (var i = 0; i < raysCount; i++)
                {
                    var hit = rayHits[i];
                    var data = _raysData[i];
                    data.Hit = hit;
                    _raysData[i] = data;

                    if (hit.distance > 0)
                    {
                        if (!SensorHitRayData.HasValue || SensorHitRayData.Value.Hit.distance >
hit.distance)
                            SensorHitRayData = data;
                    }
                }

                // draw debug gizmos
                if (drawDebug)
                {
                    DrawDebug();
                }

                // call CBs
                foreach (var cb in _callbacks)
                {
                    // call back only if sensor changed hit data
                    if (SensorHitRayData.HasValue || prevDataHasValue)
                        cb(this);
                }

                rayCommands.Dispose();
                rayHits.Dispose();

                yield return new WaitForSeconds(0.1f);
            }
        }

        private void DrawDebug()
        {
            var colorDebug = SensorHitRayData.HasValue ? Color.red : Color.green;
```

```
            if (SensorHitRayData.HasValue)
                DebugExtension.DebugPoint(SensorHitRayData.Value.Hit.point, Color.cyan, 0.5f);

            for (var i = 0; i < raysCount; i++)
            {
                Debug.DrawLine(_raysData[i].From,
                    _raysData[i].From + _raysData[i].RayDirection * _raysData[i].MaxRange,
                    colorDebug);
            }
        }

        public void Enable()
        {
            if (Enabled) return;

            Enabled = true;
            StartCoroutine(SensorUpdateData());
        }

        public void Disable()
        {
            if (!Enabled) return;

            Enabled = false;
        }

        private void OnEnable()
        {
            Enable();
        }

        private void OnDisable()
        {
            Disable();
        }

        private void OnDestroy()
        {
            _raysData.Dispose();
        }

        public void Subscribe(SensorScanCb cb)
        {
            _callbacks.Add(cb);
        }

        public void Unsubscribe(SensorScanCb cb)
        {
            _callbacks.Remove(cb);
        }
    }
}
```

ДОДАТОК Б. СКРИПТ КОНТРОЛЕРУ РОБОТА

```
using System.Collections;
using UnityEngine;

namespace Robot
{
    [RequireComponent(typeof(RobotMovementController))]
    public class RobotController : MonoBehaviour
    {
        public float batteryCapacity = 100f;

        /// <summary>
        /// Power consumption of robot per second
        /// </summary>
        public float powerConsumption = 0.1f;

        public Transform homePoint;
        public bool Enabled { get; private set; }

        public float BatteryLeft { get; set; }
        // public float BatteryLeft { get; private set; }

        public RobotMovementController MovementController { get; private set; }
        private bool _isConsumingPower;

        private void Awake()
        {
            BatteryLeft = batteryCapacity;
            MovementController = GetComponent<RobotMovementController>();
            if (homePoint == null)
                homePoint = transform;

            TurnOn();
        }

        public void TurnOn()
        {
            if (BatteryLeft <= 0) return;

            Enabled = true;
            MovementController.TurnOn();
            StartCoroutine(ConsumePower());
        }

        public void TurnOff()
        {
            Enabled = false;
            MovementController.TurnOff();
        }

        private IEnumerator ConsumePower()
        {
            if (_isConsumingPower || !Enabled) yield break;

            _isConsumingPower = true;
            while (BatteryLeft > 0 && Enabled)
            {
                BatteryLeft = Mathf.Max(BatteryLeft - powerConsumption, 0);
                if (BatteryLeft <= 20)
                {
                    StartCoroutine(MovementController.ProgramMoveToPoint(homePoint.position,
TurnOff));
                }

                yield return new WaitForSeconds(1);
            }

            _isConsumingPower = false;
            TurnOff();
        }
    }
}
```

ДОДАТОК В. СКРИПТ ПОШУКУ ШЛЯХУ

```
using JetBrains.Annotations;
using System.Collections.Generic;
using Unity.Mathematics;
using UnityEngine;

namespace Navigation
{
    public class PathNavigation : MonoBehaviour
    {
        private const int MoveStraightCost = 10;
        private const int MoveDiagonalCost = 15;

        public PathGrid<PathNode> Grid { get; private set; }
        public bool showGrid;
        public int2 gridSize = new int2(100, 100);
        public float gridCellSize = 1f;
        public Transform originPoint;

        private Vector3 _from;
        private Vector3 _to;
        private PathNode _pathNextNode;

        private void Awake()
        {
            if (originPoint == null)
                originPoint = transform;

            var origin = originPoint.position;
            var rotation = Quaternion.identity;
            var offset = -new Vector3(gridCellSize * gridSize.x / 2, 0, gridCellSize * gridSize.y /
2);

            Grid = new PathGrid<PathNode>(gridSize.x, gridSize.y, gridCellSize, origin, offset,
                (grid, x, y) => new PathNode(grid, x, y, gridCellSize, origin, rotation, offset));
            Grid.OnGridObjectChanged += (sender, args) =>
            {
                if (_to.magnitude > 0)
                {
                    if (_pathNextNode != null && _pathNextNode.Type != NodeType.Obstructed)
                        FindPath(_pathNextNode.WorldPosition, _to);
                    else if (_from.magnitude > 0)
                        FindPath(_from, _to);
                }
            };
        }

        private void OnDrawGizmos()
        {
            if (showGrid)
            {
                Grid?.DebugShowGrid(true);
                DisplayPath();
            }
        }

        public void AddObstacle(Vector3 position)
        {
            var node = Grid.FindNearestNode(position);
            if (node != null)
                node.Type = NodeType.Obstructed;
        }

        public Vector3 PathDirection(Vector3 currentLocation)
        {
            if (_pathNextNode == null)
                return Vector3.zero;

            currentLocation.y = _pathNextNode.WorldPosition.y;
            _from = currentLocation;
            var vector = _pathNextNode.WorldPosition - currentLocation;

            var minDistance = _pathNextNode.Size * 0.4f;
            if (_pathNextNode.NextNode == null)
                minDistance = 0.6f;

            if (Vector3.Distance(currentLocation, _pathNextNode.WorldPosition) < minDistance)
```

```
    {
        _pathNextNode = _pathNextNode.NextNode;
    }

    return vector;
}

[CanBeNull]
public List<PathNode> FindPath(Vector3 startWorldPosition, Vector3 endWorldPosition)
{
    Grid.GetXY(startWorldPosition, out var startX, out var startY);
    Grid.GetXY(endWorldPosition, out var endX, out var endY);

    var path = FindPath(startX, startY, endX, endY);
    if (path != null)
    {
        _from = startWorldPosition;
        _to = endWorldPosition;
        _pathNextNode = path[0];
    }
    else
    {
        _pathNextNode = null;
    }

    return path;
}

public void DisplayPath()
{
    if (_pathNextNode == null) return;
    var oldColor = Gizmos.color;

    var node = _pathNextNode;
    Gizmos.color = Color.green;
    while (node.NextNode != null)
    {
        Gizmos.DrawLine(node.WorldPosition, node.NextNode.WorldPosition);
        node = node.NextNode;
    }

    Gizmos.color = oldColor;
}

[CanBeNull]
private List<PathNode> FindPath(int startX, int startY, int endX, int endY)
{
    var startNode = Grid[startX, startY];
    var endNode = Grid[endX, endY];

    if (startNode == null || endNode == null)
    {
        // Invalid Path
        return null;
    }

    var openList = new List<PathNode> {startNode};
    var closedList = new List<PathNode>();

    for (var x = 0; x < Grid.Width; x++)
    {
        for (var y = 0; y < Grid.Height; y++)
        {
            var pathNode = Grid[x, y];
            if (pathNode != null)
            {
                pathNode.GCost = 99999999;
                pathNode.HCost = 0;
                pathNode.CameFromNode = null;
                pathNode.NextNode = null;
            }
        }
    }

    startNode.GCost = 0;
    startNode.HCost = CalculateDistanceCost(startNode, endNode);

    while (openList.Count > 0)
    {
```

```
                var currentNode = GetLowestFCostNode(openList);
                if (currentNode == endNode)
                {
                    // Reached final node
                    return CalculatePath(endNode);
                }

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

                foreach (var neighbourNode in GetNeighbourList(currentNode))
                {
                    if (closedList.Contains(neighbourNode)) continue;
                    if (neighbourNode.Type == NodeType.Obstructed)
                    {
                        closedList.Add(neighbourNode);
                        continue;
                    }

                    var tentativeGCost = currentNode.GCost + CalculateDistanceCost(currentNode,
neighbourNode);
                    if (tentativeGCost < neighbourNode.GCost)
                    {
                        neighbourNode.CameFromNode = currentNode;
                        neighbourNode.GCost = tentativeGCost;
                        neighbourNode.HCost = CalculateDistanceCost(neighbourNode, endNode);

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

            // Out of nodes on the openList
            return null;
        }

        private IEnumerable<PathNode> GetNeighbourList(PathNode currentNode)
        {
            var neighbourList = new List<PathNode>();

            // Down
            if (currentNode.Y - 1 >= 0)
                neighbourList.Add(Grid[currentNode.X, currentNode.Y - 1]);
            // Up
            if (currentNode.Y + 1 < Grid.Height)
                neighbourList.Add(Grid[currentNode.X, currentNode.Y + 1]);
            // Left
            if (currentNode.X - 1 >= 0)
                neighbourList.Add(Grid[currentNode.X - 1, currentNode.Y]);
            // Right
            if (currentNode.X + 1 < Grid.Width)
                neighbourList.Add(Grid[currentNode.X + 1, currentNode.Y]);

            // Left Down
            if (currentNode.X - 1 >= 0 && currentNode.Y - 1 >= 0)
                neighbourList.Add(Grid[currentNode.X - 1, currentNode.Y - 1]);
            // Left Up
            if (currentNode.X - 1 >= 0 && currentNode.Y + 1 < Grid.Height)
                neighbourList.Add(Grid[currentNode.X - 1, currentNode.Y + 1]);

            // Right Down
            if (currentNode.X + 1 < Grid.Width && currentNode.Y - 1 >= 0)
                neighbourList.Add(Grid[currentNode.X + 1, currentNode.Y - 1]);
            // Right Up
            if (currentNode.X + 1 < Grid.Width && currentNode.Y + 1 < Grid.Height)
                neighbourList.Add(Grid[currentNode.X + 1, currentNode.Y + 1]);

            return neighbourList;
        }

        private static List<PathNode> CalculatePath(PathNode endNode)
        {
            var path = new List<PathNode> {endNode};
            var currentNode = endNode;
            while (currentNode.CameFromNode != null)
            {
```

```
                path.Add(currentNode.CameFromNode);
                currentNode.CameFromNode.NextNode = currentNode;
                currentNode = currentNode.CameFromNode;
            }

            path.Reverse();
            return path;
        }

        private static int CalculateDistanceCost(PathNode a, PathNode b)
        {
            var xDistance = Mathf.Abs(a.X - b.X);
            var yDistance = Mathf.Abs(a.Y - b.Y);
            var remaining = Mathf.Abs(xDistance - yDistance);
            return MoveStraightCost * remaining + MoveDiagonalCost * Mathf.Min(xDistance,
yDistance);
        }

        private static PathNode GetLowestFCostNode(IReadOnlyList<PathNode> pathNodeList)
        {
            var lowestFCostNode = pathNodeList[0];
            for (var i = 1; i < pathNodeList.Count; i++)
            {
                if (pathNodeList[i].FCost < lowestFCostNode.FCost)
                {
                    lowestFCostNode = pathNodeList[i];
                }
            }

            return lowestFCostNode;
        }
    }
}
```