

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Навчання та використання штучного інтелекту
на прикладі покрокової гри-стратегії»**

Виконав студент групи ІПЗм-23-1
НАДВОРНИЙ Максим Юрійович

Керівник роботи:
ЧИЖМОТРЯ Олена Геннадіївна

Рецензент:
ЛОБАНЧИКОВА Надія Миколаївна

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«01» жовтня 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: НАДВОРНИЙ Максим Юрійович

Керівник роботи: ЧИЖМОТРЕ Олена Геннадіївна

Тема роботи: **«Навчання та використання штучного інтелекту на прикладі покрокової гри-стратегії»**, затверджена Наказом закладу вищої освіти від «1» жовтня 2024 р., №520/с

Вихідні дані для роботи: **штучний інтелект, що використовується для автоматизованого управління юнітами в покроковій гри-стратегії, зокрема алгоритми навчання на основі евристик та біхейвіорів.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О.Г.	02.10.2024	02.10.2024
2	Чижмотря О.Г.	02.10.2024	02.10.2024
3	Чижмотря О.Г.	02.10.2024	02.10.2024

РЕФЕРАТ

Кваліфікаційна робота складається з пояснювальної записки та програмного продукту - гри. Пояснювальна записка складається з вступу, трьох розділів та списку використаної літератури. Пояснювальна записка до кваліфікаційної магістерської роботи містить 76 сторінок, 21 ілюстрацію, 2 таблиці, 1 додаток та 40 посилань на зовнішні джерела.

Метою роботи є розробка покрокової гри стратегії з використанням штучного інтелекту від першої особи за допомогою рушія Unity3D.

В роботі визначено основні завдання на розробку системи, проаналізовано аналоги розробленої системи. Наведено сценарії роботи програмного комплексу, загальна структура програмного комплексу, опис алгоритмів взаємодії окремих модулів системи, опис об'єктної структури системи (діаграма класів). Також наводяться результати тестування системи.

КЛЮЧОВІ СЛОВА: ГРА, ПОКРОКОВА СТРАТЕГІЯ, AI, ІГРОВИЙ РУШІЙ.

ABSTRACT

The thesis consists of an explanatory note and a software product - a game. The explanatory note includes an introduction, three chapters, and a list of references. The explanatory note for the master's qualification thesis contains 76 pages, 21 illustrations, 2 tables, 1 appendix, and 40 references to external sources.

The aim of the work is to develop a turn-based strategy game with first-person artificial intelligence (AI) using the Unity3D engine.

The work defines the main tasks for system development, analyzes analogs of the developed system, provides scenarios for the software system operation, describes the general structure of the software system, details the interaction algorithms between individual system modules, and outlines the object structure of the system (class diagram). Additionally, the results of system testing are presented.

KEYWORDS: GAME, TURN-BASED STRATEGY, AI, GAME ENGINE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ ІГРОВИХ ДОДАТКІВ	10
1.1. Використання штучного інтелекту в ігровій індустрії	10
1.2. Існуючі аналоги програмного продукту та їх особливості	12
1.3. Постановка задачі	18
1.4. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення.....	19
Висновки до першого розділу	24
РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ	26
2.1. Визначення варіантів використання та структури системи	26
2.2 Реалізація функціоналу додатку.....	39
Висновки до другого розділу.....	42
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ДОДАТКОМ	43
3.1. Інтерфейс та порядок роботи з програмним додатком.....	43
3.2. Тестування роботи системи	49
Висновки до третього розділу	51
ВИСНОВКИ.....	52
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	54
ДОДАТКИ.....	58

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Unity – ігровий рушій Unity.

C# - мова програмування C#.

ЛКМ – ліва кнопка миші.

AI - artificial intelligence

ШІ – штучний інтелект

ВСТУП

Актуальність теми. Штучний інтелект (ШІ) є важливою складовою багатьох сучасних ігор, зокрема стратегій, де важливою є здатність системи приймати рішення на основі аналізу поточної ситуації на полі бою. У покрокових стратегіях це стає особливо актуальним, оскільки правильний вибір дій кожного юніта значною мірою визначає результат гри. Використання методів машинного навчання для покращення стратегій, адаптивності та прогнозування поведінки ботів є ключовим напрямом для підвищення інтелектуальної складності таких ігор. Такий підхід дозволяє створити більш реалістичних супротивників, здатних адаптуватися до дій гравця.

Мета дослідження. Метою даного дослідження є розробка та впровадження алгоритмів навчання ШІ для покрокової гри-стратегії, що дозволить ботам адаптувати свою поведінку на основі аналізу евристик та біхейвіорів, а також прогнозувати стратегії гравця.

Для досягнення поставленої мети необхідно вирішити такі основні завдання:

1. Дослідити особливості покрокових стратегічних ігор та основні підходи до створення ігрових ботів.
2. Розглянути сучасні методи навчання ШІ, зокрема методи навчання з підкріпленням та поведінкове моделювання.
3. Розробити алгоритми адаптивної поведінки ботів, які враховують аналіз евристик та біхейвіорів.

Об'єкт дослідження. Об'єктом дослідження є штучний інтелект, що використовується для автоматизованого управління юнітами в покроковій гри-стратегії, зокрема алгоритми навчання на основі евристик та біхейвіорів.

Предмет дослідження. Предметом дослідження є методи навчання та алгоритми прийняття рішень ШІ для покрокових стратегій, зокрема методи

епсілон-жадібної вибірки та Softmax, що застосовуються для вибору оптимальних дій юнітів.

Методи аналізу. Для досягнення мети будуть використовуватися наступні методи аналізу:

1. Аналіз алгоритмів машинного навчання для покрокових стратегій.
2. Моделювання поведінки ІІІ за допомогою евристик та біхейвіорів.
3. Аналіз ефективності алгоритмів навчання через тестування на реальних прикладах.

Інформаційна база дослідження. Інформаційною базою дослідження є:

1. Документація щодо алгоритмів машинного навчання та адаптивного штучного інтелекту.
2. Ігрові движки та системи для розробки покрокових стратегій, зокрема Unity.

Дані тестових ігор для тренування та аналізу ефективності ІІІ.

Наукова новизна. Наукова новизна дослідження полягає у впровадженні інноваційних методів навчання ІІІ для покрокових ігор-стратегій, зокрема у використанні модифікацій епсілон-жадібного підходу та Softmax для адаптивного вибору дій. Запропоновані методи дозволяють ботам враховувати не лише ситуацію на полі бою, а й прогнозувати можливі стратегії гравця, що підвищує рівень інтерактивності та складності гри.

Практичне значення дослідження полягає у створенні інтелектуальної системи штучного інтелекту для покрокової гри-стратегії, яка може бути застосована як в комерційних проєктах, так і у дослідницьких цілях для розробки адаптивного ІІІ в ігровій індустрії. Реалізовані алгоритми на основі евристик, біхейвіорів та навчальних даних забезпечують підвищення гнучкості та адаптивності ботів.

Апробація. За темою кваліфікаційної магістерської роботи було опубліковано дві тези конференції[1, 2].

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ ІГРОВИХ ДОДАТКІВ

1.1. Використання штучного інтелекту в ігровій індустрії

Штучний інтелект (AI) відіграє значну роль у сучасній ігровій індустрії. Його застосування дозволяє створювати ворогів, союзників і цілі системи, які ведуть себе реалістично і реагують на дії гравця, занурюючи його у глибший ігровий процес. Це робить AI важливим компонентом у багатьох жанрах ігор, зокрема у стратегіях, шутерах, пригодницьких іграх та симуляторах.

Одним із ключових завдань AI в іграх є створення супротивників, які можуть аналізувати дії гравця та адаптувати свої стратегії. Наприклад, у покрокових стратегіях штучний інтелект здатний розробляти плани, що враховують декілька можливих сценаріїв розвитку подій. Такий підхід дозволяє підвищити рівень виклику для гравця та зробити гру більш насиченою і непередбачуваною.

Для розробки AI в іграх існує кілька основних методів. Скриптовий підхід передбачає використання фіксованих наборів правил, які визначають поведінку ворогів. Це найпростіший варіант реалізації, але він обмежений у гнучкості, адже дії ворогів стають передбачуваними. Більш складним і ефективним є підхід із використанням дерев прийняття рішень, де кожен вузол дерева представляє умову, а гілки відповідають діям, що їх виконує AI. Такий метод забезпечує ширшу варіативність, але вимагає оптимізації з ростом кількості умов.

Алгоритми пошуку шляхів, наприклад, A*, широко застосовуються у стратегіях для планування переміщень юнітів. Вони допомагають AI швидко знаходити найкоротший шлях між точками, враховуючи перешкоди на карті. Ще одним перспективним підходом є використання машинного навчання та підкріплювального навчання, коли AI самостійно вдосконалює свої навички,

аналізуючи результати своїх дій. Наприклад, у покрокових стратегіях AI може з кожним ходом коригувати свою стратегію, враховуючи програші або перемоги в попередніх партіях.

Для досягнення високої ефективності часто застосовують гібридні моделі. Вони поєднують кілька методів, наприклад, алгоритми пошуку шляхів з машинним навчанням, щоб забезпечити AI більш комплексний підхід до ухвалення рішень. Нейронні мережі, своєю чергою, використовуються для розпізнавання шаблонів і більш складних ігрових сценаріїв. Наприклад, у відомій грі "AlphaGo" штучний інтелект завдяки нейронним мережам зміг перевершити людських чемпіонів у грі го.

У покрокових стратегіях AI повинен враховувати кілька ключових аспектів. Аналіз ризиків допомагає оцінювати ймовірність успіху кожного ходу і вибирати найбільш ефективний варіант дій. Адаптивність дозволяє AI змінювати свою тактику, залежно від того, як грає користувач. Крім того, важливим є прогнозування дій гравця, коли AI будує свої плани, враховуючи можливі сценарії поведінки суперника.

Завдяки AI покрокові стратегії стають більш складними та динамічними. Реалістичність і варіативність поведінки супротивників сприяють підвищенню цікавості до гри. Гравець стикається з непередбачуваними діями AI, що вимагає постійного вдосконалення своїх навичок і стратегій. У результаті це забезпечує більш захоплюючий ігровий досвід.

Штучний інтелект у сучасних покрокових стратегіях відкриває широкі можливості для реалізації складних сценаріїв, адаптації до гравців і створення реалістичних симуляцій [1]. Його впровадження дозволяє не тільки автоматизувати рутинні завдання, але й створити систему, яка може ефективно імітувати людську логіку та поведінку. Таким чином, AI стає ключовим інструментом у розробці ігор нового покоління, сприяючи їх унікальності та популярності.

1.2. Існуючі аналоги програмного продукту та їх особливості

ХСОМ.

Одним з прикладів успішного застосування стратегій з елементами штучного інтелекту є серія ігор ХСОМ. Ці ігри стали класикою жанру тактичних стратегій, в яких поєднуються елементи управління військовими підрозділами, ресурсами та науковими дослідженнями в умовах війни з інопланетними силами.

Серія ігор ХСОМ є тактичною стратегією в реальному часі з паузою, де гравець бере на себе роль командира елітного підрозділу, що протистоїть інопланетній загрозі. Головною метою є оборона планети від інопланетного вторгнення за допомогою різноманітних військових юнітів, технологій та стратегічних рішень.

Основними аспектами гри є кілька ключових елементів, які формують унікальність ігрового процесу. Центральним компонентом є тактичний бойовий процес, де кожен бій реалізується у форматі покрокової стратегії. Гравець отримує можливість керувати юнітами на полі бою, обираючи їхні дії з урахуванням позицій ворогів, рельєфу місцевості та доступних ресурсів. Такий підхід дозволяє максимально зануритися в стратегічне планування та створює відчуття контролю над кожним аспектом бою.

Іншою важливою складовою є інтелект віртуальних супротивників. У грі, подібній до ХСОМ, AI противників діє як динамічна система, яка постійно аналізує дії гравця. Віртуальні вороги не лише реагують на поточну ситуацію, але й пристосовуються до змін у стратегії гравця, змінюючи свою поведінку в залежності від розвитку подій на полі бою. Це забезпечує високу варіативність ігрових сценаріїв і стимулює гравців шукати нові підходи до вирішення тактичних завдань.

Ще одним важливим аспектом є ресурсне управління. Гравець не лише бере участь у боях, але й займається побудовою та розвитком бази. Цей процес включає дослідження нових технологій, збір ресурсів, які є необхідними для виробництва зброї, обладнання та утримання бази, а також

тренування юнітів. Грамотне управління цими ресурсами впливає на успіх у майбутніх боях, створюючи додатковий рівень складності та стратегічного планування.



Рис. 1.1. Тактичний бойовий інтерфейс

У XCOM штучний інтелект противників розвивається за допомогою складної системи алгоритмів, що базуються на попередньому досвіді та наведених сценаріях. Кожен ворог має свою поведінку, яка враховує кілька факторів, таких як відстань до цілі, наявність укриттів, поточний стан здоров'я та інші змінні. Це дозволяє створити відчуття інтелектуального опонента, що змінює стратегію залежно від дій гравця.

Виконаємо порівняння XCOM та кваліфікаційної роботи.

Таблиця 1.1

Порівняння ігор

Характеристика	XCOM	Кваліфікаційна робота
Тип гри	Тактична стратегія	Тактична стратегія з AI
Механіка бою	Покровоа стратегія	Покрокова стратегія
Евристики	Присутні в AI для адаптації	Є система евристик для вибору

		поведінки
Біхейвіори	Адаптивні для кожного типу ворога	Використовуються для вибору екшнів
Штучний інтелект	Адаптивний до дій гравця	Навчання через серіалізацію ходів та прогнозування
Ресурсне управління	Наукові дослідження, ресурси	Немає прямого аналога, але є управління юнітами та їх характеристиками

Gears Tactics.

Це тактична рольова гра в серії Gears of War, яка поєднує елементи покрокової стратегії з глибоким сюжетним наповненням і складною бойовою механікою. Вона фокусується на командній тактиці та взаємодії між юнітами в умовах постапокаліптичного світу, де гравець керує групою солдатів, що борються з інопланетними ворогами. У порівнянні з іншими іграми жанру тактики, Gears Tactics має особливу увагу до деталей бою, керування персонажами та реалізації сюжетних елементів.

Gears Tactics є покроковою стратегією з елементами тактичного бою, де гравець має контролювати групу персонажів, надаючи їм різні тактичні завдання для боротьби з ворогами. Гра акцентує увагу на швидких та інтенсивних боях, де кожен хід має бути ретельно продуманий.

Гра побудована на кількох ключових аспектах, які формують її основну механіку та забезпечують глибокий ігровий досвід. Перш за все, тактичний рівень гри передбачає управління відразу кількома персонажами, кожен з яких володіє унікальними навичками та здібностями. Це дозволяє створювати індивідуальні стратегії, де кожен член команди відіграє свою

роль, впливаючи на загальний хід бою. Таке різноманіття сприяє глибшому зануренню в тактичне планування та прийняття рішень.

Управління персонажами є ще одним важливим аспектом гри. Кожен герой має можливість отримувати досвід, що відкриває доступ до нових здібностей, покращує характеристики та впливає на їхню ефективність у бою. Крім того, спорядження персонажів можна налаштовувати, змінюючи зброю, броню та інші предмети, що дозволяє адаптувати команду під конкретні виклики та бойові ситуації.

Бойова механіка реалізована через використання тактичних карт, де гравець отримує можливість переміщувати своїх персонажів, використовуючи елементи навколишнього середовища. Наприклад, укриття допомагають знизити ризик отримання пошкоджень, а стратегічне розташування дозволяє атакувати ворогів з максимальною ефективністю. Можливість взаємодії з середовищем додає динаміки та варіативності кожному бою.

Супротивники, керовані AI, також відіграють важливу роль. Ворожі юніти використовують різні тактики та стратегії, створюючи виклик навіть для досвідчених гравців. Для забезпечення балансу гра пропонує можливість налаштування складності, що дозволяє адаптувати гру для новачків і досвідчених стратегів.



Рис. 1.2. Тактичний бій

Gears Tactics вирізняється кількома ключовими аспектами, які забезпечують її популярність та глибину ігрового процесу. Перш за все, гра пропонує захопливу сюжетну лінію, що базується на багатому всесвіті серії Gears of War. Завдяки цьому гравці можуть зустріти знайомих персонажів, дізнатися більше про події світу гри та відчути сильне емоційне залучення до сюжету. Цей аспект робить гру не лише тактичною стратегією, але й інтерактивною історією, що тримає у напрузі від початку до кінця.

Розвиток персонажів є ще одним вагомим елементом гри. Гравцям надається можливість формувати унікальні команди, адаптуючи їх під свій стиль гри. Вдосконалення героїв включає розвиток їхніх навичок, доступ до нового спорядження та створення індивідуальних тактичних комбінацій. Така система дозволяє кожному гравцеві експериментувати з різними стратегіями та підходами, що додає глибини ігровому процесу.

Взаємодія з оточенням також відіграє важливу роль у грі. Використання укриттів, стратегічне розташування на полі бою та врахування особливостей ландшафту відкривають додаткові можливості для тактичних маневрів. Гравець має змогу продумувати розташування своїх юнітів для

досягнення максимальної ефективності в атаці чи обороні, що додає грі інтелектуального виклику.

Інтенсивність боїв у Gears Tactics робить гру надзвичайно динамічною. Кожна битва сповнена напруження та вимагає від гравця швидких і точних рішень. Поєднання високої швидкості дій та тактичної глибини створює унікальний ігровий досвід, який тримає увагу гравця на кожному етапі битви.

Виконаємо порівняння Gears of War та кваліфікаційної роботи.

Таблиця 1.2

Порівняння ігор

Характеристика	Gears of War	Кваліфікаційна робота
Тип гри	Тактична стратегія	Тактична стратегія з AI
Механіка бою	Покровоа стратегія	Покрокова стратегія
Евристики	Відсутні	Є система евристик для вибору поведінки
Біхейвіори	Відсутні	Використовуються для вибору екшнів
Штучний інтелект	Вороги діють за сценаріями	Навчання через серіалізацію ходів та прогнозування
Ресурсне управління	Оновлення спорядження, навички юнітів	Немає прямого аналога, але є управління юнітами та їх

		характеристиками
Кастомізація персонажів	Глибока кастомізація спорядження	Відсутня
Прогресія	Сюжетно орієнтовані місії	Поступове вдосконалення AI і механік через адаптацію
Графічне оформлення	Високий рівень деталізації	Залежить від реалізації

1.3. Постановка задачі

Гра створена в жанрі покрокової стратегії з тактичними боями на сітчастій карті. Основною метою гравця є знищення ворогів, керуючи власними персонажами та використовуючи їхні унікальні можливості. Інтерфейс гри включає індикатори дій, візуалізацію доступних рухів на сітці та чітку послідовність ходів.

Основні режими гри:

1. Ручний режим (гравець самостійно контролює персонажів, кожен із яких має набір дій):

- Рух: пересування на певну кількість клітинок за допомогою A*-алгоритму.
- Стрільба: атака на відстані, залежно від лінії видимості.
- Кидання гранати: завдання групового урону у радіусі вибуху.
- Обертання: Юніт обертається на 360 градусів.
- Ближній бій мечем: точкова атака ворога поруч.

Закінчення очок дій завершує хід юніта, і після натискання на кнопку “Завершити хід” ініціатива переходить до супротивника.

2. Автоматичний режим:

- Штучний інтелект контролює як персонажів гравця, так і ворогів. Поведінка базується на аналізі ігрових умов, евристик та моделей поведінки.

Основні елементи:

1. Сітчаста карта: чітка структура поля бою допомагає прогнозувати тактичні маневри.
2. Юніти: гравець має більше дій ніж ворог, таким чином у гравця є перевага над ворогами (кидання гранати, обертання на 360 градусів).
3. Візуалізація ходів: активні дії підсвічуються, а доступні маршрути відображаються як комірки на мапі.

Гра завершується коли загинули всі юніти гравця або ворога. Основна мета гри знищити ворожі юніти.

1.4. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Сьогодні серед найпопулярніших рушіїв для розробки ігор можна виділити Unity3D, UE4, Marmalade та Cocos2D, кожен з яких має свої переваги. Однак, враховуючи потребу в розробці кроссплатформної гри з використанням фізичного рушія, Unity займає лідируючу позицію за актуальними дослідженнями[15].

Рушій Unity є потужним інструментом для створення ігор та інтерактивних додатків, надаючи розробникам можливості для створення вражаючої графіки, фізики, анімації, звукових ефектів та ігрового геймплею. Він особливо популярний завдяки своїм численним перевагам, серед яких можна відзначити хрестоматійність[16]. Unity має велику та активну спільноту, яка підтримує розробників, надаючи документацію, плагіни та ресурси, що робить його доступним для вивчення та командної роботи.

Ще однією важливою перевагою є кроссплатформеність. Unity підтримує різноманітні платформи, включаючи ПК, мобільні пристрої,

консольні платформи, веб-браузери та віртуальну реальність[17]. Це дозволяє розробникам створювати проекти, які можуть бути без труднощів адаптовані під різні пристрої без переписування коду.

Рушій також пропонує широкий функціонал, включаючи вбудовані інструменти для графіки, фізики, анімації, штучного інтелекту та звуку, що дозволяє швидко розробляти та прототипувати ігри[39]. Окрім цього, Unity має інтегроване середовище розробки (IDE), яке забезпечує зручний інтерфейс для написання, налагодження та управління проектами[18]. Завдяки цьому розробники можуть ефективно працювати зі сценами, об'єктами, компонентами та скриптами.

Важливою рисою Unity є його розширюваність[40]. Використовуючи плагіни та скрипти, розробники можуть додавати нові функціональні можливості до своїх проектів[19]. Підтримка мови програмування C# дає змогу створювати власні скрипти для ігрової логіки, використовуючи потужні інструменти для програмування.

Однією з головних переваг Unity є швидкість розробки. Інструменти, такі як перетягування та розміщення об'єктів, візуальний редактор анімації, а також вбудовані фізичні двигуни, дозволяють розробникам ефективно прототипувати, тестувати та вдосконалювати свої ідеї, що особливо важливо в контексті ігрової розробки[20].

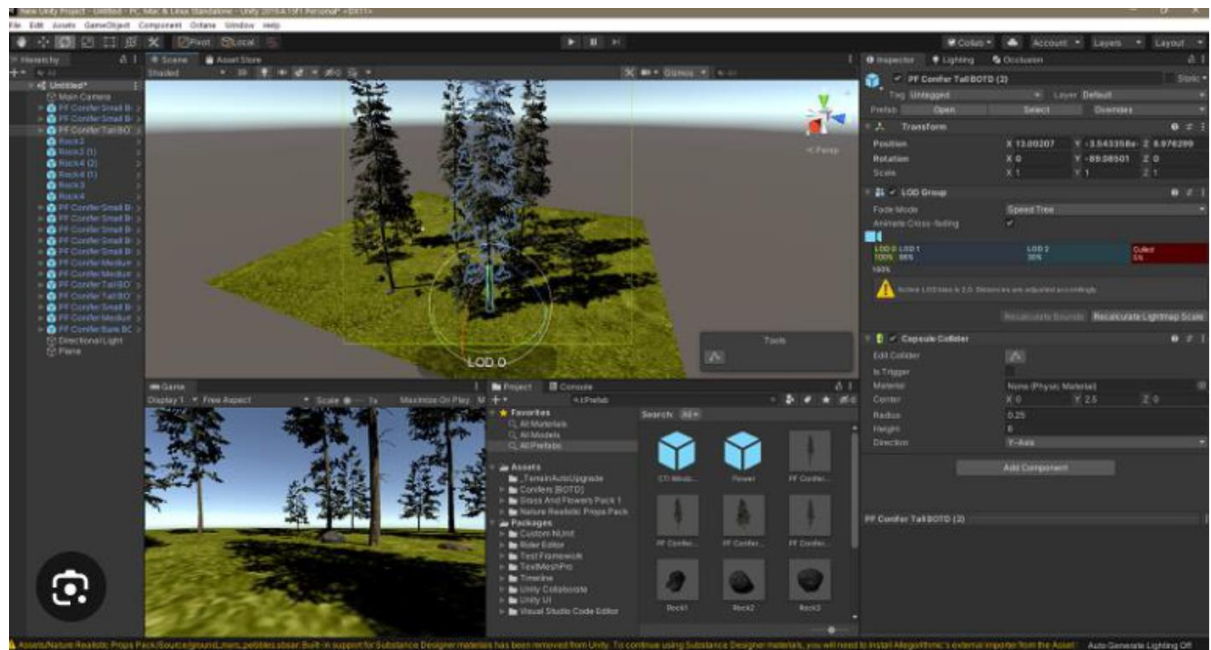


Рис.1.3. Unity Editor

Інтегроване середовище розробки, таке як Visual Studio Code (VS Code), має кілька значних переваг. Однією з основних є його зручний та інтуїтивно зрозумілий інтерфейс користувача. Це дозволяє розробникам зосереджуватись на написанні коду, не відволікаючись на складні налаштування. Панелі розташовані ергономічно, є швидкі переходи між файлами, а також інші функції, що роблять процес розробки зручним та ефективним [21].

Окрім цього, VS Code підтримує велику кількість розширень, які дозволяють користувачам налаштовувати редактор під свої потреби. Розширення доступні для різних мов програмування, фреймворків та інструментів розробки, що робить VS Code універсальним у використанні [22].

Також, важливою перевагою є вбудована підтримка системи контролю версій Git, що дозволяє розробникам відстежувати зміни в коді, виконувати коміти та керувати гілками. Це значно спрощує процес командної роботи та співпраці [23].

Ще однією значущою характеристикою є потужні інструменти для налагодження коду. VS Code надає можливість точкового зупинення коду,

крокування по програмах, перевірки значень змінних, що дозволяє швидко знаходити та виправляти помилки [24].

У редакторі є й інші корисні функції, такі як автодоповнення коду, підсвічування синтаксису, перехід до визначення функцій та змінних, а також швидкий пошук і заміна тексту. Ці можливості допомагають розробникам працювати швидше та ефективніше [25].

Завдяки великій та активній спільноті, VS Code отримує постійну підтримку, документацію та нові плагіни, що значно полегшує роботу з редактором. Користувачі можуть легко знаходити відповіді на запитання та використовувати корисні ресурси [26].

Також VS Code можна інтегрувати з іншими інструментами розробки, такими як системи керування проектами, лінери, тестувальні фреймворки, що дозволяє створювати ефективне середовище для роботи над проектами.

Blender 3.5 — це потужний 3D-редактор, який має багато переваг, що роблять його популярним серед професіоналів та новачків у сфері 3D-графіки та анімації. Однією з ключових переваг є безкоштовність. Blender є програмним забезпеченням з відкритим вихідним кодом, що дозволяє користувачам безкоштовно його використовувати без обмежень на ліцензії та витрати. Крім того, користувачі можуть змінювати програму та налаштовувати її під свої конкретні потреби, оскільки вихідний код доступний для всіх [27].

Ще однією важливою перевагою є кросплатформеність. Blender підтримує різні операційні системи, такі як Windows, macOS і Linux. Це дозволяє використовувати Blender на будь-якому комп'ютері, незалежно від операційної системи, що забезпечує широке охоплення користувачів по всьому світу [28].

Blender також відомий своєю комплексністю та функціональністю. Він надає вражаючий набір інструментів для створення 3D-графіки, охоплюючи такі аспекти, як моделювання, анімація, текстурювання, освітлення, рендеринг, симуляція фізики та інші важливі елементи. Це

дозволяє користувачам виконувати всю необхідну роботу в одному програмному забезпеченні, без потреби використовувати додаткові програми [29].

Іншою важливою характеристикою є розширюваність Blender. Користувачі можуть додавати розширення, які додають нові інструменти, ефекти, шаблони та інші ресурси. Це дозволяє налаштовувати Blender під індивідуальні потреби користувачів та розширювати його можливості [30].

Активна спільнота Blender є ще однією перевагою. Велика та активна спільнота розробників, художників і користувачів надає підтримку, відповіді на питання та ресурси для навчання, що дозволяє вдосконалювати навички використання програми [31].

Blender також надає розширені можливості для моделювання. Він підтримує різні методи створення 3D-моделей, такі як полігонове, скульптурне, реберне та поверхнєве моделювання. Це дозволяє створювати складні форми і детальні текстури для проектів [32].

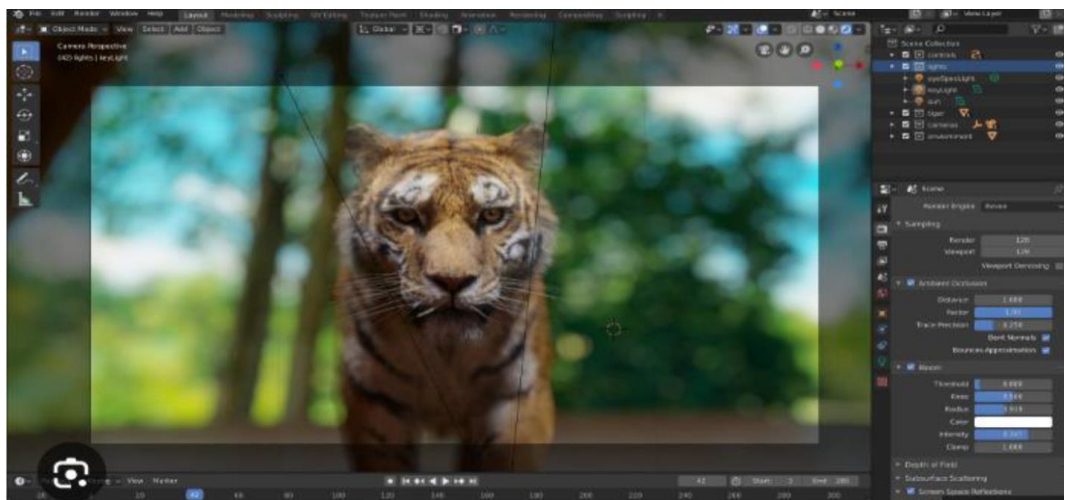


Рис. 1.4. Blender 3.5

C# є однією з найбільш популярних мов програмування завдяки своїм численним перевагам. Однією з основних є простота вивчення мови. Завдяки логічній структурі та простому синтаксису, C# є відносно легким для освоєння, особливо для початківців[38]. Багато аспектів цієї мови схожі з іншими популярними мовами, такими як C++ і Java, що робить її вивчення корисним для роботи з ними в майбутньому [33].

Ще однією важливою особливістю C# є підтримка об'єктно-орієнтованого програмування (ООП). Це дозволяє розробникам організовувати код у вигляді класів та об'єктів, що підвищує модульність і сприяє повторному використанню коду. Таке структурування значно полегшує процес розширення програми в майбутньому, забезпечуючи кращу організацію та підтримку великого коду [34].

Мова C# є основною для розробки програмного забезпечення на платформі .NET. .NET пропонує безліч бібліотек і фреймворків, які значно полегшують розробку додатків[37]. Завдяки цій платформі, розробники отримують доступ до потужних інструментів і можливостей для створення різноманітних програмних продуктів, що розширюють функціональність створених додатків [35].

Також важливою перевагою C# є його мультиплатформеність. Мова підтримує розробку додатків, які можуть працювати на різних операційних системах, таких як Windows, macOS і Linux. Це дозволяє розробникам створювати програми, які є сумісними з різними платформами, використовуючи єдину мову програмування [36].

Висновки до першого розділу

Проведений аналіз предметної області дозволив визначити основні аспекти гри у жанрі тактичних стратегій. Огляд наявних аналогів, а також виявлення протиріч між існуючими рішеннями та потребами, показав, що головними функціями майбутньої системи мають бути: адаптивний штучний інтелект, який здатний прогнозувати дії гравця; покрокова механіка бою; гнучка система евристик для вибору поведінки юнітів; а також зрозумілий і зручний графічний інтерфейс.

Особливу увагу приділено механізмам навчання AI, які забезпечать високий рівень автономності та адаптації до тактичних сценаріїв. Бажаними функціями також є можливість управління характеристиками юнітів,

додавання нових біхейвіорів для ворогів, а також підтримка гравцем складних стратегій через доступні й інтуїтивно зрозумілі інструменти.

Визначившись з оптимальною архітектурою, обрано необхідний набір програмних інструментів для реалізації поставлених задач: рушій з підтримкою AI-алгоритмів, редактор рівнів для створення ігрового світу, а також бібліотеки для візуалізації та роботи з алгоритмами штучного інтелекту.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

2.1. Визначення варіантів використання та структури системи

Система покрокової гри-стратегії створюється з метою максимального занурення гравця у процес стратегічного планування, аналізу та взаємодії із противниками, які керуються штучним інтелектом[3].

Вимоги користувачів:

Користувачі мають доступ до наступних функцій:

1. Старт гри;
2. Пауза гри;
3. Рух по карті;
4. Стрільба;
5. Удар мечем;
6. Кидання гранати;
7. Оберт навколо тіла;
8. Знищення перешкод гранатою;
9. Бої проти AI ворогів.

Функціональні вимоги:

1. Гемплей: користувач матиме можливість знищувати ворогів в покроковій грі стратегії для цього можна використовувати стрільбу, кидання гранати, удар мечем, також гранатою можна знищувати перешкоди.

2. Не потрібно перебувати онлайн: користувач може грати без підключення до Інтернету в довільний момент часу.

Нефункціональні вимоги:

1. Сприйняття:
 - Час, потрібний для навчання роботи з програмою для звичайних користувачів — 40 хвилин, для досвідчених — 15 хвилин.

- Час відповіді системи для звичайних запитів не повинен перевищувати 0.016 секунди, а для більш складних запитів — 0.03 секунди.
- Інтерфейс симулятора повинен бути інтуїтивно зручним для користувача та не вимагати від нього додаткової підготовки.
- Доступність - час, потрібний для обслуговування системи не повинен перевищувати 10 % від загального часу роботи.
- Середній час безперервної роботи — 6 годин.
- Максимальна норма помилок та дефектів в роботі програми - 1 помилка на 1000 запитів користувача.

2. Продуктивність. Гра повинна підтримувати безперебійну роботу користувача.

3. Можливість експлуатації:

- Масштабування – після оновлення все повинно працювати без збоїв, не мають виникати жодні помилки при розширенні гри.
- Оновлення версій – оновлення версій повинно здійснюватися за допомогою патчу.



Рис 2.1. Варіанти використання

Розглянемо класи та за що вони відповідають:

На рис зображено діаграму класів, що містить різні класи поведінки для AI-персонажів. Кожен файл описує певну модель поведінки для персонажа.

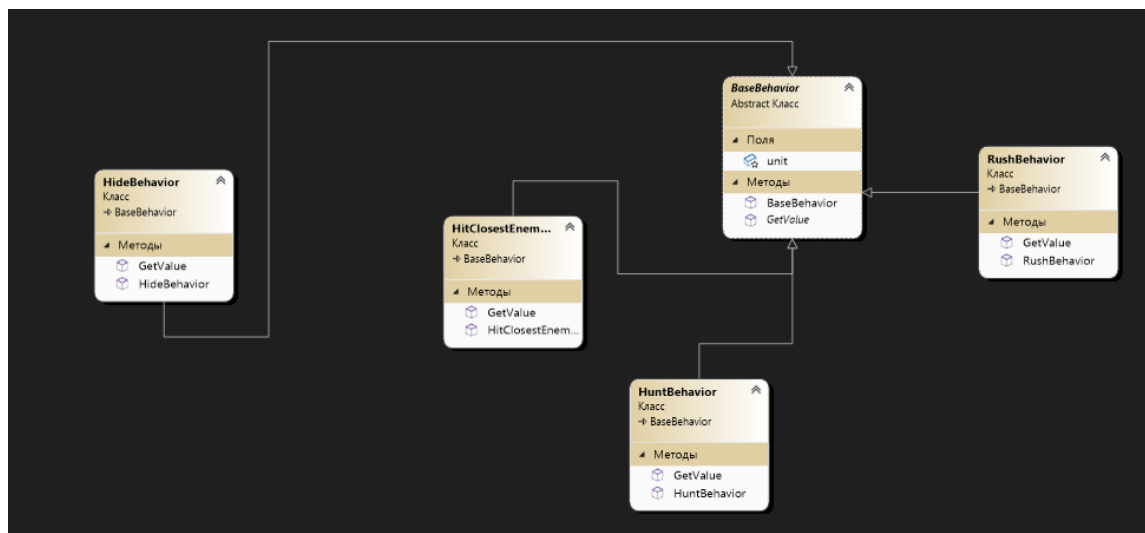


Рис.2.2 Діаграма класів поведінки для AI – персонажів.

BaseBehavior.cs — Базовий клас поведінки:

GetValue() - абстрактний метод, який повинен бути реалізований у всіх похідних класах

HideBehavior.cs — Поведінка, яка дозволяє персонажу ховатися. Може використовуватися, коли AI вирішує, що необхідно уникнути конфлікту:

GetValue() - визначає значення (пріоритет) дії та вибирає найбільш підходящу дію, яку потрібно виконати

HitClosestEnemyBehavior.cs — Поведінка атаки найближчого ворога. Підходить для агресивного AI, який фокусується на найближчих супротивниках[12]:

GetValue() - Визначає, наскільки підходяща дана поведінка у поточній ситуації (пріоритет у діапазоні від 0 до 50) і вибирає відповідну дію (action).

HuntBehavior.cs — Поведінка переслідування. Використовується для AI, який активно шукає ворога або мету для нападу:

GetValue() - Визначає пріоритет поточної поведінки та вибирає відповідну дію для об'єкта Unit

RushBehavior.cs — Поведінка швидкої атаки або ривка до ворога. Це може бути агресивний стиль, де AI швидко наближається до цілі:

GetValue() - Визначає пріоритет поточної поведінки в діапазоні від 0 до 50 і вибирає відповідну дію (action).

На рис зображено діаграму класів, що відповідає за евристичні методи — різні способи оцінки ситуації, щоб допомогти AI прийняти рішення

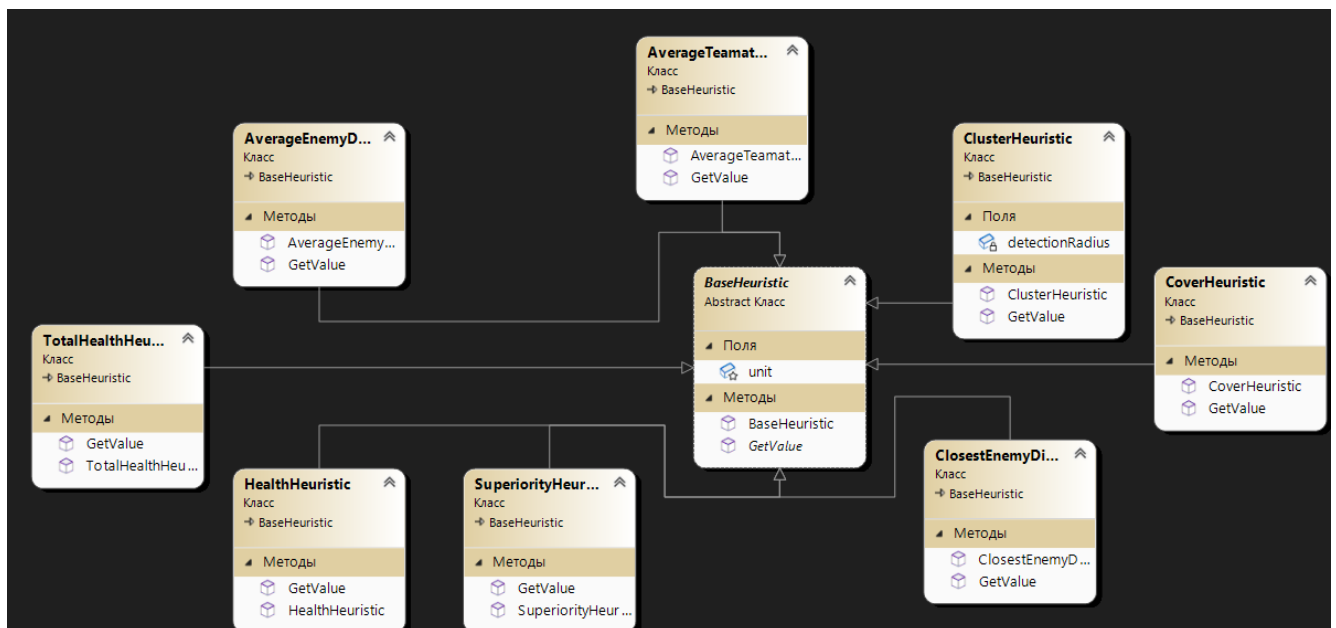


Рис 2.3. Діаграма класів евристик

AverageEnemyDistanceHeuristic.cs — Оцінка середньої відстані до ворогів. Може використовуватися, щоб AI розумів, наскільки він близький до групи[10]:

GetValue() - Повертає середню відстань до всіх супротивників у вигляді цілого числа

AverageTeamatesDistanceHeuristic.cs — Середня відстань до союзників. Допомагає AI оцінити, наскільки далеко він від своєї команди:

GetValue() - Повертає середню відстань до всіх союзників у вигляді цілого числа

BaseHeuristic.cs — Базовий клас для евристичних підходів. Інші евристики можуть успадковувати цей клас, щоб уніфікувати методи оцінки.

`ClosestEnemyDistanceHeuristic.cs` — Визначає відстань до найближчого ворога, що корисно для рішень про напад чи втечу[11]:

`GetValue()` - Повертає відстань до найближчого ворога, округлену до цілого числа.

`ClusterHeuristic.cs` — оцінює скупчення ворогів у певному радіусі. Використовується для розуміння того, чи вигідно атакувати групу[9]:

`GetValue()` - обчислює кількість ворогів, які знаходяться в певному радіусі один від одного.

`CoverHeuristic.cs` — Визначає можливість знайти укриття, щоб сховатися від ворогів:

`GetValue()` - перевіряє, чи знаходиться юніт під укриттям (тобто чи є перешкода між ним і найближчим ворогом), що може захистити його від пострілу

`HealthHeuristic.cs` — Враховує рівень здоров'я AI, щоб визначити, чи варто ризикувати або ж залишитися у безпеці:

`GetValue()` - обчислює здоров'я юніта як відсоток від його максимально можливого здоров'я.

`SuperiorityHeuristic.cs` — Оцінює кількісну чи силову перевагу. Якщо AI має перевагу, він може атакувати, а при її відсутності — ухилятися:

`GetValue()` - оцінює суперництво між дружніми та ворожими юнітами на полі бою, повертаючи числову величину, яка вказує на перевагу чи меншину однієї з команд.

`TotalHealthHeuristic.cs` — Враховує загальне здоров'я союзників чи ворогів, щоб AI міг оцінити ймовірність успіху в бою:

`GetValue()` -розраховує сумарне здоров'я всіх дружніх юнітів, яке може використовуватися для оцінки загальної витривалості команди.

На рис зображено діаграму класів, для навчання AI-моделей та збору даних.

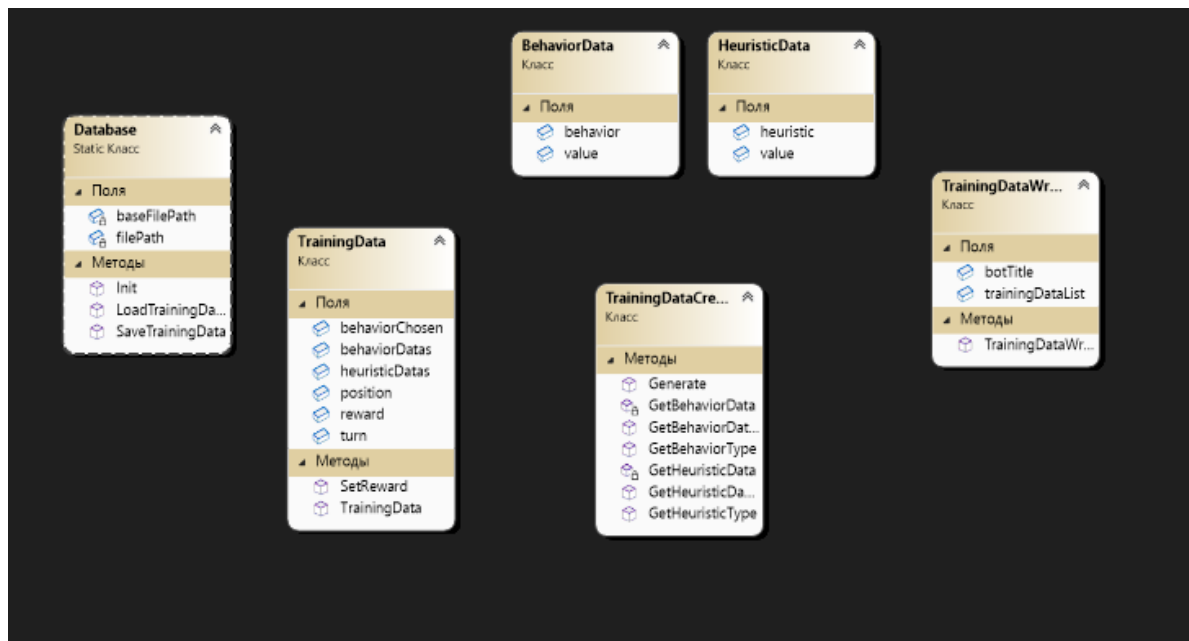


Рис. 2.4. Діаграма класів для навчання AI – моделей та збору даних.

Database — База даних для зберігання інформації, яку використовують для навчання моделі.

TrainingData — Містить набір даних для навчання AI, щоб він міг покращувати свою поведінку.

TrainingDataCreator — Клас, який збирає або створює навчальні дані. Використовується для підготовки AI до роботи, щоб він міг навчатися на нових ситуаціях[8].

На рис зображено діаграму класів для керування юнітами та поведінки AI ворога.

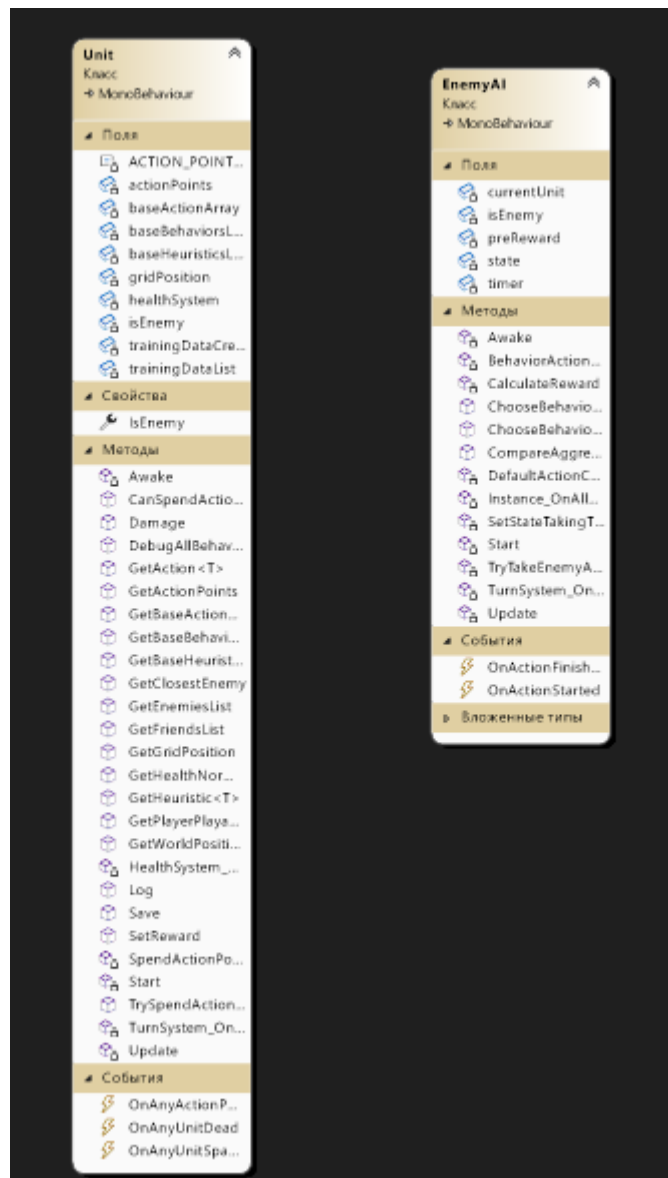


Рис. 2.5. Діаграма класів для керування юнітами та поведінки AI ворога.

Клас EnemyAI визначає поведінку штучного інтелекту ворогів у грі, використовуючи стратегії прийняття рішень для кожного юніта ворога залежно від ситуації на полі бою. Відповідає за обробку чергових ходів ворога та вибір оптимальних дій для досягнення цілей:

Awake() - ініціалізує початковий стан об'єкта на WaitingForEnemyTurn.

Start() - визначає, чи активувати AI для ворогів, перевіряє автоматичний режим гравця, підписується на події зміни ходу та знищення ворогів.

Instance_OnAllEnemiesDead() - вимикає AI та зберігає стан усіх дружніх юнітів, якщо всі вороги мертві[5].

Update() - реалізує логіку станів:

- Очікування свого ходу.
- Взяття ходу (спроба виконати дію або завершення ходу).
- Зайнятість (очікування виконання дії).

SetStateTakingTurn() - завершує дію AI-юніта, оновлює нагороду за стан та переходить до наступного ходу.

CalculateReward() - обчислює "нагороду" як загальний рівень здоров'я ворогів.

TurnSystem_OnTurnChanged() - Встановлює стан TakingTurn і задає таймер, якщо зараз хід ворогів.

TryTakeEnemyAIAction(Action onEnemyAIActionComplete) - обирає юніта для дії. Якщо вдало, активує його дію, інакше завершує хід.

TryTakeEnemyAIAction() - визначає найкращу дію для конкретного юніта та виконує її.

DefaultActionChoosing() - вибирає найкращу дію для юніта серед усіх доступних, порівнюючи їх значення.

BehaviorActionChoosing() - обирає дію юніта на основі поведінкових стратегій і вибирає найкращий варіант[4].

ChooseBehaviorUsingSoftmax() - використовує Softmax для вибору поведінки на основі ймовірностей.

ChooseBehaviorUsingEpsilonGreedy() - використовує epsilon-greedy підхід для вибору поведінки: випадково або найкращий варіант[6].

CompareAggressiveness() - порівнює поточний рівень агресивності гравця зі спрогнозованим середнім рівнем[7].

Клас Unit представляє основну логіку юніта:

Awake() - ініціалізує компоненти (HealthSystem, дії, евристики, поведінки) для юніта.

Start() - встановлює початкову позицію на сітці, додає юніта до сітки та підписується на події зміни ходу і смерті.

Update() - оновлює позицію юніта на сітці, якщо він пересунувся.

HealthSystem_OnDead() - видаляє юніта із сітки, зберігає його дані та знищує об'єкт після смерті.

GetAction<T>() - повертає дію, яка відповідає заданому типу T.

GetHeuristic<T>() - повертає евристику, яка відповідає заданому типу T.

TrySpendActionPointsToTakeAction() - перевіряє та витрачає очки дій для виконання заданої дії.

CanSpendActionPointsToTakeAction() - перевіряє, чи вистачає очок дій для виконання заданої дії.

SpendActionPoints() - витрачає задану кількість очок дій та викликає подію зміни очок дій.

GetGridPosition() / GetWorldPosition() - повертає позицію юніта на сітці або в світових координатах.

GetClosestEnemy() - знаходить найближчого ворога на сітці.

GetEnemiesList() - повертає список ворогів (залежно від того, чи є юніт ворогом).

GetFriendsList() - повертає список союзників (залежно від типу юніта).

GetActionPoints() - повертає поточну кількість очок дій.

GetHealthNormalized() - повертає нормалізоване значення здоров'я (від 0 до 1).

Damage(int damageAmount) - завдає пошкодження юніту через систему здоров'я.

Log(int turn, BaseBehavior behavior) - зберігає дані про вибрану поведінку для тренування AI.

SetReward(int reward) - присвоює винагороду останнім зібраним даним.

Save() - зберігає всі зібрані дані тренування у базу даних.

DebugAllBehaviors() - виводить у консоль значення всіх поведінок юніта для налагодження.

TurnSystem_OnTurnChanged(object sender, EventArgs e) - відновлює очки дій на початку ходу юніта.

На рис зображено діаграму класів для дій юніта.

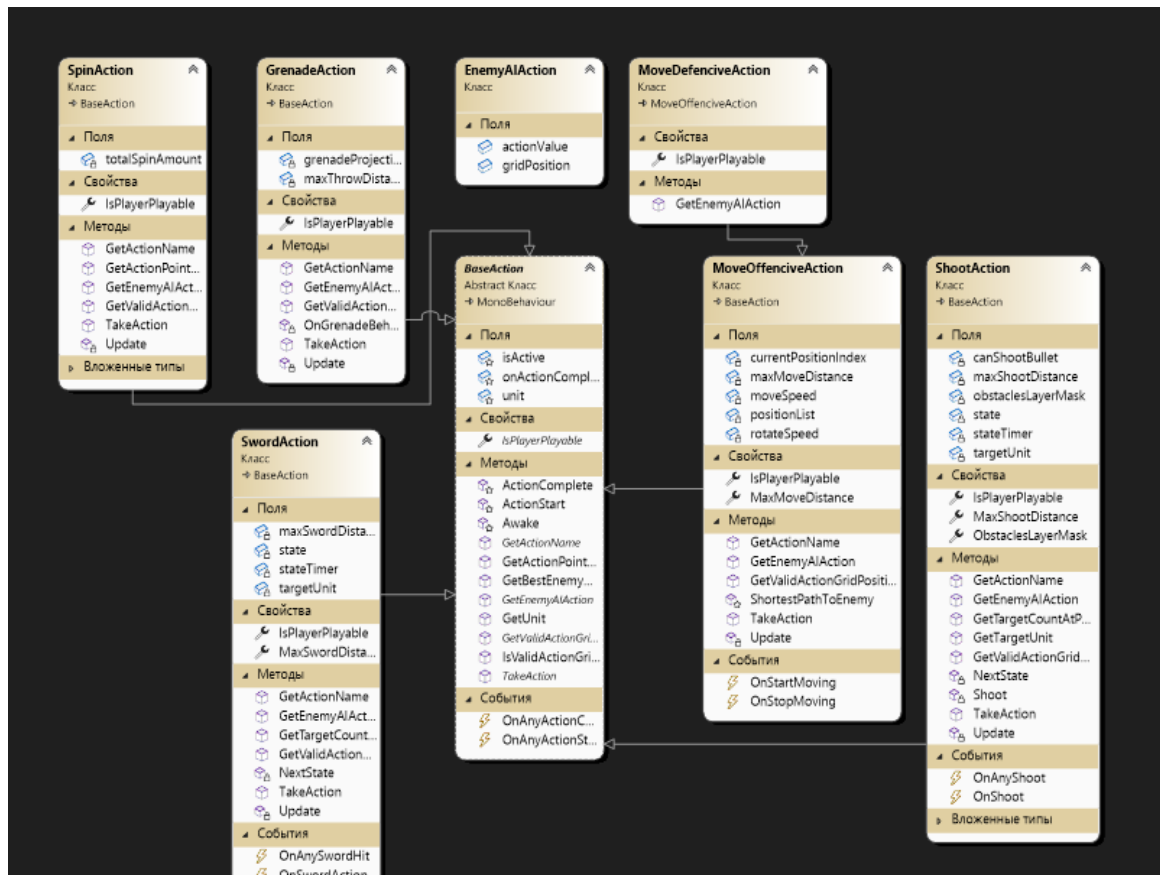


Рис. 2.6. Діаграма класів для дій юніта

BaseAction.cs - визначає базову структуру для дій, які можуть виконувати юніти (як вороги, так і гравець):

Awake() - ініціалізує юніт, якому належить дія, за допомогою GetComponent<Unit>().

GetActionName() – повертає імена дій.

TakeAction() – реалізує логіку виконання дії для конкретної позиції на сітці.

IsValidActionGridPosition() – перевіряє, чи є обрана позиція дійсною для дії.

`GetValidActionGridPositionList()` – повертає список усіх допустимих позицій для виконання дії.

`GetActionPointsCost()` – повертає вартість дії в очках дій

`ActionStart()` – сигналізує про початок дії та викликає глобальну подію `OnAnyActionStarted`

`ActionComplete()` – сигналізує про завершення дії та викликає глобальну подію `OnAnyActionCompleted`

`GetBestEnemyAIAction()` – аналізує список доступних позицій для дії. Сортує їх за значенням ефективності та повертає найкращу дію для AI

`GetEnemyAIAction()` – повертає конкретну дію AI для заданої позиції

`GetUnit()` - повертає об'єкт `Unit`, який виконує дію

`EnemyAIAction.cs` - структура даних, що використовується для представлення потенційної дії ворожого AI у грі. Допомогає зберігати позицію та цінність дії, щоб AI міг обрати найкращий хід серед усіх можливих варіантів.

`GrenadeAction.cs` - відповідає за реалізацію дії кидання гранати в грі для юніта. Клас визначає умови, за яких ця дія можлива, і виконує анімацію та логіку створення проєктиву гранати:

`GetActionName()` - повертає назву дії.

`GetValidActionGridPositionList()` - розраховує список допустимих позицій для кидка гранати на основі `maxThrowDistance` (максимальної відстані кидка). Перевіряє кожну клітинку на валідність через `LevelGrid`.

`TakeAction()` - створює проєktiv гранати на основі префабу, передає йому цільову позицію та функцію, що виконується після завершення.

`GetEnemyAIAction()` - повертає об'єкт `EnemyAIAction`, що містить цільову позицію з нульовою цінністю.

`OnGrenadeBehaviorComplete()` - викликається після завершення дії гранати та завершує активну дію викликом `ActionComplete`.

`Update()` - оновлює стан дії лише, коли вона активна.

MoveDefensiveAction.cs - реалізує захисну поведінку для юнітів під контролем AI, обчислюючи оптимальні дії для переміщення з урахуванням відстані до ворогів.

MoveOffensiveAction.cs - відповідає за реалізацію агресивної поведінки юнітів на полі бою, дозволяючи їм переміщатися до позицій для атаки або наближення до ворогів:

Update() - обробляє переміщення юніта по заздалегідь розрахованому шляху. Враховує швидкість руху та обертання юніта.

TakeAction() - розраховує шлях до позиції за допомогою алгоритму Pathfinding. Ініціює рух юніта вздовж цього шляху.

GetValidActionGridPositionList() - генерує список доступних позицій для переміщення (максимальна відстань руху, прохідність клітинок, наявність інших юнітів).

ShortestPathToEnemy() - обчислює найкоротший шлях до найближчого ворога.

GetEnemyAIAction() - розраховує цінність дії для AI (кількість ворогів у заданій позиції, відстань до цілей по найкоротшому шляху).

GetActionName() - повертає назву дії.

ShootAction.cs - відповідає за обробку дії стрільби юніта:

Update()- обробляє три стани дії

TakeAction() - ініціює дію стрільби по цілі на вказаній позиції. Переходить у стан прицілювання.

Shoot() - завдає 25 одиниць шкоди цільовому юніту. Викликає події OnAnyShoot та OnShoot для анімацій або обробки інших ефектів.

GetValidActionGridPositionList()- повертає список доступних позицій для стрільби

GetEnemyAIAction() - розраховує цінність дії для AI на основі здоров'я цільового юніта.

GetTargetCountAtPosition() - повертає кількість доступних цілей для стрільби з вказаної позиції.

GetActionName()- повертає назву дії.

SpinAction.cs - відповідає за обертання юніта на місці на 360 градусів:

Update() - обертає юніта по Y-осі на 360 градусів. Завершує дію після повного обертання.

TakeAction() - ініціює дію обертання та обнуляє кількість виконаних обертів.

GetActionName() - повертає назву дії.

GetValidActionGridPositionList() - повертає позицію юніта, оскільки дія виконується на місці.

GetActionPointsCost() - повертає вартість дії у 1 очко дії.

GetEnemyAIAction() - повертає нульове значення actionValue, оскільки дія не має тактичного значення для AI.

SwordAction.cs - визначає дію атаки мечем, яку може виконувати юніт:

Update() - Виконує управління станами анімації удару мечем. Юніт спочатку наводиться на ціль, а після цього завдає шкоди та завершує дію.

NextState() - Змінює стан дії:

GetActionName() - Повертає назву дії.

GetEnemyAIAction() - Повертає дію для AI з actionValue = 200, що вказує на високу пріоритетність атаки.

GetValidActionGridPositionList() - Визначає всі можливі позиції для атаки мечем (радіус 1 клітина). Перевіряє наявність ворожих юнітів на сусідніх клітинах та додає ці позиції до списку доступних.

TakeAction() - Ініціює дію атаки, задає ціль для удару та змінює стан на SwingingSwordBeforeHit. Викликає подію OnSwordActionStarted для відстеження початку дії.

GetTargetCountAtPosition() - Повертає кількість можливих цілей для атаки мечем у межах дії юніта.

2.2 Реалізація функціоналу додатку.

Розглянемо клас EnemyAI, який визначає поведінку штучного інтелекту ворогів у грі:

Метод Update() контролює хід ворога. Якщо зараз черга гравця, то ворог не робить нічого. Лістинг коду:

```
private void Update()
{
    if (TurnSystem.Instance.IsPlayerTurn() == isEnemy)
    {
        return;
    }
    switch (state)
    {
        case State.WaitingForEnemyTurn:
            break;
        case State.TakingTurn:
            timer -= Time.deltaTime;
            if (timer <= 0f)
            {
                if (TryTakeEnemyAIAction(SetStateTakingTurn))
                {
                    state = State.Busy;
                }
                else
                {
                    TurnSystem.Instance.NextTurn();
                }
                //TurnSystem.Instance.NextTurn();
            }
            break;
        case State.Busy:
            break;
    }
}
```

Метод SetStateTakingTurn():

Обчислює різницю винагород preReward - CalculateReward() для оцінки успішності дій.

Викликає OnActionFinished після закінчення дії та встановлює затримку в timer. Лістинг коду:

```
private void SetStateTakingTurn()
{
    currentUnit.SetReward(preReward - CalculateReward());
    OnActionFinished?.Invoke();
    timer = 0.5f;
    state = State.TakingTurn;
}
```

Метод CalculateReward():

Оцінює винагороду на основі здоров'я ворогів (enemiesHealth), що може використовуватися для оптимізації поведінки. Лістинг коду:

```
private int CalculateReward()
{
    var enemies = currentUnit.GetEnemiesList();
    float enemiesHealth = enemies.Sum(x => x.GetHealthNormalized() * 100f);
    int reward = (int)enemiesHealth;
    return reward;
}
```

Метод TurnSystem_OnTurnChanged():

Якщо зараз черга на хід для ворога, метод встановлює state в TakingTurn та ініціалізує timer. Лістинг коду:

```
private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
{
    if (!(!TurnSystem.Instance.IsPlayerTurn() ^ isEnemy))
    {
        state = State.TakingTurn;
        timer = 2f; } }
```

Метод TryTakeEnemyAIAction():

Працює з юнітами, щоб знайти оптимальну дію. Якщо знаходиться дія, викликається OnActionStarted. Лістинг коду:

```
private bool TryTakeEnemyAIAction(Action onEnemyAIActionComplete)
{
    //Debug.Log("Take Enemy AI Action");
    List<Unit> teammates = isEnemy ?
        UnitManager.Instance.GetEnemyUnitList() :
        UnitManager.Instance.GetFriendlyUnitList();
    foreach (Unit unit in teammates)
    {
        currentUnit = unit;
        preReward = CalculateReward();
        if (TryTakeEnemyAIAction(unit, onEnemyAIActionComplete))
        {
            OnActionStarted?.Invoke(unit);
            return true;
        }
    }
    return false;
}
```

Метод BehaviorActionChoosing():

Виконує вибір поведінки на основі методу Softmax або EpsilonGreedy.

Використовує значення Softmax для визначення ймовірності вибору дії відповідно до оцінки. Лістинг коду:

```
private void BehaviorActionChoosing(Unit enemyUnit, ref EnemyAIAction bestEnemyAIAction, ref
BaseAction bestBaseAction, out BaseBehavior choosenBehavior)
{
```

```

        enemyUnit.DebugAllBehaviors();
        choosenBehavior = ChooseBehaviorUsingSoftmax(enemyUnit.GetBaseBehaviorList());
        int value = choosenBehavior.GetValue(out bestBaseAction);
        bestEnemyAIAction = bestBaseAction.GetBestEnemyAIAction();
        Debug.Log(">>> CHOOSSED BEST: " + choosenBehavior.GetType() + " - " +
bestBaseAction.GetType() + " - " + value);
    }

```

Метод ChooseBehaviorUsingSoftmax():

Обчислює ймовірності для кожної поведінки та вибирає поведінку за допомогою випадкової вибірки. Лістинг коду:

```

public BaseBehavior ChooseBehaviorUsingSoftmax(List<BaseBehavior> baseBehaviorsList, float
temperature = 1.0f)
{
    float[] probabilities = new float[baseBehaviorsList.Count];
    float sum = 0f;
    for (int i = 0; i < baseBehaviorsList.Count; i++)
    {
        probabilities[i] = Mathf.Exp(baseBehaviorsList[i].GetValue(out _) / temperature);
        sum += probabilities[i];
    }
    for (int i = 0; i < probabilities.Length; i++)
    {
        probabilities[i] /= sum;
    }
    float randomValue = UnityEngine.Random.value;
    float cumulative = 0f;
    for (int i = 0; i < baseBehaviorsList.Count; i++)
    {
        cumulative += probabilities[i];
        if (randomValue <= cumulative)
        {
            return baseBehaviorsList[i];
        }
    }
    return baseBehaviorsList[baseBehaviorsList.Count - 1];
}

```

Метод ChooseBehaviorUsingEpsilonGreedy():

Іноді обирає випадкову поведінку для забезпечення варіативності.

Лістинг коду:

```

public BaseBehavior ChooseBehaviorUsingEpsilonGreedy(List<BaseBehavior> baseBehaviorsList, float
epsilon = 0.1f)
{
    float randomValue = UnityEngine.Random.value;
    if (randomValue < epsilon)
    {
        int randomIndex = UnityEngine.Random.Range(0, baseBehaviorsList.Count);
        return baseBehaviorsList[randomIndex];
    }
    else
    {
        BaseBehavior bestBehavior = baseBehaviorsList[0];
        int bestValue = bestBehavior.GetValue(out _);
        for (int i = 1; i < baseBehaviorsList.Count; i++)
        {
            int value = baseBehaviorsList[i].GetValue(out _);
            if (value > bestValue)

```

```

        {
            bestBehavior = baseBehaviorsList[i];
            bestValue = value;
        }
    }
    return bestBehavior;
}
}

```

Метод CompareAggressiveness():

Оцінює рівень агресивності гравця, порівнюючи поточне середнє здоров'я ворогів з передбаченим. Лістинг коду:

```

public int CompareAggressiveness()
{
    int predictedAverageHealth =
Prediction.Instance.GetAverageTotalPlayerHealth(TurnSystem.Instance.Turn-1);
    var currentAverageHealth =
UnitManager.Instance.GetFriendlyUnitList()[0].GetHeuristic<TotalHealthHeuristic>().GetValue();
    return predictedAverageHealth - currentAverageHealth;
}

```

Висновки до другого розділу

Система симуляції в моїй грі реалізує функціонал покрокової стратегії з елементами штучного інтелекту, що включає управління юнітами та взаємодію з ворогами. Симуляція забезпечується наборами скриптів для обчислення рішень та дій на кожен хід.

Було розглянуто алгоритми основних процесів системи та проаналізовано взаємодію класів системи під час процесів гри.

В останньому пункті наводиться детальний розгляд реалізації штучного інтелекту юніта.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ДОДАТКОМ

3.1. Інтерфейс та порядок роботи з програмним додатком

Опис роботи з додатком включає в себе розробку текстової та ілюстраційної частини, які демонструють зовнішній вигляд закінченого продукту.

Після завантаження користувач опиняється у головному меню (рис. 3.1).

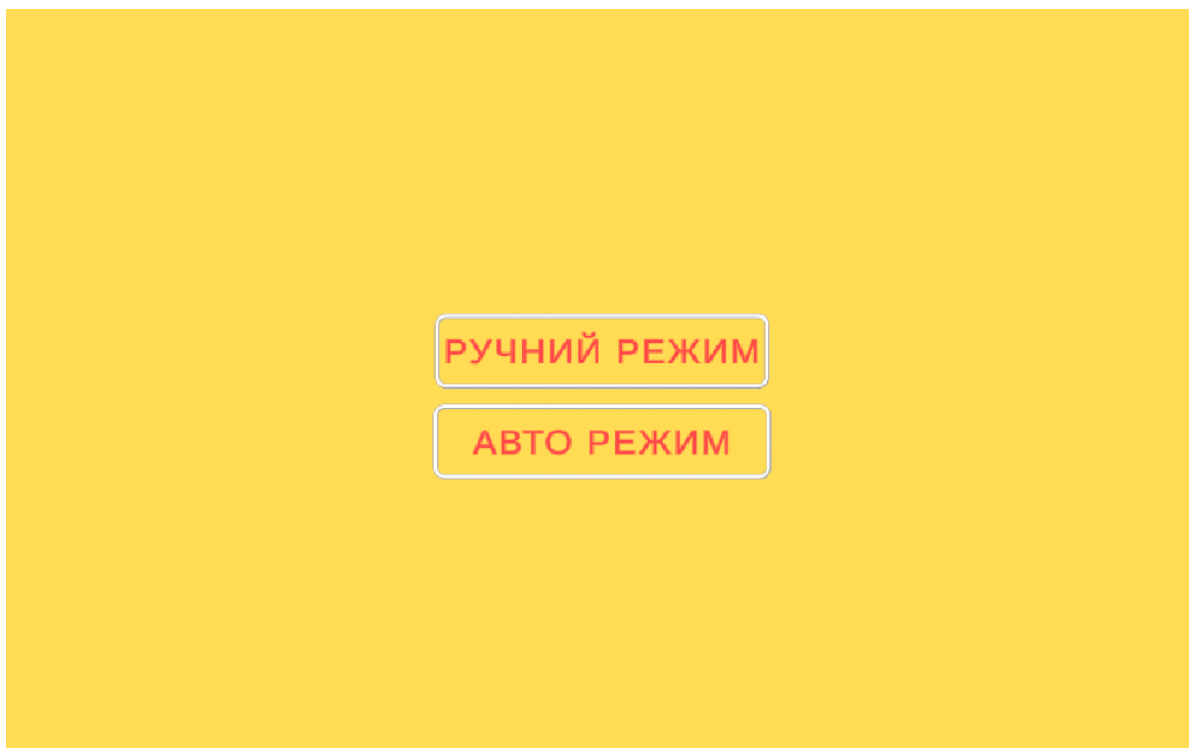


Рис. 3.1. Головне меню

Зображено два елементи керування, що представлені у вигляді кнопок: “Ручний режим” та “Авто режим”. При натисканні кнопки “Ручний режим” починається гра, де користувач може грати (рис. 3.2). Якщо натиснути кнопку “Авто режим” не потрібно нічого робити, гра сама почне грати з використанням ші як за союзних юнітів так і за ворожих (рис. 3.3).

Загальне керування. На екрані гравця з’являється сцена проходження гри, автоматично вибирається дія “Рух”, ця дія дозволяє гравцеві переміщатись по карті за допомогою сітки, гравець натискає ЛКМ по сітці і юніт може виконувати переміщення на нову позицію. Поточний юніт підсвічується зеленим кругом, для того щоб змінити керування на іншого

юніта потрібно на нього натиснути. За один хід кожен юніт може виконати 3 ходи, коли всі юніти гравця виконали ходи, то потрібно натиснути на кнопку “End turn”, щоб закінчити хід гравця і хід перейшов до ші ворогів.



Рис 3.2. Початок гри при ручному режимі

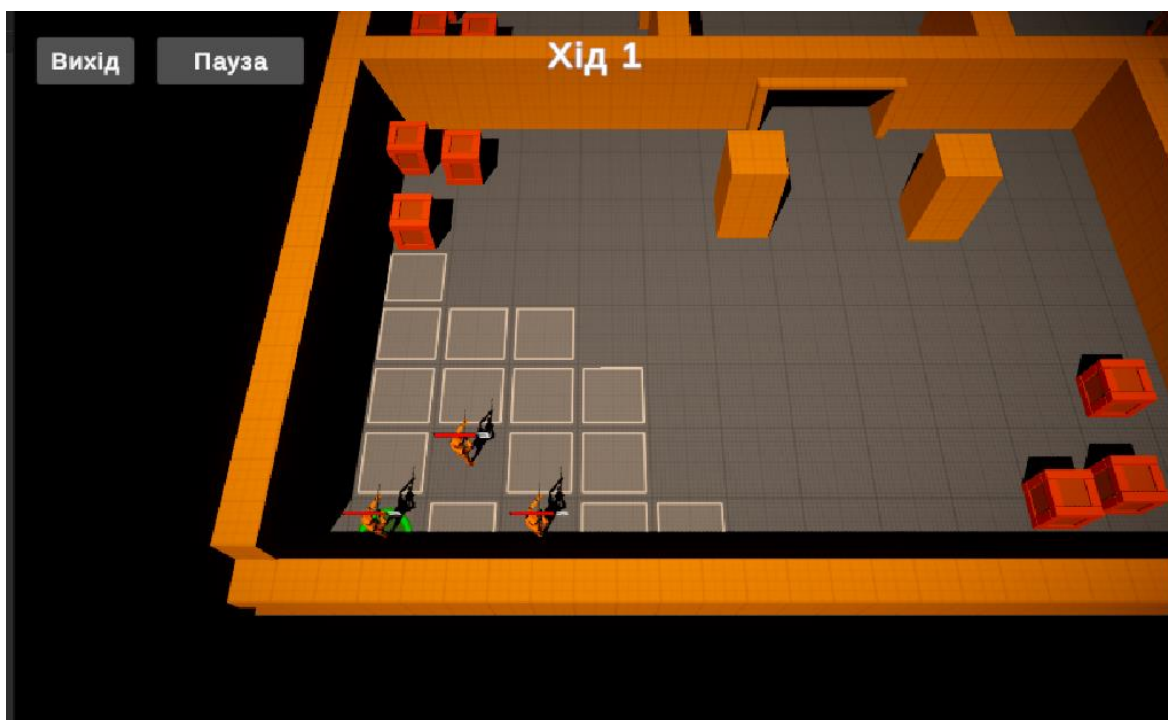


Рис 3.3. Початок гри при авто режимі

Для того, щоб продивлятися всю карту користувач натискає клавіші W S A D Q E (рис. 3.4), де:

- W – рух вперед;
- S – рух назад;
- A – рух вліво;
- D – рух вправо;
- Q – обертання від 0 до 360 градусів в правий бік(рис. 3.4);
- E – обертання від 0 до 360 градусів в лівий бік.



Рис. 3.4. Обертання від 0 до 360 градусів натискаючи клавіатуру “Q”

Знищення ворогів. Пересуваючись по локації гравець може натрапити на ворогів, тому їх потрібно знищити, для того щоб знищити ворога можна використати 3 дії:

- Стрільба (рис. 3.5, рис. 3.6)
- Граната (рис. 3.9, рис. 3.10)
- Меч (рис. 3.7, рис. 3.8)

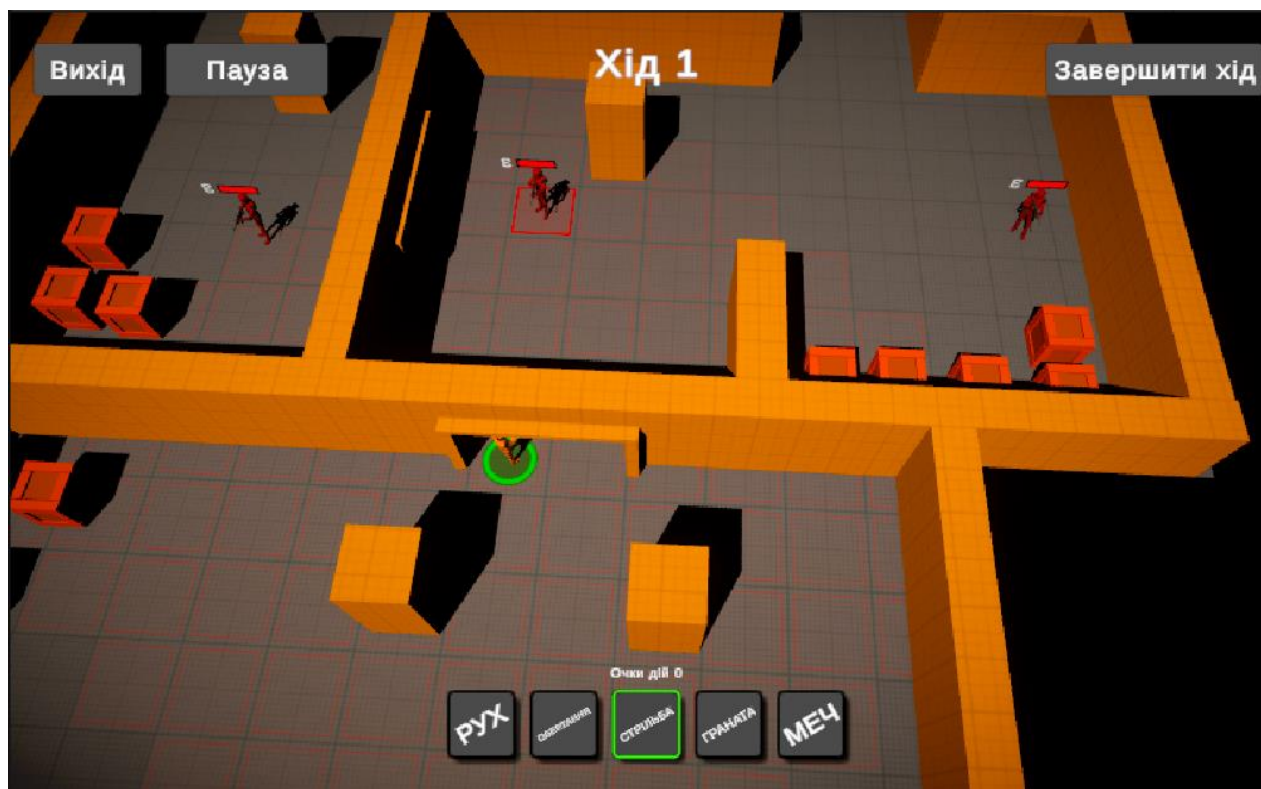


Рис.3.5. Підготовка до стрільби юніта

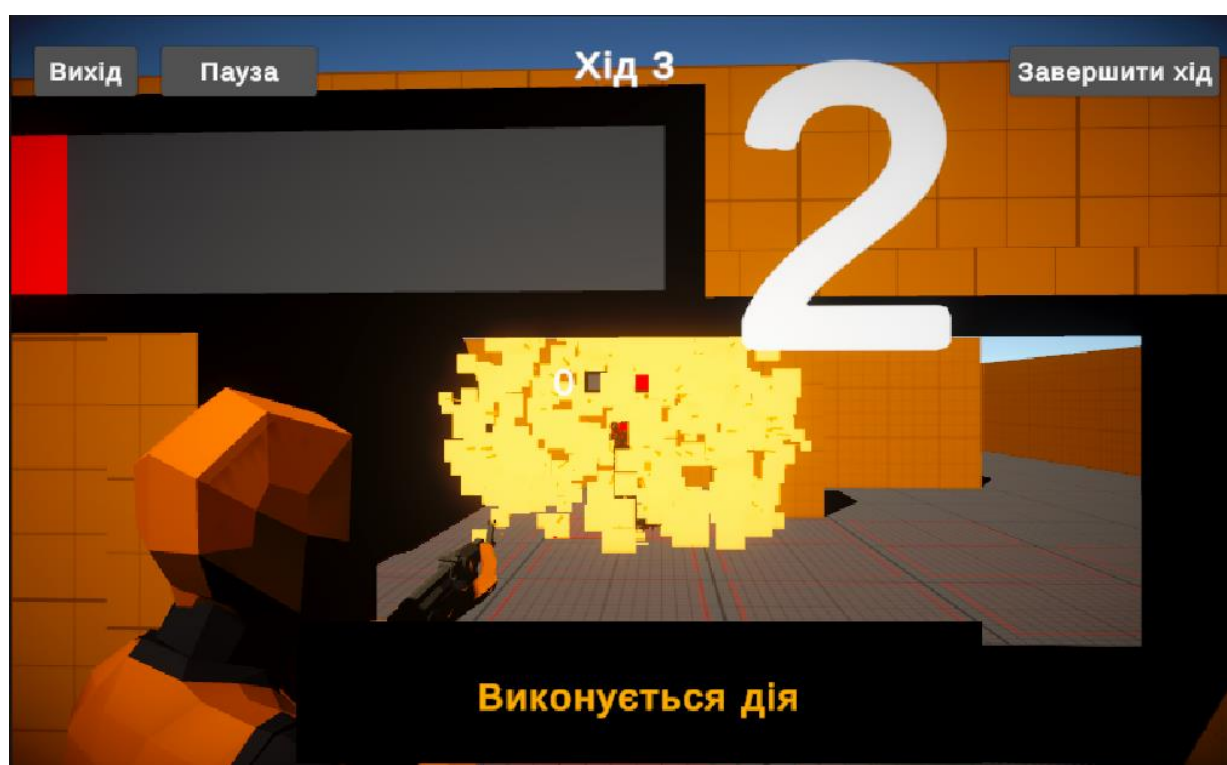


Рис.3.6. Стрільба юніта



Рис.3.7. Підготовка до удара мечем

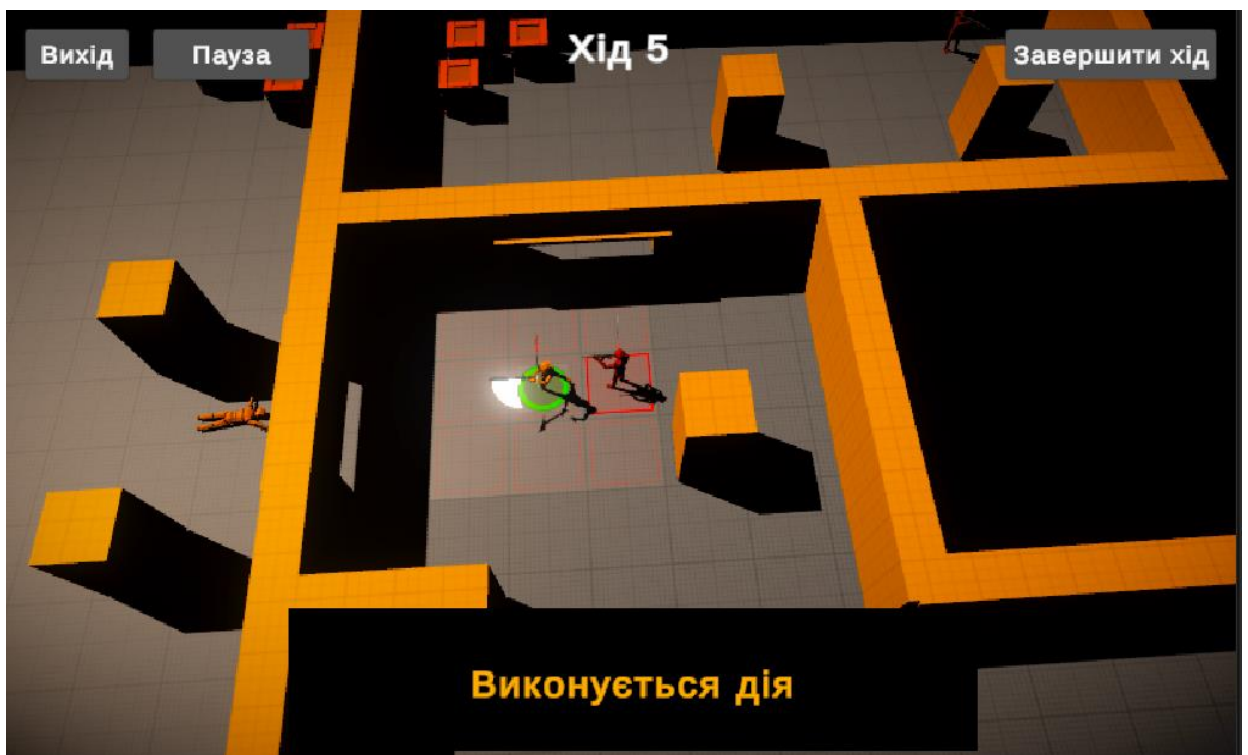


Рис.3.8. Удар мечем юніта

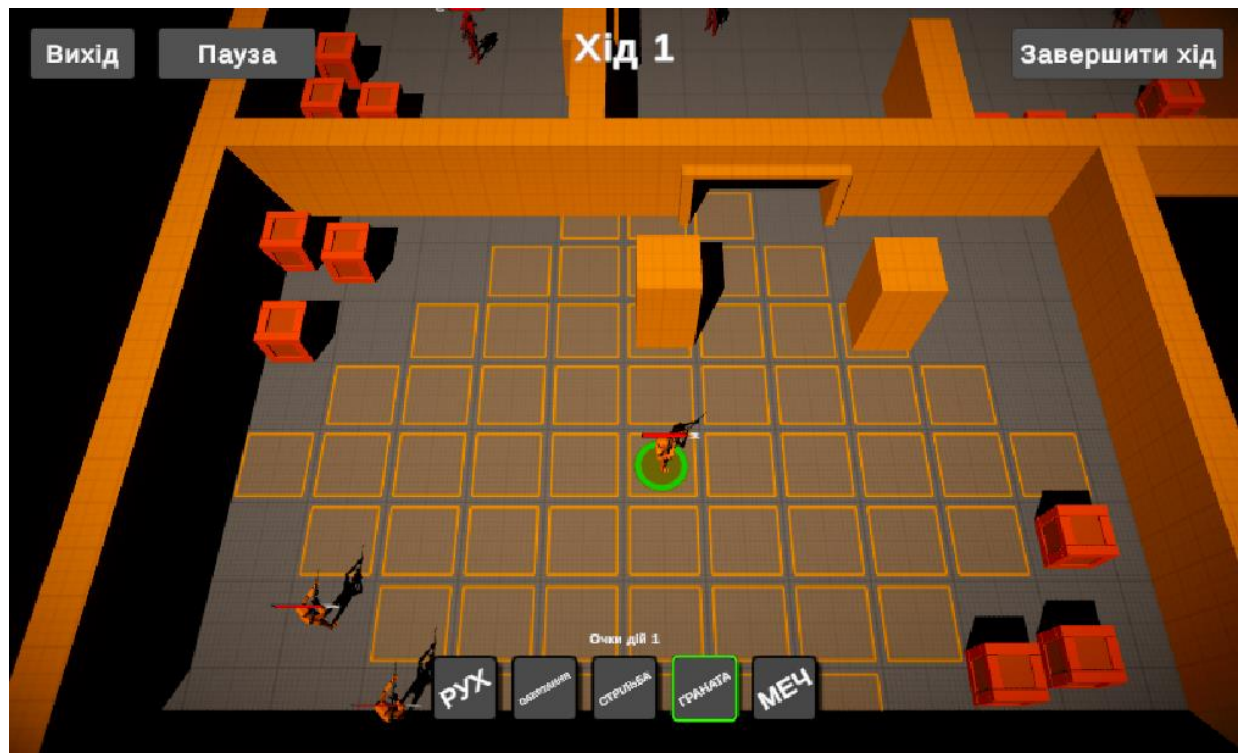


Рис.3.9. Підготовка до кидання гранати

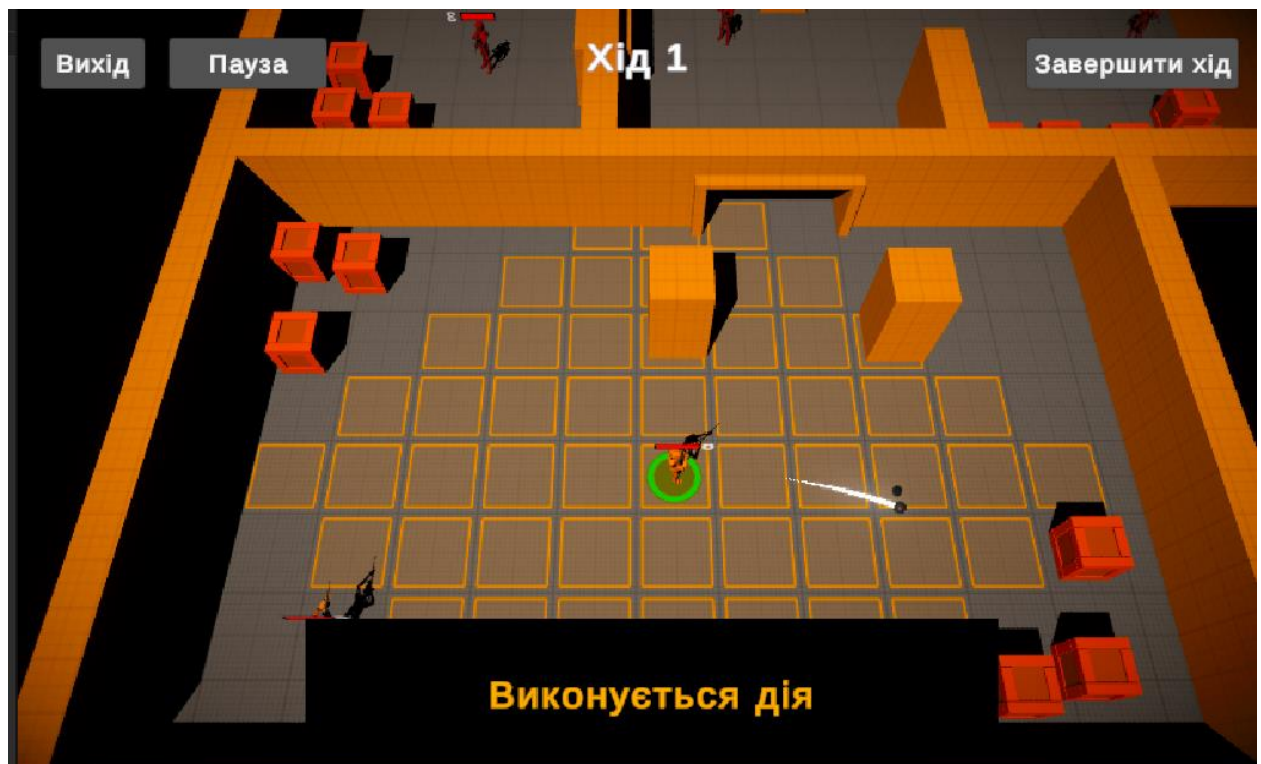


Рис.3.10. Кидання гранати юнітом

Також гравець може виконати дію “Обертання”, яка просто використовує один хід, якщо не потрібно йти в атаку, або пересуватись по локації (рис. 3.11).

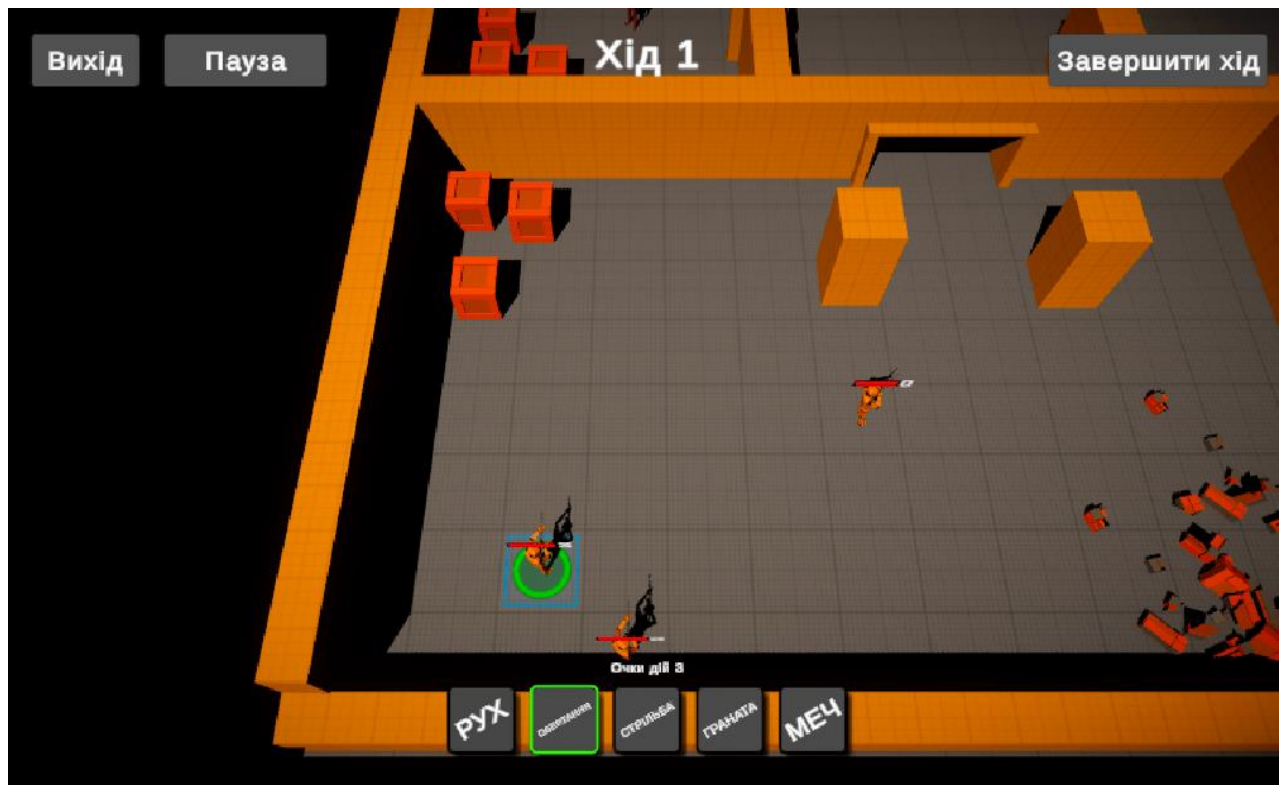


Рис.3.11. Обертання юніта

Гравець може зупинити гру натиснувши на кнопку “Пауза”. Для того, щоб розпочати гру, спочатку гравець може натиснути кнопку “Вихід”. Після цього він переходить до головного меню.

3.2. Тестування роботи системи

Перевірка на працездатність – невід’ємний процес під час розробки програмного коду системи[20]. Ігрове тестування в певних аспектах відрізняється від тестування інтернет-ресурсів, мобільних додатків та інших продуктів[21].

Тестування ігрових систем містить такі основні види:

- тестування функціональності;
- тестування контенту;
- тестування геймплею;
- користувацьке тестування.

Тестування функціональності. Нижче наведено тест-кейси по основним діям гравця.

Тест № 1. Кнопка «Manual Play» та «Auto Play»

Передумови: Гравець перебуває у головному меню гри.

Кроки:

1.Запустити гру.

2.Натиснути кнопку «Manual Play» або «Auto Play».

Очікуваний результат: Вибраний режим активується, і гравець переходить до відповідного сценарію гри.

Фактичний результат: Вибраний режим активується, і гравець переходить до відповідного сценарію гри.

Тест № 2. Кнопка «Пауза» (Під час гри)

Передумови: Гравець активує гру та починає бій (TURN 1).

Кроки:

1.Натиснути кнопку «Пауза» у верхньому лівому куті.

Очікуваний результат: Гра зупиняє всі дії, юніти припиняють рух, а інтерфейс повідомляє про паузу.

Фактичний результат: Гра зупиняє всі дії, юніти припиняють рух, а інтерфейс повідомляє про паузу.

Тест № 3. Підсвічування кнопок дій юніта

Передумови: Гравець активує юніта в бойовій сесії.

Кроки:

1.Обрати юніта у грі.

2.Перевірити кнопки нижнього інтерфейсу: «РУХ», «СТРІЛЬБА», «ГРАНАТА», «МЕЧ».

Очікуваний результат: Обрана кнопка дії підсвічується чітко (наприклад, зеленою рамкою або іншим кольором).

Фактичний результат: Обрана кнопка дії підсвічується чітко (наприклад, зеленою рамкою або іншим кольором).

Тест № 6. Робота автогри (Auto Play)

Передумови: Гравець обирає режим «Auto Play» у головному меню.

Кроки:

1.Запустити режим «Auto Play».

2.Спостерігати за діями юнітів.

Очікуваний результат: Юніти автоматично приймають рішення та виконують хід без втручання гравця.

Фактичний результат: Юніти автоматично приймають рішення та виконують хід без втручання гравця.

Тестування геймплею. Було проведено тестування геймплейних особливостей, включаючи основну – керування гравцем його рух, стрільба і тд. Під час тестування доступного функціоналу у нього не було виявлено жодних недоліків чи помилок у роботі.

Висновки до третього розділу

В даному розділі було наведено порядок встановлення та налаштування додатку та визначено мінімальну конфігурацію системи для комфортної гри.

Реалізована система симуляції містить інтерфейси головного меню, меню паузи та основний ігровий інтерфейс, робота та функції елементів якого були наведені. Також було проведено функціональне та геймплейне тестування додатку. Знайдені помилки було виправлено, система працює коректно та виконує поставлені задачі.

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи було розроблено систему адаптивного штучного інтелекту для покрокової гри-стратегії, яка використовує методи машинного навчання для оптимізації поведінки бота.

Було проаналізовано ігри, які є аналогами програмного продукту, що розробляється. При аналізі розглянуто основні елементи гри, їх геймплей та тематику. Розібрано основні елементи та характеристики жанру. Також отримано теоретичний фундамент для подальшої роботи і повністю сформовано уявлення про гру, що створюється.

Основним результатом є впровадження інтелектуальних алгоритмів, зокрема епсілон-жадібного підходу та Softmax, для вибору оптимальних дій на основі евристик і біхейвіорів юнітів. Це дозволяє ботам не тільки реагувати на поточну ситуацію на полі бою, але й прогнозувати дії гравця, підвищуючи рівень складності гри.

Розроблені алгоритми мають високу практичну цінність для використання в покрокових стратегіях, оскільки значно покращують взаємодію гравця з комп'ютерним супротивником, роблячи його більш реалістичним і адаптивним. Вони можуть бути інтегровані в інші ігрові проекти, зокрема в стратегічні ігри, де важливе передбачення і адаптація поведінки бота. Удосконалення методів машинного навчання дозволить ще більше покращити ефективність системи ШІ, підвищуючи інтелектуальну складність ігрових супротивників.

Всі механіки працюють належним чином без збоїв. Гра розроблялася на рушії Unity3D, який добре себе показав під час роботи. Інтуїтивно зрозумілий інтерфейс, швидкість роботи та величезна кількість документації і повчальних роликів на YouTube допомагають в будь-якому питанні. Великі можливості для повторного використання коду та вбудована оптимізація графіки, фізики. Він повністю задовольняє нас за своїми базовими

характеристиками до того ж має потужну підтримку товариства, що значно знижує поріг входження.

Подальші дослідження можуть бути спрямовані на:

1. Розширення алгоритмів навчання для включення більш складних стратегій і адаптації до непередбачуваних дій гравця.
2. Оптимізація часу обробки для забезпечення швидкої реакції в реальному часі без втрати ефективності.
3. Інтеграція з іншими методами машинного навчання, такими як глибоке навчання, для підвищення точності прогнозування і адаптивності.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Надворний М. Ю., Чижмотря О. Г. Обґрунтування актуальності теми навчання та використання штучного інтелекту на прикладі покрокової гри-стратегії. Комп'ютерні технології: інновації, проблеми, рішення: тези VII Всеукраїнської науково-технічної конференції, м. Житомир, 02-03 грудня 2024 р. – Житомир: Державний університет «Житомирська політехніка», 2024.
2. Надворний М. Ю., Чижмотря О. Г. Класифікація та аналіз біхейвіорів на основі евристик у покрокових іграх. Комп'ютерні технології: інновації, проблеми, рішення: тези VII Всеукраїнської науково-технічної конференції, м. Житомир, 02-03 грудня 2024 р. – Житомир: Державний університет «Житомирська політехніка», 2024.
3. Russell, S., Norvig, P. Artificial Intelligence: A Modern Approach. 3rd ed. New Jersey: Prentice Hall, 2010. 1152 p.
4. Sutton, R., Barto, A. Reinforcement Learning: An Introduction. 2nd ed. Cambridge: MIT Press, 2018. 552 p.
5. Sicart, M., Möller, F. Game AI Pro: Collected Wisdom of Game AI Professionals. New York: CRC Press, 2013. 350 p.
6. Bengio, Y., Courville, A., Vincent, P. Deep Learning for Artificial Intelligence. Journal of Machine Learning Research, 2013. Vol. 27, No. 5. pp. 1231–1256.
7. Silver, D., Huang, A., Maddison, C. Mastering the Game of Go with Deep Neural Networks and Tree Search. Nature, 2016. Vol. 529, No. 7587. pp. 484–489.
8. Laird, J. E., van Lent, M. High-Level AI in Computer Games. AI Magazine, 2001. Vol. 22, No. 2. pp. 15–25.
9. Gardi, A., Gracia, M. AI Technologies for Interactive Digital Entertainment. IEEE Transactions on Computational Intelligence and AI in Games, 2019. Vol. 11, No. 3. pp. 290–298.

10. Gough, A., Harris, M. Advanced AI Techniques in Strategy Games. *International Journal of Artificial Intelligence in Games*, 2021. Vol. 6, No. 4. pp. 320–335.
11. Tessler, C., Higgins, J. Deep Hierarchical Learning in Minecraft. *ICLR Proceedings*, 2017.
12. Yan, H., Li, Y. Pathfinding and Decision Making in Strategy Games: A Review. *Game Development Journal*, 2019. Vol. 7, No. 2. pp. 110–123.
13. Glenford J. Myers, Corey Sandler, Tom Badgett. *The Art of Software Testing*. 256 p.
14. Alan Page, Ken Johnston, Bj Rollison. *How We Test Software at Microsoft*. 405 p.
15. Mahmoud, R., Silver, D. Unity3D: A Platform for Interactive Applications and Games. *Journal of Computer Science*, 2018. Vol. 45, No. 3, pp. 189–202.
16. Johnson, C., Lee, T. *Cross-Platform Game Development Using Unity*. Cambridge: MIT Press, 2019. 368 p.
17. Wright, S., Paterson, M. *Enhancing Unity Capabilities Through Plugins*. New York: Springer, 2020. 310 p.
18. Wilson, J., Miller, R. *Speeding Development in Unity: A Practical Guide*. *Journal of Game Design*, 2017. Vol. 29, No. 4, pp. 215–228.
19. Hay, M., Peters, D. *Integrated Development Environment in Unity: An In-Depth Review*. Oxford: Oxford University Press, 2016. 275 p.
20. Thompson, L., Davis, M. *Physics and Animation in Unity: Theory and Practice*. London: Wiley, 2018. 305 p.
21. Martin, R. Modern Development Tools: Insights and Approaches. *Technical Programmers' Journal*, 2021. Vol. 39, No. 2, pp. 55–67.
22. Johnson, C. *Extending Code Editors: Practical Guidelines*. Cambridge: MIT Press, 2020. 258 p.
23. Smith, T. Integrating Git into Modern IDEs. *Journal of Programming*, 2019. Vol. 44, No. 3, pp. 88–96.

- 24.Lindberg, C. Debugging in VS Code: Essential Tools. New York: O'Reilly Media, 2020. 210 p.
- 25.Adams, B. Improving Developer Productivity in VS Code. Software Review Journal, 2021. Vol. 33, No. 5, pp. 102–111.
- 26.Robinson, P. Developer Communities: Role in Software Tool Evolution. Cambridge: MIT Press, 2018. 322 p.
- 27.Derry, A. Open Tools: Benefits of Open Source Software. Programming and Development Journal, 2022. Vol. 34, No. 4, pp. 56–68.
- 28.Smith, J. Cross-Platform Compatibility in Modern 3D Editors: A Look at Blender. Computer Technologies, 2021. Vol. 39, No. 3, pp. 122–134.
- 29.Barnes, M. Comprehensive Tools for 3D Graphics: Blender as a Market Leader. Journal of 3D Modeling, 2020. Vol. 28, No. 6, pp. 75–86.
- 30.Reynolds, S. Extensibility of 3D Editors: Interfaces and Blender Plugins. Software Review Journal, 2021. Vol. 32, No. 7, pp. 101–110.
- 31.Low, P. Blender Programming Community: Support and Resources. Technical Journal, 2019. Vol. 23, No. 2, pp. 45–52.
- 32.Clark, D. Modeling Techniques in Blender: Advanced Capabilities. 3D Technologies, 2021. Vol. 15, No. 9, pp. 130–145.
- 33.Hill, T. Programming Basics: Quickly Mastering C#. Beginners' Book, 2020. pp. 58–70.
- 34.James, M. Overview of Object-Oriented Programming with C#. Professional Programming Journal, 2019. Vol. 22, No. 4, pp. 123–137.
- 35.Smith, A. .NET Framework: Advantages for C# Developers. Software Journal, 2021. Vol. 34, No. 2, pp. 45–58.
- 36.Roberts, L. Cross-Platform Development and Growth with C#. International Development Journal, 2020. Vol. 19, No. 3, pp. 77–90.
- 37.Phelps, D. Modern Language Tools and Technologies: C# Compared to Other Languages. Programming and Technologies, 2021. Vol. 28, No. 5, pp. 101–113.

- 38.Lee, K. How C# Has Revolutionized the Programming Industry: An Insider's View. *Technology Review*, 2022. Vol. 12, No. 8, pp. 30–42.
- 39.Johnson, C. *Unity Game Development: Full Course with C#*. Udemy, 2021.
- 40.Smith, D. *C# for Unity: How to Create Games with C#*. Udemy, 2020.

ДОДАТКИ

BaseAction.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class BaseAction : MonoBehaviour
{
    public static event EventHandler OnAnyActionStarted;
    public static event EventHandler OnAnyActionCompleted;
    protected Unit unit;
    protected bool isActive;
    protected Action onActionComplete;
    public abstract bool IsPlayerPlayable { get; }
    protected virtual void Awake()
    {
        unit = GetComponent<Unit>();
    }
    public abstract string GetActionName();
    public abstract void TakeAction(GridPosition gridPosition, Action onActionComplete);
    public virtual bool IsValidActionGridPosition(GridPosition gridPosition)
    {
        List<GridPosition> validGridPositionList = GetValidActionGridPositionList();
        return validGridPositionList.Contains(gridPosition);
    }
    public abstract List<GridPosition> GetValidActionGridPositionList();
    public virtual int GetActionPointsCost()
    {
        return 1;
    }
    protected void ActionStart(Action onActionComplete)
    {
        isActive = true;
        this.onActionComplete = onActionComplete;
        OnAnyActionStarted?.Invoke(this, EventArgs.Empty);
    }
    protected void ActionComplete()
    {
        isActive = false;
        onActionComplete();
        OnAnyActionCompleted?.Invoke(this, EventArgs.Empty);
    }
    public Unit GetUnit()
    {
        return unit;
    }
    public EnemyAIAction GetBestEnemyAIAction()
    {
        List<EnemyAIAction> enemyAIActionList = new List<EnemyAIAction>();
        List<GridPosition> validActionGridPositionList = GetValidActionGridPositionList();
        foreach (GridPosition gridPosition in validActionGridPositionList)
        {
            EnemyAIAction enemyAIAction = GetEnemyAIAction(gridPosition);
            enemyAIActionList.Add(enemyAIAction);
        }
        if (enemyAIActionList.Count > 0)
        {
            enemyAIActionList.Sort((EnemyAIAction a, EnemyAIAction b) => b.actionValue - a.actionValue);
            return enemyAIActionList[0];
        }
    }
}

```

```
    }  
    else  
    {  
        return null;  
    }  
}  
public abstract EnemyAIAction GetEnemyAIAction(GridPosition gridPosition);  
}
```

GrenadeAction.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class GrenadeAction : BaseAction
{
    [SerializeField] private Transform grenadeProjectilePrefab;
    public override bool IsPlayerPlayable => true;
    private int maxThrowDistance = 5;
    private void Update()
    {
        if (!IsActive)
        {
            return;
        }
    }
    public override string GetActionName()
    {
        return "Граната";
    }
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 0,
        };
    }

    public override List<GridPosition> GetValidActionGridPositionList()
    {
        List<GridPosition> validGridPositionList = new List<GridPosition>();
        GridPosition unitGridPosition = unit.GetGridPosition();
        for (int x = -maxThrowDistance; x <= maxThrowDistance; x++)
        {
            for (int z = -maxThrowDistance; z <= maxThrowDistance; z++)
            {
                GridPosition offsetGridPosition = new GridPosition(x, z);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;
                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
                {
                    continue;
                }
                int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
                if (testDistance > maxThrowDistance)
                {
                    continue;
                }
                validGridPositionList.Add(testGridPosition);
                // Debug.Log(testGridPosition);
            }
        }
        return validGridPositionList;
    }
    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        Transform grenadeProjectileTransform = Instantiate(grenadeProjectilePrefab, unit.GetWorldPosition(),
        Quaternion.identity);
        GrenadeProjectile grenadeProjectile = grenadeProjectileTransform.GetComponent<GrenadeProjectile>();
        grenadeProjectile.Setup(gridPosition, OnGrenadeBehaviorComplete);
    }
}
```

```
        //Debug.Log("GrenadeAction");
        ActionStart(onActionComplete);
    }
    private void OnGrenadeBehaviorComplete()
    {
        ActionComplete();
    }
}
```

MoveDefensiveAction.cs

```
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class MoveDefensiveAction : MoveOffensiveAction
{
    public override bool IsPlayerPlayable => false;
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        int actionValue = 0;
        int targetCountAtGridPosition = unit.GetAction<ShootAction>().GetTargetCountAtPosition(gridPosition);
        var bestPath = ShortestPathToEnemy();
        GridPosition bestPathEnd = bestPath.LastOrDefault();
        actionValue = (int)bestPathEnd.DistanceTo(gridPosition);
        if (!unit.IsEnemy) actionValue = -actionValue;
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = actionValue,
        };
    }
}
```

MoveOffensiveAction.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class MoveOffensiveAction : BaseAction
{
    [SerializeField] private int maxMoveDistance = 4;
    public override bool IsPlayerPlayable => true;
    private float moveSpeed = 10f;
    private float rotateSpeed = 10f;
    public int MaxMoveDistance => maxMoveDistance;
    public event EventHandler OnStartMoving;
    public event EventHandler OnStopMoving;
    private List<Vector3> positionList;
    private int currentPositionIndex;
    private void Update()
    {
        if (!IsActive)
        {
            return;
        }
        Vector3 targetPosition = positionList[currentPositionIndex];
        Vector3 moveDirection = (targetPosition - transform.position).normalized;
        transform.forward = Vector3.Lerp(transform.forward, moveDirection, Time.deltaTime * rotateSpeed);

        float stoppingDistance = .1f;
        if (Vector3.Distance(transform.position, targetPosition) > stoppingDistance)
        {
            transform.position += moveDirection * moveSpeed * Time.deltaTime;
        }
        else
        {
            currentPositionIndex++;
            if (currentPositionIndex >= positionList.Count)
            {
                OnStopMoving?.Invoke(this, EventArgs.Empty);
                ActionComplete();
            }
        }
    }
    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        List<GridPosition> pathGridPositionList = Pathfinding.Instance.FindPath(unit.GetGridPosition(), gridPosition,
out int pathLength);
        currentPositionIndex = 0;
        positionList = new List<Vector3>();
        foreach (GridPosition pathGridPosition in pathGridPositionList)
        {
            positionList.Add(LevelGrid.Instance.GetWorldPosition(pathGridPosition));
        }
        OnStartMoving?.Invoke(this, EventArgs.Empty);
        ActionStart(onActionComplete);
    }
    public override List<GridPosition> GetValidActionGridPositionList()
    {
        List<GridPosition> validGridPositionList = new List<GridPosition>();
        GridPosition unitGridPosition = unit.GetGridPosition();
```

```

for (int x = -maxMoveDistance; x <= maxMoveDistance; x++)
{
    for (int z = -maxMoveDistance; z <= maxMoveDistance; z++)
    {
        GridPosition offsetGridPosition = new GridPosition(x, z);
        GridPosition testGridPosition = unitGridPosition + offsetGridPosition;
        if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
        {
            continue;
        }
        if (unitGridPosition == testGridPosition)
        {
            continue;
        }
        if (LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition))
        {
            continue;
        }
        if (!Pathfinding.Instance.IsWalkableGridPosition(testGridPosition))
        {
            continue;
        }
        if (!Pathfinding.Instance.HasPath(unitGridPosition, testGridPosition))
        {
            continue;
        }

        int pathfindingDistanceMultiplier = 10;
        if (Pathfinding.Instance.GetPathLength(unitGridPosition, testGridPosition) > maxMoveDistance *
pathfindingDistanceMultiplier)
        {
            continue;
        }
        validGridPositionList.Add(testGridPosition);
        // Debug.Log(testGridPosition);
    }
}
return validGridPositionList;
}

protected List<GridPosition> ShortestPathToEnemy()
{
    GridPosition unitGridPosition = unit.GetGridPosition();
    List<List<GridPosition>> pathes = new List<List<GridPosition>>();
    foreach (var enemy in unit.GetEnemiesList())
    {
        GridPosition enemyPosition = enemy.GetGridPosition();
        List<GridPosition> pathGridPositionList = Pathfinding.Instance.FindPath(unitGridPosition, enemyPosition,
out int pathLength);
        if (pathGridPositionList != null) pathes.Add(pathGridPositionList);
    }
    List<GridPosition> shortest = pathes.OrderBy(x => x.Count).FirstOrDefault();
    if (shortest == null) return new List<GridPosition>();
    return shortest.Take(maxMoveDistance * 10).ToList();
}

public override string GetActionName()
{
    return "Pyx";
}

public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
{
    int actionValue = 0;

```

```

int targetCountAtGridPosition = unit.GetAction<ShootAction>().GetTargetCountAtPosition(gridPosition);
if (targetCountAtGridPosition == 0)
{
    var bestPath = ShortestPathToEnemy();
    bestPath.Reverse();
    int index = bestPath.IndexOf(gridPosition);
    if (index != -1)
        actionValue = (int)(10 * (bestPath.Count / index));
    else
        actionValue = 0;
}
else
{
    actionValue = targetCountAtGridPosition * 10;
}
return new EnemyAIAction
{
    gridPosition = gridPosition,
    actionValue = actionValue,
};
}
}

```

ShootAction.cs

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

public class ShootAction : BaseAction
{
    public override bool IsPlayerPlayable => true;
    public static event EventHandler<OnShootEventArgs> OnAnyShoot;
    public event EventHandler<OnShootEventArgs> OnShoot;
    public class OnShootEventArgs : EventArgs
    {
        public Unit targetUnit;
        public Unit shootingUnit;
    }
    private enum State
    {
        Aiming,
        Shooting,
        Cooloff,
    }
    [SerializeField] private LayerMask obstaclesLayerMask;
    public LayerMask ObstaclesLayerMask => obstaclesLayerMask;
    private State state;
    private int maxShootDistance = 7;
    public int MaxShootDistance => maxShootDistance;
    private float stateTimer;
    private Unit targetUnit;
    private bool canShootBullet;

    private void Update()
    {
        if (!IsActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;

        switch (state)
        {
            case State.Aiming:
                Vector3 aimDir = (targetUnit.GetWorldPosition() - unit.GetWorldPosition()).normalized;

                float rotateSpeed = 10f;
                transform.forward = Vector3.Lerp(transform.forward, aimDir, Time.deltaTime * rotateSpeed);
                break;
            case State.Shooting:
                if (canShootBullet)
                {
                    Shoot();
                    canShootBullet = false;
                }
                break;
            case State.Cooloff:
                break;
        }
        if (stateTimer <= 0f)
        {
            NextState();
        }
    }
}
```

```

    }

    private void NextState()
    {
        switch (state)
        {
            case State.Aiming:
                state = State.Shooting;
                float shootingStateTime = 0.1f;
                stateTimer = shootingStateTime;
                break;
            case State.Shooting:
                state = State.Cooloff;
                float coolOffStateTime = 0.5f;
                stateTimer = coolOffStateTime;
                break;
            case State.Cooloff:
                ActionComplete();
                break;
        }
    }

    private void Shoot()
    {
        OnAnyShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit
        });
        OnShoot?.Invoke(this, new OnShootEventArgs
        {
            targetUnit = targetUnit,
            shootingUnit = unit
        });

        targetUnit.Damage(25);
    }

    public override string GetActionName()
    {
        return "Стрільба";
    }

    public override List<GridPosition> GetValidActionGridPositionList()
    {
        GridPosition unitGridPosition = unit.GetGridPosition();
        return GetValidActionGridPositionList(unitGridPosition);
    }

    public List<GridPosition> GetValidActionGridPositionList(GridPosition unitGridPosition)
    {
        List<GridPosition> validGridPositionList = new List<GridPosition>();
        // GridPosition unitGridPosition = unit.GetGridPosition();
        for (int x = -maxShootDistance; x <= maxShootDistance; x++)
        {
            for (int z = -maxShootDistance; z <= maxShootDistance; z++)
            {
                GridPosition offsetGridPosition = new GridPosition(x, z);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;
                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
                {
                    continue;
                }
                int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
                if (testDistance > maxShootDistance)
                {

```

```

        continue;
    }
    if (!LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition))
    {
        continue;
    }
    Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(testGridPosition);
    if (targetUnit.IsEnemy == unit.IsEnemy)
    {
        continue;
    }
    Vector3 unitWorldPosition = LevelGrid.Instance.GetWorldPosition(unitGridPosition);
    Vector3 shootDir = (targetUnit.GetWorldPosition() - unitWorldPosition).normalized;
    float unitShoulderHeight = 1.7f;
    if (Physics.Raycast(
        unitWorldPosition + Vector3.up * unitShoulderHeight,
        shootDir,
        Vector3.Distance(unitWorldPosition, targetUnit.GetWorldPosition()),
        obstaclesLayerMask))
    {
        continue;
    }
    validGridPositionList.Add(testGridPosition);
    // Debug.Log(testGridPosition);
}
}
return validGridPositionList;
}
public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
{
    targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);
    state = State.Aiming;
    float aimingStateTime = 1f;
    stateTimer = aimingStateTime;
    canShootBullet = true;
    ActionStart(onActionComplete);
}
public Unit GetTargetUnit()
{
    return targetUnit;
}
public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
{
    Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);
    return new EnemyAIAction
    {
        gridPosition = gridPosition,
        actionValue = 100 + Mathf.RoundToInt((1 - targetUnit.GetHealthNormalized()) * 100f),
    };
}
public int GetTargetCountAtPosition(GridPosition gridPosition)
{
    return GetValidActionGridPositionList(gridPosition).Count; }}

```

SwordAction.cs

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

public class SwordAction : BaseAction
{
    public override bool IsPlayerPlayable => true;
    public static event EventHandler OnAnySwordHit;
    public event EventHandler OnSwordActionStarted;
    public event EventHandler OnSwordActionCompleted;
    private enum State
    {
        SwingingSwordBeforeHit,
        SwingingSwordAfterHit,
    }
    private int maxSwordDistance = 1;
    public int MaxSwordDistance => maxSwordDistance;
    private State state;
    private float stateTimer;
    private Unit targetUnit;
    private void Update()
    {
        if (!IsActive)
        {
            return;
        }
        stateTimer -= Time.deltaTime;

        switch (state)
        {
            case State.SwingingSwordBeforeHit:
                Vector3 aimDir = (targetUnit.GetWorldPosition() - unit.GetWorldPosition()).normalized;
                float rotateSpeed = 10f;
                transform.forward = Vector3.Lerp(transform.forward, aimDir, Time.deltaTime * rotateSpeed);
                break;
            case State.SwingingSwordAfterHit:
                break;
        }
        if (stateTimer <= 0f)
        {
            NextState();
        }
    }
    private void NextState()
    {
        switch (state)
        {
            case State.SwingingSwordBeforeHit:
                state = State.SwingingSwordAfterHit;
                float afterHitStateTime = 0.5f;
                stateTimer = afterHitStateTime;
                targetUnit.Damage(40);
                OnAnySwordHit?.Invoke(this, EventArgs.Empty);
                break;
            case State.SwingingSwordAfterHit:
                OnSwordActionCompleted?.Invoke(this, EventArgs.Empty);
                ActionComplete();
                break;
        }
    }
}
```

```

    }
    public override string GetActionName()
    {
        return "Меч";
    }
    public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition)
    {
        return new EnemyAIAction
        {
            gridPosition = gridPosition,
            actionValue = 200,
        };
    }
    public override List<GridPosition> GetValidActionGridPositionList()
    {
        List<GridPosition> validGridPositionList = new List<GridPosition>();
        GridPosition unitGridPosition = unit.GetGridPosition();
        for (int x = -maxSwordDistance; x <= maxSwordDistance; x++)
        {
            for (int z = -maxSwordDistance; z <= maxSwordDistance; z++)
            {
                GridPosition offsetGridPosition = new GridPosition(x, z);
                GridPosition testGridPosition = unitGridPosition + offsetGridPosition;
                if (!LevelGrid.Instance.IsValidGridPosition(testGridPosition))
                {
                    continue;
                }
                if (!LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition))
                {
                    continue;
                }
                Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(testGridPosition);
                if (targetUnit.IsEnemy == unit.IsEnemy)
                {
                    continue;
                }
                validGridPositionList.Add(testGridPosition);
                // Debug.Log(testGridPosition);
            }
        }
        return validGridPositionList;
    }
    public override void TakeAction(GridPosition gridPosition, Action onActionComplete)
    {
        targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);
        state = State.SwingingSwordBeforeHit;
        float beforeHitStateTime = 0.7f;
        stateTimer = beforeHitStateTime;
        OnSwordActionStarted?.Invoke(this, EventArgs.Empty);
        ActionStart(onActionComplete);
    }
    public int GetTargetCountAtPosition()
    {
        return GetValidActionGridPositionList().Count;
    }
}

```

EnemyAI.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class EnemyAI : MonoBehaviour
{
    [SerializeField] bool isEnemy;
    public event Action<Unit> OnActionStarted;
    public event Action OnActionFinished;
    private enum State
    {
        WaitingForEnemyTurn,
        TakingTurn,
        Busy
    }
    private State state;
    private float timer;
    private Unit currentUnit;
    private int preReward;
    private void Awake()
    {
        state = State.WaitingForEnemyTurn;
    }
    private void Start()
    {
        if (!isEnemy)
        {
            if (PlayerPrefs.GetInt("isAuto") == 0)
            {
                enabled = false;
                return;
            }
            else
            {
                TurnSystem_OnTurnChanged(null, null);
            }
        }
        TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
        UnitManager.Instance.OnAllEnemiesDead += Instance_OnAllEnemiesDead;
    }
    private void Instance_OnAllEnemiesDead(bool obj)
    {
        enabled = false;
        List<Unit> teammates = isEnemy ?
            UnitManager.Instance.GetEnemyUnitList() :
            UnitManager.Instance.GetFriendlyUnitList();
        foreach (var unit in teammates)
        {
            unit.Save();
        }
    }
    private void Update()
    {
        if (TurnSystem.Instance.IsPlayerTurn() == isEnemy)
        {
            return;
        }
        switch (state)
```

```

{
    case State.WaitingForEnemyTurn:
        break;
    case State.TakingTurn:
        timer -= Time.deltaTime;
        if (timer <= 0f)
        {
            if (TryTakeEnemyAIAction(SetStateTakingTurn))
            {
                state = State.Busy;
            }
            else
            {
                TurnSystem.Instance.NextTurn();
            }
            //TurnSystem.Instance.NextTurn();
        }
        break;
    case State.Busy:
        break;
}

}

private void SetStateTakingTurn()
{
    currentUnit.SetReward(preReward - CalculateReward());
    OnActionFinished?.Invoke();
    timer = 0.5f;
    state = State.TakingTurn;
}

private int CalculateReward()
{
    //var friends = currentUnit.GetFriendsList();
    var enemies = currentUnit.GetEnemiesList();
    //float teammatesHealth = friends.Sum(x => x.GetHealthNormalized() * 100f);
    float enemiesHealth = enemies.Sum(x => x.GetHealthNormalized() * 100f);
    // calculating for 3 vs 3
    int reward = (int)enemiesHealth;
    return reward;
}

private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
{
    if (!(!TurnSystem.Instance.IsPlayerTurn() ^ isEnemy))
    {
        state = State.TakingTurn;
        timer = 2f;
    }
}

private bool TryTakeEnemyAIAction(Action onEnemyAIActionComplete)
{
    //Debug.Log("Take Enemy AI Action");
    List<Unit> teammates = isEnemy ?
        UnitManager.Instance.GetEnemyUnitList() :
        UnitManager.Instance.GetFriendlyUnitList();
    foreach (Unit unit in teammates)
    {
        currentUnit = unit;
        preReward = CalculateReward();
        if (TryTakeEnemyAIAction(unit, onEnemyAIActionComplete))
        {
            OnActionStarted?.Invoke(unit);
            return true;
        }
    }
}

```

```

    }
}
return false;
}
private bool TryTakeEnemyAIAction(Unit enemyUnit, Action onEnemyAIActionComplete)
{
    if (!enabled) return false;

    EnemyAIAction bestEnemyAIAction = null;
    BaseAction bestBaseAction = null;
    //DefaultActionChoosing(enemyUnit, ref bestEnemyAIAction, ref bestBaseAction);
    BehaviorActionChoosing(enemyUnit, ref bestEnemyAIAction, ref bestBaseAction, out BaseBehavior
chosenBehavior);
    if (bestEnemyAIAction != null && enemyUnit.TrySpendActionPointsToTakeAction(bestBaseAction))
    {
        enemyUnit.Log(TurnSystem.Instance.Turn, chosenBehavior);
        bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition, onEnemyAIActionComplete);
        return true;
    }
    else
    {
        return false;
    }
}
private void DefaultActionChoosing(Unit enemyUnit, ref EnemyAIAction bestEnemyAIAction, ref BaseAction
bestBaseAction)
{
    foreach (BaseAction baseAction in enemyUnit.GetBaseActionArray())
    {
        if (!enemyUnit.CanSpendActionPointsToTakeAction(baseAction))
        {
            continue;
        }
        if (bestEnemyAIAction == null)
        {
            bestEnemyAIAction = baseAction.GetBestEnemyAIAction();
            bestBaseAction = baseAction;
            if (bestEnemyAIAction != null) Debug.Log(baseAction.GetType() + ": " +
bestEnemyAIAction.actionValue);
        }
        else
        {
            EnemyAIAction testEnemyAIAction = baseAction.GetBestEnemyAIAction();
            if (testEnemyAIAction != null) Debug.Log(baseAction.GetType() + ": " +
testEnemyAIAction.actionValue);
            if (testEnemyAIAction != null && testEnemyAIAction.actionValue > bestEnemyAIAction.actionValue)
            {
                bestEnemyAIAction = testEnemyAIAction;
                bestBaseAction = baseAction;
            }
        }
        // baseAction.GetBestEnemyAIAction();
    }
    //Debug.Log("BEST: " + bestBaseAction.GetType() + ": " + bestEnemyAIAction.actionValue);
    //Debug.Log("-----");
}
private void BehaviorActionChoosing(Unit enemyUnit, ref EnemyAIAction bestEnemyAIAction, ref BaseAction
bestBaseAction, out BaseBehavior chosenBehavior)
{
    enemyUnit.DebugAllBehaviors();
    chosenBehavior = ChooseBehaviorUsingSoftmax(enemyUnit.GetBaseBehaviorList());
    //BaseBehavior chosen = ChooseBehaviorUsingEpsilonGreedy(enemyUnit.GetBaseBehaviorList());
}

```

```

        int value = chosenBehavior.GetValue(out bestBaseAction);
        bestEnemyAIAction = bestBaseAction.GetBestEnemyAIAction();
        Debug.Log(">>> CHOOSE BEST: " + chosenBehavior.GetType() + " - " + bestBaseAction.GetType() + " - " + value);
    }

    public BaseBehavior ChooseBehaviorUsingSoftmax(List<BaseBehavior> baseBehaviorsList, float temperature = 1.0f)
    {
        float[] probabilities = new float[baseBehaviorsList.Count];
        float sum = 0f;

        for (int i = 0; i < baseBehaviorsList.Count; i++)
        {
            probabilities[i] = Mathf.Exp(baseBehaviorsList[i].GetValue(out _) / temperature);
            sum += probabilities[i];
        }
        for (int i = 0; i < probabilities.Length; i++)
        {
            probabilities[i] /= sum;
        }
        float randomValue = UnityEngine.Random.value;
        float cumulative = 0f;

        for (int i = 0; i < baseBehaviorsList.Count; i++)
        {
            cumulative += probabilities[i];
            if (randomValue <= cumulative)
            {
                return baseBehaviorsList[i];
            }
        }
        return baseBehaviorsList[baseBehaviorsList.Count - 1];
    }

    public BaseBehavior ChooseBehaviorUsingEpsilonGreedy(List<BaseBehavior> baseBehaviorsList, float epsilon = 0.1f)
    {
        float randomValue = UnityEngine.Random.value;

        if (randomValue < epsilon)
        {
            int randomIndex = UnityEngine.Random.Range(0, baseBehaviorsList.Count);
            return baseBehaviorsList[randomIndex];
        }
        else
        {
            BaseBehavior bestBehavior = baseBehaviorsList[0];
            int bestValue = bestBehavior.GetValue(out _);

            for (int i = 1; i < baseBehaviorsList.Count; i++)
            {
                int value = baseBehaviorsList[i].GetValue(out _);
                if (value > bestValue)
                {
                    bestBehavior = baseBehaviorsList[i];
                    bestValue = value;
                }
            }
            return bestBehavior;
        }
    }
}

```

```
// greater value means player plays more aggressive now in comparison to average
public int CompareAggressiveness()
{
    int predictedAverageHealth = Prediction.Instance.GetAverageTotalPlayerHealth(TurnSystem.Instance.Turn-1);
    var currentAverageHealth =
UnitManager.Instance.GetFriendlyUnitList()[0].GetHeuristic<TotalHealthHeuristic>().GetValue();
    return predictedAverageHealth - currentAverageHealth;
}
}
```