Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

Биндю Семена Костянтиновича
Прізвище, ім'я, по батькові студента(ки)

на здобуття ступеня вищої освіти Бакалавра

Покрокова гра-стратегія з використанням алгоритму Minimax
Назва теми

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ. 200243.01.02.ПЗ

Виконав студент IV курсу, група ІПЗ-20-1 _____ Семен БИНДЮ
Підпис Ім'я. ПРІЗВИЩЕ

Керівник асистент _____ В'ячеслав БОЙКО
Науковий ступінь, вчене звання Підпис Ім'я. ПРІЗВИЩЕ

Нормоконтролер асистент _____ В'ячеслав БОЙКО
Посада Підпис Ім'я. ПРІЗВИЩЕ

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Леонід БЕДРАТЮК
Підпис Ім'я. ПРІЗВИЩЕ

21 червня 2024 р.

Хмельницький 2024

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Перший (бакалаврський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
02 01 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Биндю Семену Костянтиновичу
Прізвище, ім'я, по батькові студента

1. Тема роботи Покрокова гра-стратегія з використанням алгоритму Minimax

Керівник роботи Бойко В'ячеслав Олександрович, асистент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 08.01.2024 р. № 6-КП
2. Строк подання студентом роботи на кафедру 01.06.2024 р.
3. Вихідні дані до роботи Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Аналіз предметної області та постановка задачі
Проєктування програмного забезпечення
Програмна реалізація та тестування програмного забезпечення
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Три креслення:
1. Діаграма класів
2. DFD-діаграма
3. Діаграма процесів гри

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Бойко В. О., асистент	13.06.2024	21.06.2024
Антиплагіат	Форкун Ю. В., доцент	30.05.24	06.06.24

7. Дата видачі завдання « 02 » січня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1 Дослідження суб'єкта господарювання відповідно до теми КвР; визначення задач та вимог до ПЗ, розроблення технічного завдання	1-й – 3-й тиждень	
2 Проєктування та розроблення загальної архітектури і структури ПЗ, інтерфейсу користувача; вибір засобів реалізації ПЗ	4-й – 8-й тиждень	
3 Програмна реалізація та тестування ПЗ	9-й – 12-й тиждень	
4 Написання тексту пояснювальної записки та розроблення графічних матеріалів	13-й – 14-й тиждень	
Попередній захист КвР	Травень	згідно з графіком
6 Остаточне коригування КвР з урахуванням зауважень керівника; оформлення КвР як документа відповідно до вимог	15-й тиждень	
Отримання супровідних документів (відгуку керівника, рецензії, довідок про перевірку на плагіат); проходження нормоконтролю; брошурування (зшиття) пояснювальної записки	16-й тиждень	
Здача КвР на кафедру; підготовка КвР для розміщення у репозитарії університету; підготовка до захисту та захист КвР	З 01.06.2024	

Студент _____ Семен БИНДЮ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи _____ В'ячеслав БОЙКО
Підпис Ім'я, ПРІЗВИЩЕ

АНОТАЦІЯ

Тема кваліфікаційоної роботи: «Покрокова гра-стратегія з використанням алгоритму Minimax».

Автор роботи: Биндю Семен Костянтинович.

Керівник роботи: Бойко В'ячеслав Олександрович.

Пояснювальна записка: 64 с., 23 рис., 2 дод., 30 джерел

Графічна частина: 3 креслення ф. А3.

Метою кваліфікаційної роботи є розроблення програмного забезпечення для покрокової гри–стратегії з використанням алгоритму Мінімакс.

У кваліфікаційні роботі проаналізовано предметну область покрокових ігор-стратегій, досліджено особливості алгоритму Мінімакс та його застосування в ігрових системах, розглянуто існуючі програмні рішення та бібліотеки для розробки ігор-стратегій.

Для реалізації програмного забезпечення обрано мову програмування C# та ігровий рушій Unity, які забезпечують необхідну функціональність та продуктивність для створення покрокової гри-стратегії.

В результаті проєктування здійснено програмну реалізацію покрокової гри–стратегії з використанням алгоритму Мінімакс, яка відповідає визначеним вимогам та забезпечує зручний інтерфейс для користувачів.

21.06.2024
Дата

Підпис

ВІДОМІСТЬ ДОКУМЕНТІВ

№ рядка	Формат	Позначення документа	Найменування документа	К–сть аркушів	№ екз.	Примітка
			Текстові документи			
1	А4	КвРІПЗ. 200243.01.02.ПЗ	Пояснювальна	63		
2	А4		Завдання на	1		
			ційну роботу			
3	А4		Анотація	1		
			Графічні документи			
4	А3	КвРІПЗ. 200243.01.02.Е8	Діаграма класів	1		
5	А3	КвРІПЗ. 200243.01.02.Е8	DFD-діаграма	1		
6	А3	КвРІПЗ. 200243.01.02.Е8	Діаграма процесів	1		

Змн.	Арк.	№ докум.	Підпис	Дата	КвРІПЗ. 200243.01.02.ВД		
Виконав		Биндю С. К.		21.06	Покрокова гра-стратегія з використанням алгоритму Minimax	Літ.	Арк. 1 / Аркушів 1
Керівник		Бойко В. О.		21.06			
Н. контр.		Бойко В. О.		21.06	Відомість документів	ХНУ, ІПЗ-20-1	
Зав. каф.		Бедратюк Л.П.		21.06			

ЗМІСТ

Вступ 6
1 Дослідження предметної області та постановка задачі 8
1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей 8
1.2 Аналіз наявного програмно–технічного забезпечення предметної області 9
1.3 Визначення вимог до програмного забезпечення та розробка технічного завдання 11
2 Проектування програмного забезпечення 14
2.1 Архітектурне проектування 14
2.1.1 Аналіз архітектури покрокових ігор–стратегій 14
2.1.2 Аналіз існуючих ігрових рушіїв та бібліотек 15
2.2 Детальне проектування 17
2.2.1 Ігрове поле 19
2.2.2 Юніт 20
2.2.3 Вибір алгоритму пошуку шляху 23
2.3 Середовище програмування 25
3 Програмна реалізація та тестування 28
3.1 Рух та вибір юнітів 28
3.2 Grid система 33

3.3 Дії та UI 38
3.4 Вороги та битва..... 41
3.5 Юніт–тестування..... 48
3.6 Робота з юніт–тестами в середовищі Unity..... 50
3.7 Написання юніт–тесту..... 56
Висновки..... 59
Перелік джерел посилання... 61
Додаток А 64
Додаток Б..... 112

ВСТУП

У прогресивному сучасному світі комп’ютерні ігри відіграють не лише роль розваги, але й викликають реальний інтерес як серед молоді, так і серед дорослих. Цей феномен не лише розвиває та розважає, але і ставить перед гравцями високі вимоги щодо рівня інтелектуальної складності геймплею. Здатність адаптуватися до різних сценаріїв та приймати стратегічно обґрунтовані рішення стає критичною для інтелектуальних агентів у гральній індустрії. Серед ключових аспектів розвитку інтелектуальних агентів для стратегічних ігор вирішальне значення має розробка алгоритмів прийняття рішень. У цьому контексті алгоритм Minimax виступає як один із найефективніших та широко використовуваних методів в галузі прийняття рішень для інтелектуальних агентів у стратегічних іграх. Теоретичні основи Minimax. Minimax – це алгоритм, розроблений для ігор з повністю інформованими учасниками, які мають повну інформацію про стан гри. Основна ідея полягає в тому, щоб агент максимізував свій виграш, при цьому мінімізуючи можливі втрати. Це досягається аналізом всіх можливих ходів та їх наслідків, що дозволяє створити оптимальну стратегію, враховуючи дії опонента. Мета даного дослідження – глибокий аналіз та оцінка ефективності алгоритму Minimax у контексті покрокових стратегій. Специфічні цілі включають вивчення теоретичних основ алгоритму Minimax, його вплив на стратегічні аспекти гри, розгляд переваг та недоліків у різних умовах гри, а також розгляд можливостей оптимізації та розширення для досягнення кращих результатів у реальних сценаріях геймплею. Теоретичний аналіз Minimax. Minimax базується на принципі «мінімаксу», де агент максимізує свій виграш, а опонент – мінімізує його. Цей алгоритм широко використовується в галузі штучного інтелекту та геймдизайну, забезпечуючи інтелектуальним агентам оптимальні стратегії.

Вплив Minimax на стратегічний аспект гри. Minimax визначається як ефективний метод прийняття рішень, що дозволяє агентам аналізувати всі можливі ходи та їх наслідки, максимізуючи виграш та мінімізуючи можливі

втрати. Це сприяє розвитку стратегій, які ефективно пристосовуються до дій опонента та забезпечують оптимальний результат. Переваги та недоліки Minimax у різних умовах гри. Попри свою ефективність, Minimax має свої обмеження. У іграх з великою кількістю можливих ходів він може стати витратним з точки зору обчислювальних ресурсів. Крім того, алгоритм передбачає, що всі гравці мають повну інформацію про гру, що не завжди відповідає реальності. Актуальність обраної теми визначається швидким розвитком галузі комп'ютерних ігор та штучного інтелекту. Комп'ютерні ігри стають все більш складними та реалістичними, що вимагає від інтелектуальних агентів у гральній індустрії високого рівня адаптації та прийняття оптимальних стратегій. Вибір оптимальних ходів у грі є ключовим завданням, а алгоритм Minimax в цьому контексті залишається одним із найбільш перспективних інструментів. Одночасно з цим постає завдання оптимізації алгоритмів прийняття рішень для підтримки геймплею в реальному часі. Minimax, як алгоритм, що спрямований на повністю інформованих учасників, відповідає цим вимогам, але йому потрібна подальша адаптація та оптимізація для ефективного використання в сучасних геймдизайн проектах. Можливості оптимізації та розширення Minimax. Оптимізація та розширення алгоритму Minimax можуть бути ключовими для підвищення його ефективності у реальних геймплей сценаріях. Можливість адаптації алгоритму до конкретних умов гри та оптимізація його обчислювальних аспектів є перспективним напрямком дослідження.

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Змістовний аналіз предметної області, її структурних та функціональних особливостей

Покрокові ігри – це ігри, в яких гравці по черзі роблять ходи, змінюючи стан гри. В таких іграх важливим є розробка стратегії, яка дозволить гравцеві досягти перемоги.

Одним із методів розробки стратегій для покрокових ігор є алгоритм Minimax. Алгоритм Minimax працює наступним чином:

– Гравець вибирає хід, який мінімізує максимальний результат, який може отримати його суперник.

– Суперник вибирає хід, який максимізує мінімальний результат, який може отримати гравець.

Цей процес повторюється до тих пір, поки не буде досягнуто кінцевого стану гри.

В якості прикладу розглянемо гру «Крестики–нолики». У цій грі два гравці, X і O, по черзі ставлять свої фігури на дошці 3x3. Гравець, який першим поставить три фігури свого кольору в ряд, стовпчик або діагональ, перемагає.

Стратегія гравця X, яка базується на алгоритмі Minimax, може бути наступною:

– Якщо на дошці немає жодної фігури, гравець X ставить фігуру в центр дошки.

– Якщо на дошці є одна фігура гравця X, гравець X ставить фігуру в кут, який не зайнятий фігурами гравця O.

– Якщо на дошці є дві фігури гравця X, гравець X ставить фігуру в вільний клітинку, яка дозволяє йому поставити три фігури свого кольору в ряд, стовпчик або діагональ.

– Якщо на дошці є дві фігури гравця О, гравець Х ставить фігуру в вільну клітинку, яка блокує можливість гравця О поставити три фігури свого кольору в ряд, стовпчик або діагональ.

Ця стратегія є досить ефективною і дозволяє гравцеві Х досягти перемоги в більшості випадків.

Алгоритм Minimax можна використовувати для розробки стратегій для будь–яких покрокових ігор. Однак, ефективність цієї стратегії залежить від складності гри. У простих іграх, таких як «Крестики–нолики», алгоритм Minimax може забезпечити гравцеві перемогу в більшості випадків. У складніших іграх, таких як шахи або го, алгоритм Minimax може бути менш ефективним.

Існують різні методи, які дозволяють покращити ефективність алгоритму Minimax. Одним із таких методів є використання евристичного пошуку. Евристичні алгоритми використовують додаткову інформацію про гру, щоб скоротити час пошуку найкращого ходу.

Іншим методом, який дозволяє покращити ефективність алгоритму Minimax, є використання нейронних мереж. Нейронні мережі можуть навчитися прогнозувати результат гри для різних можливих ходів. Це дозволяє гравцеві вибирати хід, який має найбільший шанс на перемогу.

Алгоритм Minimax є потужним інструментом, який можна використовувати для розробки стратегій для покрокових ігор. Однак, ефективність цієї стратегії залежить від складності гри. У простих іграх, алгоритм Minimax може забезпечити гравцеві перемогу в більшості випадків. У складніших іграх, алгоритм Minimax може бути менш ефективним.

1.2 Аналіз наявного програмно–технічного забезпечення предметної області

Наявне програмно–технічне забезпечення для ігор–стратегій з використанням алгоритму мінімакс можна розділити на кілька категорій:

– Програми для гри в конкретні ігри. До цієї категорії належать програми, які розроблені для гри в конкретну гру–стратегію, наприклад, шахи, шашки, го, реверси тощо. Такі програми, як правило, є дуже потужними і можуть перемагати навіть найсильніших людей–гравців.

– Програмні бібліотеки для ігор–стратегій. До цієї категорії належать бібліотеки, які надають програмістам інструменти для розробки власних програм для ігор–стратегій. Такі бібліотеки, як правило, містять реалізацію алгоритму мінімакс, а також інші корисні функції, такі як генерація випадкових позицій, перевірка законності ходів тощо.

– Онлайн–сервіси для ігор–стратегій. До цієї категорії належать сервіси, які дозволяють гравцям грати в ігри–стратегії онлайн. Такі сервіси, як правило, використовують алгоритм мінімакс для визначення найкращого ходу для кожної зі сторін у грі.

Огляд деяких програмних продуктів для ігор–стратегій з використанням алгоритму мінімакс:

- Програми для гри в конкретні ігри:
  - ChessBase Fritz – одна з найпотужніших програм для гри в шахи. Вона використовує алгоритм мінімакс для визначення найкращого ходу в будь–якій шаховій позиції.
  - Komodo – ще одна потужна програма для гри в шахи. Вона використовує алгоритм мінімакс із деякими покращеннями.
  - Leela Chess Zero – безкоштовна програма для гри в шахи, яка навчена на основі даних із шахових ігор. Вона використовує алгоритм мінімакс із деякими покращеннями.
- Програмні бібліотеки для ігор–стратегій:
  - Minimax – бібліотека для ігор–стратегій, яка містить реалізацію алгоритму мінімакс.
  - PyGame – бібліотека для розробки ігор, яка містить реалізацію алгоритму мінімакс.

- Unity – фреймворк для розробки ігор, який містить реалізацію алгоритму мінімакс.

– Онлайн–сервіси для ігор–стратегій:

- Lichess – безкоштовний онлайн–сервіс для гри в шахи. Він використовує алгоритм мінімакс для визначення найкращого ходу для кожної зі сторін у грі.
- Chess.com – платний онлайн–сервіс для гри в шахи. Він використовує алгоритм мінімакс для визначення найкращого ходу для кожної зі сторін у грі.
- ICC – платний онлайн–сервіс для гри в шахи. Він використовує алгоритм мінімакс для визначення найкращого ходу для кожної зі сторін у грі.

1.3 Визначення вимог до програмного забезпечення та розробка технічного завдання

Цільовою аудиторією цього програмного забезпечення є гравці в ігри стратегії.

Основними вимогами до програмного забезпечення є такими:

– Гра повинна бути покрокової. Це означає, що гравці повинні робити ходи по одному, і що гравець, який робить хід, не бачить ходу іншого гравця до тих пір, поки він не буде зроблений.

– Гра повинна використовувати алгоритм мінімакс для визначення найкращого ходу для кожної зі сторін. Цей алгоритм є потужним методом пошуку оптимального рішення в іграх з нульовою сумою.

Функціональні вимоги до програмного забезпечення описують, що програмне забезпечення має робити.Гра повинна підтримувати кілька різних ігор стратегії. Прикладами таких ігор є шахи, шашки, го, реверси тощо.

– Гра повинна дозволяти гравцям грати проти комп'ютера або один

проти одного.

– Гра повинна підтримувати різні рівні складності. Це дозволить гравцям вибирати рівень складності, який відповідає їхньому рівню майстерності.

– Гра повинна мати інтерфейс користувача, який є простим у використанні.

Нефункціональні вимоги до програмного забезпечення описують, як програмне забезпечення має працювати.

Гра повинна бути надійною. Це означає, що вона повинна працювати коректно і не повинна зависати або видавати помилки.

– Гра повинна бути доступною. Це означає, що вона повинна бути доступна для широкого кола користувачів, незалежно від їхнього рівня комп'ютерної підготовки.

– Гра повинна бути швидкою. Це означає, що вона повинна швидко обробляти ходи гравців.

Технічні обмеження до програмного забезпечення описують технічні фактори, які необхідно враховувати при розробці програмного забезпечення.

– Гра повинна бути написана на мові програмування C#.

– Гра повинна використовувати ігровий рушій Unity для відображення графіки та анімації.

– Гра повинна бути розроблена для операційних систем Windows, macOS та Linux.

Вимоги до програмного забезпечення були оцінені за такими критеріями:

– Повна повнота – вимоги є повним, тобто вони описують все, що програмне забезпечення має робити.

– Чіткість – вимоги є чіткими, тобто вони зрозумілі для всіх зацікавлених сторін.

– Несуперечливість – вимоги не є суперечливими, тобто вони узгоджені між собою.

– Тестабельність – вимоги є тестабельними, тобто вони дозволяють розробникам та тестувальникам перевірити, чи відповідає програмне забезпечення цим вимогам.

Вимоги були оцінені як високоякісні. Вони є повним, чіткими, несуперечливими та тестабельними.

Вимоги до програмного забезпечення для реалізації покрокової гри стратегії з використанням алгоритму мінімакс були визначені та документовані. Вимоги є високоякісними і забезпечують, щоб програмне забезпечення відповідало потребам зацікавлених сторін.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Архітектурне проектування

2.1.1 Аналіз архітектури покрокових ігор–стратегій

Покрокові ігри–стратегії відрізняються від інших жанрів ігор своєю специфічною структурою та принципами взаємодії гравців. Зазвичай, вони мають чітко визначені фази ходу, де кожен гравець по черзі приймає рішення, а результат цих рішень впливає на стан гри та можливості наступних ходів.

Одним з ключових аспектів архітектури покрокових ігор–стратегій є управління станом гри. Це включає в себе збереження інформації про позиції юнітів, ресурси, доступні дії та інші параметри, що визначають поточний стан гри. Для ефективного управління станом гри використовуються різні структури даних та алгоритми, що забезпечують швидкий доступ та оновлення інформації.

Важливим елементом архітектури покрокових ігор–стратегій є також реалізація механізмів прийняття рішень гравцями та їх впливу на стан гри. Це може включати в себе обробку введення користувача, перевірку допустимості дій, розрахунок наслідків та оновлення стану гри відповідно до прийнятих рішень.

Для реалізації штучного інтелекту (AI) в покрокових іграх–стратегіях використовуються різноманітні алгоритми та методи, такі як дерево пошуку, евристичні функції оцінки, машинне навчання тощо. Алгоритм Мінімакс, що є одним з класичних методів прийняття рішень в іграх з двома гравцями, дозволяє AI вибрати оптимальний хід, враховуючи можливі ходи суперника та їх наслідки.

Отже, архітектура покрокових ігор–стратегій включає в себе різноманітні компоненти та механізми, що забезпечують управління станом гри, взаємодію гравців та реалізацію штучного інтелекту. Для реалізації мети даного дипломного проєкту було обрано архітектуру покрокової гри–стратегії з використанням алгоритму Мінімакс, що дозволяє створити цікаву та інтелектуальну гру з високим рівнем реіграбельності.

2.1.2 Аналіз існуючих ігрових рушіїв та бібліотек

Існує безліч ігрових рушіїв та бібліотек, що надають інструменти та функціональність для розробки ігор різних жанрів. Серед популярних ігрових рушіїв можна виділити Unity, Unreal Engine, Godot тощо. Кожен з них має свої переваги та недоліки, тому вибір конкретного рушія залежить від вимог проєкту, досвіду розробників та інших факторів.

Unity є одним з найпопулярніших ігрових рушіїв, що пропонує широкий спектр інструментів та можливостей для розробки ігор різних жанрів, включаючи покрокові ігри–стратегії. Він має зручний інтерфейс, велику спільноту розробників та безліч готових ассетів та плагінів, що полегшують та прискорюють процес розробки. Unity - це кросплатформенний ігровий рушій, розроблений компанією Unity Technologies, вперше анонсований і випущений у червні 2005 року на конференції Apple Worldwide Developers Conference як ігровий рушій для Mac OS X. З того часу рушій поступово розширювався для підтримки різноманітних настільних, мобільних, консольних платформ, платформ доповненої та віртуальної реальності. Він особливо популярний для розробки мобільних ігор для iOS та Android, вважається простим у використанні для розробників-початківців та популярний для розробки інді-ігор. Рушій можна використовувати для створення тривимірних (3D) і двовимірних (2D) ігор, а також інтерактивних симуляцій. Рушій був прийнятий в інших галузях, окрім відеоігор, таких як кіно, автомобільна промисловість, архітектура, інженерія, будівництво та Збройні сили США. Ігровий рушій Unity був запущений у 2005 році з метою "демократизувати" розробку ігор, зробивши її доступною для більшої кількості розробників. Він був показаний на Всесвітній конференції розробників 2005 року Скоттом Форстолом на Mac OS X. Наступного року Unity посів друге місце в категорії "Найкраще використання графіки Mac OS X" на конкурсі Apple Design Awards, організованому компанією Apple Inc.[12]. Спочатку Unity був випущений для Mac OS X, а згодом була додана підтримка

Microsoft Windows та веб-браузерів.

Unreal Engine є потужним ігровим рушієм, що використовується для створення високоякісних ігор з реалістичною графікою. Він має власну мову програмування (C++), що забезпечує високу продуктивність та гнучкість, але вимагає більшого досвіду від розробників. Unreal Engine (UE) - це серія комп'ютерної 3D-графіки та ігрових рушіїв, розроблених компанією Epic Games, вперше продемонстрована у відеогрі-шутері від першої особи Unreal 1998 року. Спочатку розроблений для комп'ютерних шутерів від першої особи, він з тих пір використовується в різних жанрах ігор і був прийнятий іншими індустріями, особливо в кіно і телеіндустрії. Unreal Engine написаний на C++ і відрізняється високим ступенем портативності, підтримуючи широкий спектр настільних, мобільних, консольних платформ і платформ віртуальної реальності. Останнє покоління, Unreal Engine 5, було запущено у квітні 2022 року. Його вихідний код доступний на GitHub, а комерційне використання надається на основі моделі роялті: Epic стягує 5% від доходу понад 1 мільйон доларів США, що не стягується з ігор, опублікованих в Epic Games Store. Epic включила в рушій функції придбаних компаній, таких як Quixel, що, як вважається, сприяло зростанню доходів Fortnite. У 2014 році Unreal Engine був названий "найуспішнішим рушієм для відеоігор" за версією Книги рекордів Гіннеса[2].

Godot є безкоштовним ігровим рушієм з відкритим вихідним кодом, що пропонує простий та інтуїтивно зрозумілий інтерфейс та власну мову програмування (GDScript). Він підходить для розробки невеликих та середніх ігор, але може бути недостатньо потужним для створення великих та складних проєктів. Godot – це кросплатформенний, безкоштовний ігровий рушій з відкритим вихідним кодом, випущений під дозвільною ліцензією MIT. Спочатку він був розроблений у Буенос-Айресі аргентинськими розробниками програмного забезпечення Хуаном Лінецьким та Аріелем Манзуром[6] для кількох компаній у Латинській Америці до його публічного релізу у 2014 р.[7] Середовище розробки

працює на багатьох платформах і може експортуватися ще на кілька. Воно призначене для створення 2D і 3D ігор для ПК, мобільних і веб-платформ, а також може використовуватися для розробки неігрового програмного забезпечення, зокрема редакторів. Godot дозволяє розробникам відеоігор створювати як 3D, так і 2D ігри, використовуючи різні мови програмування, такі як C++, C# та GDScript. Він використовує ієрархію вузлів для полегшення процесу розробки. Класи можуть бути похідними від типу вузла для створення більш спеціалізованих типів вузлів, які успадковують поведінку. Вузли організовані всередині "сцен", які є багаторазовими, екземплярами, успадкованими та вкладеними групами вузлів. Усі ресурси гри, включно зі скриптами та графічними ресурсами, зберігаються у файловій системі комп'ютера (а не в базі даних). Таке рішення для зберігання призначене для полегшення співпраці між командами розробників ігор, що використовують системи контролю версій програмного забезпечення.

Окрім ігрових рушіїв, існують також спеціалізовані бібліотеки, що надають функціональність для розробки окремих аспектів ігор, таких як штучний інтелект, фізика, графіка тощо. Наприклад, бібліотека Minimax AI надає готові алгоритми та структури даних для реалізації алгоритму Мінімакс в іграх.

Для реалізації програмного забезпечення в рамках даного дипломного проєкту було обрано ігровий рушій Unity та мову програмування C#, що забезпечують необхідну функціональність та продуктивність для створення покрокової гри–стратегії з використанням алгоритму Мінімакс.

2.2 Детальне проектування

Для реалізації програмного забезпечення необхідно визначити основні компоненти та їх взаємодію. Враховуючи специфіку покрокових ігор–стратегій, важливо забезпечити ефективне управління станом гри, обробку ходів гравців та реалізацію алгоритму Мінімакс. Програмна система буде складатися з наступних основних компонентів: ігрового рушія (Game Engine), менеджера стану гри (Game

State Manager), модуля логіки гри (Game Logic Module), модуля штучного інтелекту (AI Module) та інтерфейсу користувача (User Interface).

Ігровий рушій буде основним компонентом, відповідальним за управління станом гри, обробку введення користувача, візуалізацію ігрового світу та взаємодію з іншими компонентами системи. Менеджер стану гри буде відповідальним за збереження та оновлення інформації про поточний стан гри, такий як позиції юнітів, ресурси, доступні дії тощо. Модуль логіки гри буде містити правила гри, механізми взаємодії юнітів, розрахунок наслідків дій та інші аспекти, що визначають логіку гри. Модуль штучного інтелекту буде відповідальним за реалізацію алгоритму Мінімакс та інших методів прийняття рішень для комп'ютерного опонента. Інтерфейс користувача забезпечить взаємодію гравця з грою, відображення ігрового світу, меню, кнопок управління тощо.

Взаємодія між компонентами буде здійснюватися за допомогою подій та повідомлень. Наприклад, коли гравець робить хід, інтерфейс користувача надсилає повідомлення ігровому рушію, який оновлює стан гри та передає інформацію модулю логіки гри для розрахунку наслідків. Модуль штучного інтелекту отримує інформацію про стан гри та використовує алгоритм Мінімакс для вибору оптимального ходу.

Для реалізації програмного забезпечення буде використано мову програмування C# та ігровий рушій Unity. Unity надає широкий спектр інструментів та можливостей для розробки ігор, включаючи вбудовані функції для обробки введення користувача, фізики, графіки та анімації. C# є об'єктно–орієнтованою мовою програмування, що дозволяє створювати структурований та легко підтримуваний код.

2.2.1. Ігрове поле

Ефективне зберігання та обробка даних про ігровий світ, юнітів, їх характеристики та взаємодії є ключовим аспектом розробки покрокової стратегії.

Для цього необхідно ретельно спроектувати структури даних, які забезпечать швидкий доступ до інформації, ефективне оновлення стану гри та зручність роботи з даними для розробників.

Ігрове поле(Grid)у покроковій стратегії зазвичай представляється у вигляді сітки (grid). Кожна клітинка сітки відповідає певному місцю на полі бою та містить інформацію про тип місцевості (рівнина, ліс, гори, вода тощо), наявність перешкод (будівлі, скелі, річки тощо), розташування юнітів та інші параметри, що впливають на ігровий процес.

Для представлення ігрового поля можна використовувати двовимірний масив або список списків. Кожен елемент масиву або списку відповідатиме одній клітинці поля та міститиме всю необхідну інформацію про неї. Наприклад, структура даних для клітинки може виглядати так, як показано на рисунку 2.1.

```
class Cell {
    TerrainType terrainType; // Тип місцевості
    bool isOccupied;         // Чи зайнята клітинка юнітом
    Unit occupyingUnit;      // Посилання на юніта, що займає клітинку (якщо є)
    bool isObstacle;         // Чи є на клітинці перешкода
    // ... інші параметри ...
}
```

Рисунок 2.1 – Код класу Cell

Використання двовимірного масиву дозволяє легко отримувати доступ до клітинок за їх координатами, що спрощує реалізацію алгоритмів переміщення юнітів, обчислення видимості та інших ігрових механік. Крім того, масив забезпечує ефективне використання пам'яті, оскільки всі клітинки зберігаються послідовно.

Однак, якщо розмір ігрового поля може змінюватися динамічно, або якщо потрібно зберігати додаткову інформацію про клітинки (наприклад, список юнітів, що знаходяться в радіусі дії), то використання списку списків може бути більш гнучким рішенням.

Вибір конкретної структури даних для представлення ігрового поля

залежить від особливостей гри та вимог до продуктивності. У більшості випадків двовимірний масив є достатнім та ефективним рішенням.

2.2.2. Юніт

Структура даних «Юніт» є однією з ключових у покроковій стратегії, оскільки вона зберігає всю необхідну інформацію про кожен ігровий юніт. Ця інформація використовується для відображення юнітів на полі бою, розрахунку їх дій та взаємодій, а також для прийняття рішень алгоритмом Мінімакс.

Поля структури даних «Юніт» можуть включати:

– Тип юніта (UnitType): Визначає тип юніта (піхота, танк, літак тощо). Це поле може бути переліченням (enum) або цілочисельним значенням.

– Характеристики (Stats): Структура або клас, що містить різні характеристики юніта, такі як:

  – Здоров'я (Health): Поточний та максимальний рівень здоров'я юніта.

  – Атака (Attack) – сила атаки юніта.

  – Захист (Defense) – рівень захисту юніта.

  – Швидкість переміщення (MovementSpeed) – кількість клітинок, на яку юніт може переміститися за один хід.

  – Дальність атаки (AttackRange) – максимальна відстань, на якій юніт може атакувати.

  – Інші характеристики, залежно від особливостей гри (наприклад, тип броні, мораль, спеціальні здібності тощо).

– Поточна позиція (CurrentPosition) – координати (x, y) клітинки на полі бою, яку займає юніт.

– Стан (State) – поточний стан юніта (активний, поранений, знищений тощо). Це поле може бути переліченням або цілочисельним значенням.

– Фракція (Faction) – визначає, до якої фракції належить юніт (гравець, комп'ютер, нейтральна тощо).

– Додаткові параметри – залежно від особливостей гри, структура даних «Юніт» може містити й інші параметри, такі як досвід, рівень, інвентар, список доступних дій тощо. Приклад структури даних «Юніт» на C# зобюраєжено на рисинку 2.2.

```C#
public class Unit {
    public UnitType UnitType;
    public Stats Stats;
    public Vector2Int CurrentPosition;
    public UnitState State;
    public Faction Faction;
    // ... інші параметри ...
}
```

Рисунок 2.2 – Код класу Unit

Структура даних «Юніт» є основою для реалізації багатьох ігрових механік, таких як переміщення юнітів, бій, отримання досвіду та підвищення рівня, використання спеціальних здібностей тощо. Ефективне проектування та реалізація цієї структури дозволить забезпечити гнучкість та масштабованість ігрової системи.

Тип юніта (UnitType) є одним з ключових параметрів, що визначає роль та функціональність юніта у грі. Він може бути представлений у вигляді перелічення (enum) або цілочисельного значення. Вибір конкретного способу представлення залежить від особливостей гри та уподобань розробника.

Використання перелічення дозволяє явно визначити всі можливі типи юнітів, що робить код більш читабельним та зрозумілим (рис 2.3).

```
public enum UnitType {
    Infantry,
    Tank,
    Airplane,
    // ... інші типи юнітів ...
}
```

Рисунок 2.3 – Код класу UnitType

Використання цілочисельних значень для представлення типів юнітів може бути більш компактним та ефективним з точки зору використання пам'яті. Однак, такий підхід вимагає додаткових коментарів або документації для пояснення, яке значення відповідає якому типу юніта (рис 2.4).

```
public const int UNIT_TYPE_INFANTRY = 0;
public const int UNIT_TYPE_TANK = 1;
public const int UNIT_TYPE_AIRPLANE = 2;
// ... інші типи юнітів ...
```

Рисунок 2.4 – Представлення типу юнітів

Поточна позиція юніта на полі бою визначається координатами (x, y) клітинки, яку він займає. Для зберігання координат можна використовувати структуру Vector2Int з Unity або власну структуру з двома цілочисельними полями.

Стан юніта відображає його поточний стан у грі (активний, поранений, знищений тощо). Він може бути представлений у вигляді перелічення або цілочисельного значення, аналогічно типу юніта.

Фракція визначає, до якої сторони конфлікту належить юніт (гравець, комп'ютер, нейтральна тощо). Це поле може бути переліченням або цілочисельним значенням.

Залежно від особливостей гри, структура даних «Юніт» може містити й інші параметри, такі як досвід, рівень, інвентар, список доступних дій тощо. Ці параметри можуть бути представлені у вигляді окремих полів або вкладених структур/класів.

2.2.3 Вибір алгоритму пошуку шляху

Алгоритми пошуку шляху є невід'ємною частиною багатьох ігор, особливо стратегій та RPG, де персонажі або юніти повинні переміщуватися складним ігровим світом. У контексті покрокової стратегії, ці алгоритми використовуються для визначення оптимального шляху юніта з однієї точки на полі бою до іншої, враховуючи різноманітні перешкоди та особливості ландшафту. Існує кілька популярних алгоритмів пошуку шляху, кожен з яких має свої переваги та недоліки: Алгоритм Дейкстри (Dijkstra's Algorithm), Алгоритм A (A Search), Алгоритм пошуку в ширину (Breadth–First Search, BFS), Алгоритм пошуку в глибину (Depth–First Search, DFS)

Алгоритм Дейкстри – це класичний алгоритм пошуку найкоротшого шляху в графі. Він був розроблений нідерландським вченим Едсгером Дейкстрою в 1956 році і досі широко використовується в різних галузях, включаючи розробку ігор. Алгоритм Дейкстри починає з початкової вершини графа та поступово розширює множину вершин, для яких відома найкоротша відстань від початку. На кожному кроці алгоритм вибирає вершину з найменшою відстанню серед тих, що ще не були розглянуті, та оновлює відстані до її сусідів. Цей процес повторюється, поки не будуть знайдені найкоротші шляхи до всіх вершин графа. Алгоритм Дейкстри завжди знаходить найкоротший шлях між двома вершинами, якщо такий шлях існує. Алгоритм має відносно просту логіку та може бути легко реалізований на різних мовах програмування. Алгоритм має часову складність O(V^2), де V – кількість вершин у графі. Це означає, що для великих графів (наприклад, великих ігрових карт) алгоритм може працювати досить повільно. Алгоритм Дейкстри знаходить найкоротші шляхи до всіх вершин графа, навіть якщо нас цікавить лише шлях до однієї конкретної вершини.

Алгоритм A* (A* Search) є одним з найпопулярніших та ефективних алгоритмів пошуку шляху в іграх та інших задачах, де необхідно знайти оптимальний маршрут між двома точками. Він поєднує в собі переваги алгоритму

Дейкстри (гарантія оптимальності) та жадібного алгоритму «найкращого першого пошуку» (швидкість), використовуючи евристичну функцію для оцінки вартості шляху. На кожному кроці алгоритм вибирає з відкритого списку вершину з найменшою оцінкою вартості шляху ($f(n) = g(n) + h(n)$), де: $g(n)$ – вартість шляху від початкової точки до вершини n; $h(n)$ – евристична оцінка вартості шляху від вершини n до кінцевої точки. Вибрана вершина переміщується до закритого списку, а її сусіди додаються до відкритого списку (якщо вони ще не були розглянуті). При цьому оновлюються значення $g(n)$ та $f(n)$ для сусідів. Цей процес повторюється, поки не буде знайдена кінцева вершина або відкритий список не стане порожнім. Евристична функція $h(n)$ є ключовим елементом алгоритму A*. Вона повинна оцінювати вартість шляху від вершини n до кінцевої точки, але не переоцінювати її. Чим точніша евристична функція, тим швидше працює алгоритм A*. У покрокових стратегіях евристична функція може враховувати різні фактори, такі як відстань між вершинами (наприклад, манхеттенська відстань або евклідова відстань), тип місцевості, наявність перешкод тощо.

Алгоритм пошуку в ширину (Breadth–First Search, BFS) є одним з базових алгоритмів обходу графів та дерев. Він досліджує всі вершини графа рівень за рівнем, починаючи з початкової вершини. На кожному рівні алгоритм відвідує всі сусідні вершини поточної вершини, перш ніж перейти до наступного рівня. Основна ідея алгоритму полягає у тому, що початкова вершина додається у чергу. Основаною перевага – Алгоритм BFS має просту та інтуїтивно зрозумілу логіку, що робить його легким для розуміння та реалізації, а також те, що якщо всі ребра графа мають однакову вагу, BFS гарантовано знаходить найкоротший шлях від початкової до кінцевої вершини.

Алгоритм пошуку в глибину (Depth–First Search, DFS) є ще одним фундаментальним алгоритмом обходу графів та дерев. Він відрізняється від BFS тим, що досліджує граф «в глибину», тобто намагається пройти якомога далі вздовж обраного шляху, перш ніж повернутися та досліджувати інші гілки. Початкова вершина додається в стек. Поки стек не порожній – витягується верхня

вершина зі стека; якщо ця вершина є кінцевою, пошук завершується, і знайдений шлях повертається; якщо ця вершина не є кінцевою, всі її сусіди, які ще не були відвідані, додаються в стек.

Вибір конкретного алгоритму пошуку шляху залежить від розміру та складності ігрового поля, вимог до швидкості пошуку та інших факторів. Наприклад, для невеликих карт з простим ландшафтом може бути достатньо BFS, тоді як для великих карт зі складним ландшафтом краще використати A*.

Враховуючи, що покрокові стратегії зазвичай мають досить складні ігрові поля з різними типами ландшафту та перешкодами, алгоритм A* буде найбільш підходящим вибором для реалізації пошуку шляху в даній грі. Хоча він може бути складнішим у реалізації та вимагати більше пам'яті, ніж BFS, його здатність знаходити оптимальний шлях швидше за рахунок використання евристичної функції є важливою перевагою. Це особливо актуально для покрокових стратегій, де час пошуку шляху може впливати на загальну швидкість гри та досвід гравця.

2.3 Середовище програмування

У якості середовища програмування було обрано Rider.

Rider – це потужне інтегроване середовище розробки (IDE) від JetBrains, створене спеціально для роботи з мовами C# та .NET. Воно пропонує широкий спектр інструментів та функцій, які роблять розробку більш продуктивною та приємною.

Rider відомий своєю швидкістю та ефективністю роботи, навіть з великими проектами. Він швидко індексує код, забезпечуючи миттєвий доступ до навігації, рефакторингу та інших функцій.

Редактор коду в Rider пропонує автодоповнення, аналіз коду в реальному часі, підказки щодо помилок та швидкі виправлення. Він також підтримує різні мови програмування, включаючи C#, VB.NET, F#, JavaScript, TypeScript, HTML, CSS та інші.

Rider має широкий набір інструментів рефакторингу, які допомагають покращити структуру та якість коду без зміни його поведінки.

Відладчик Rider дозволяє легко знаходити та виправляти помилки в коді. Він підтримує віддалене відлагодження, покрокове виконання коду, перегляд значень змінних та багато іншого.

Rider має вбудовану підтримку Git, Mercurial, Subversion та інших популярних систем контролю версій. Він дозволяє легко керувати комітами, гілками, злиттями та іншими операціями VCS.

Rider має вбудовану підтримку популярних фреймворків юніт–тестування, таких як NUnit, xUnit.net та MSTest. Він дозволяє легко запускати тести, переглядати результати та аналізувати покриття коду тестами.

Rider має велику бібліотеку плагінів, які дозволяють розширити його функціональність та налаштувати його під свої потреби.

Rider доступний для Windows, macOS та Linux, що дозволяє використовувати його на різних операційних системах.

Rider – відмінний вибір для розробників C# та .NET, які шукають потужний, швидкий та зручний інструмент для розробки. Він підійде як для початківців, так і для досвідчених професіоналів.

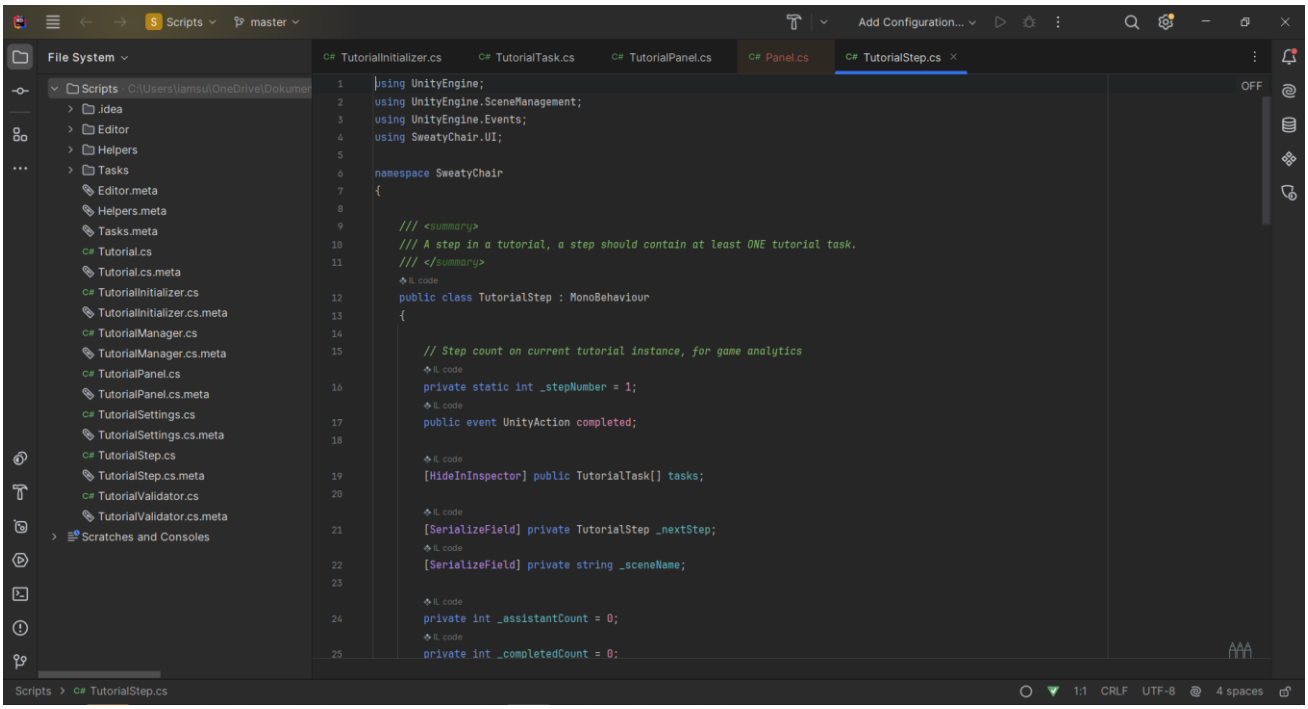


Рисунок 2.5 – Інтерфейс IDE Rider

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Рух та вибір юнітів

В першу чергу нам необхідно, щоб ігрові юніти рухались по ігровому полю. Для цього ми реалізуємо клас MoveAction, який успадкує від BaseAction. Цей клас буде відповідати за логіку переміщення юнітів між клітинками сітки.

У класі MoveAction ми визначимо такі ключові методи та властивості:

- Perform(): Цей метод буде виконувати власне переміщення юніта. Він отримає цільову позицію на сітці, перевірить, чи є цей рух допустимим (наприклад, чи не зайнята ця клітинка іншим юнітом), і якщо так, то перемістить юніта на нову позицію.
- GetValidActionGridPositionList(): Цей метод буде повертати список всіх допустимих позицій на сітці, куди юніт може переміститися за один хід. Це дозволить нам відобразити гравцеві можливі варіанти переміщення.

Для реалізації логіки перевірки доступності руху ми будемо використовувати інформацію з класу LevelGrid. Цей клас зберігає дані про стан ігрового поля, включаючи позиції всіх юнітів та перешкоди. В першу чергу нам необхідно, щоб ігрові юніти рухались по ігровому полю. Для цього ми реалізуємо клас MoveAction, який успадкує від BaseAction.

Після успішного виконання дії MoveAction ми викличемо подію OnAnyActionPointsChanged, щоб оновити інтерфейс користувача та відобразити зміни в кількості очок дії юніта.

Таким чином, реалізація класу MoveAction забезпечить базову механіку переміщення юнітів у нашій грі. Це дозволить гравцеві маневрувати своїми юнітами по полю бою, що є важливим елементом тактичної глибини гри.

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

public class Unit : MonoBehaviour
{
    private const int ACTION_POINTS_MAX = 3;


    public static event EventHandler OnAnyActionPointsChanged;
    public static event EventHandler OnAnyUnitSpawned;
    public static event EventHandler OnAnyUnitDead;


    [SerializeField] private bool isEnemy;


    private GridPosition gridPosition;
    private HealthSystem healthSystem;
    private BaseAction[] baseActionArray;
    private int actionPoints = ACTION_POINTS_MAX;

    private void Awake()
    {
        healthSystem = GetComponent<HealthSystem>();
        baseActionArray = GetComponents<BaseAction>();
    }

    private void Start()
    {
        gridPosition                                          =
LevelGrid.Instance.GetGridPosition(transform.position);
        LevelGrid.Instance.AddUnitAtGridPosition(gridPosition,
this);

        TurnSystem.Instance.OnTurnChanged                    +=
TurnSystem_OnTurnChanged;

        healthSystem.OnDead += HealthSystem_OnDead;

        OnAnyUnitSpawned?.Invoke(this, EventArgs.Empty);
    }

    private void Update()
    {
        GridPosition             newGridPosition             =
LevelGrid.Instance.GetGridPosition(transform.position);
        if (newGridPosition != gridPosition)
        {
            // Unit changed Grid Position
            GridPosition oldGridPosition = gridPosition;
```

```
                gridPosition = newGridPosition;

                LevelGrid.Instance.UnitMovedGridPosition(this,
oldGridPosition, newGridPosition);
            }
        }

        public T GetAction<T>() where T : BaseAction
        {
            foreach (BaseAction baseAction in baseActionArray)
            {
                if (baseAction is T)
                {
                    return (T)baseAction;
                }
            }
            return null;
        }

        public GridPosition GetGridPosition()
        {
            return gridPosition;
        }

        public Vector3 GetWorldPosition()
        {
            return transform.position;
        }

        public BaseAction[] GetBaseActionArray()
        {
            return baseActionArray;
        }

        public   bool   TrySpendActionPointsToTakeAction(BaseAction
baseAction)
        {
            if (CanSpendActionPointsToTakeAction(baseAction))
            {

SpendActionPoints(baseAction.GetActionPointsCost());
                return true;
            } else
            {
                return false;
            }
        }

        public   bool   CanSpendActionPointsToTakeAction(BaseAction
baseAction)
        {
            if (actionPoints >= baseAction.GetActionPointsCost())
```

```
            {
                return true;
            } else
            {
                return false;
            }
        }

        private void SpendActionPoints(int amount)
        {
            actionPoints -= amount;

            OnAnyActionPointsChanged?.Invoke(this,
EventArgs.Empty);
        }

        public int GetActionPoints()
        {
            return actionPoints;
        }

        private   void   TurnSystem_OnTurnChanged(object   sender,
EventArgs e)
        {
            if ((IsEnemy() && !TurnSystem.Instance.IsPlayerTurn())
||
                (!IsEnemy() && TurnSystem.Instance.IsPlayerTurn()))
            {
                actionPoints = ACTION_POINTS_MAX;

                OnAnyActionPointsChanged?.Invoke(this,
EventArgs.Empty);
            }
        }

        public bool IsEnemy()
        {
            return isEnemy;
        }

        public void Damage(int damageAmount)
        {
            healthSystem.Damage(damageAmount);
        }

        private void HealthSystem_OnDead(object sender, EventArgs
e)
        {

LevelGrid.Instance.RemoveUnitAtGridPosition(gridPosition, this);

            Destroy(gameObject);
```

```
            OnAnyUnitDead?.Invoke(this, EventArgs.Empty);
        }

        public float GetHealthNormalized()
        {
            return healthSystem.GetHealthNormalized();
        }
    }
```

Цей C#-скрипт, названий Unit, визначає фундаментальну схему для юнітів у покроковій стратегічній грі, розробленій в Unity. Він інкапсулює основні атрибути та поведінку, що характеризують юнітів у такій грі, включаючи управління точками дії, взаємодію з покроковою системою та ігровою сіткою, виконання дій, управління здоров'ям та комунікацію за допомогою подій.

В основі можливостей юнітів лежить система очок дій. Кожен підрозділ має максимальну кількість очок дій, які визначають обсяг дій, які він може здійснити протягом одного ходу. Сценарій включає механізми, які перевіряють, чи може підрозділ дозволити собі певну дію, виходячи з того, скільки очок дії залишилося, і віднімають ці очки, якщо дія була виконана. Ця система гарантує, що підрозділи діють в рамках обмежень покрокової структури, сприяючи прийняттю стратегічних рішень.

Клас Юнітів тісно інтегрований з покроковою системою гри. Він стежить за змінами ходів і, коли настає його черга, поповнює свої очки дій до максимального значення. Цей механізм поповнення дозволяє юніту виконувати новий набір дій кожного ходу, підтримуючи динамічний характер гри.

Для навігації в ігровому світі юніти взаємодіють з системою сіток, представленою класом LevelGrid. Скрипт зберігає поточну позицію юніта на цій сітці і постійно відстежує будь-які зміни позиції. Якщо юніт рухається, скрипт оновлює його позицію на сітці та інформує LevelGrid про зміну, гарантуючи, що місцезнаходження юніта завжди точне.

Клас Unit також надає функціональність для виконання дій. Скрипт може отримувати певні типи дій, такі як рух або атака, з масиву доступних дій. Крім

того, він показує всі дії, які може виконувати юніт, що дозволяє іншим частинам гри запитувати цю інформацію для прийняття рішень або відображення.

Здоров'ям юніта керує компонент HealthSystem. Сценарій дозволяє юніту отримувати пошкодження, які відображаються на його стані здоров'я. Коли рівень здоров'я юніта знижується до нуля, спеціальний метод обробляє його загибель, видаляючи його з сітки та знищуючи пов'язаний з ним ігровий об'єкт. Крім того, запускається подія, яка сповіщає інші компоненти про смерть юніта, що полегшує будь-які необхідні оновлення ігрової логіки.

Для зв'язку з іншими частинами гри клас юнітів використовує події. Ці події виникають щоразу, коли змінюються очки дії юніта, народжується новий юніт або юніт знищується. Ця архітектура, керована подіями, сприяє вільному зв'язку між компонентами, підвищуючи зручність супроводу та гнучкість кодової бази.

Крім того, скрипт пропонує різні способи доступу до атрибутів юнітів, таких як їхня позиція в сітці, світова позиція, статус ворога та нормалізоване значення здоров'я. Така доступність полегшує взаємодію інших ігрових елементів з юнітами, сприяючи створенню цілісної та добре структурованої ігрової системи.

На завершення, клас Юнітів слугує для комплексного представлення окремих сутностей у покрокових стратегічних іграх. Його функції охоплюють управління точками дії, взаємодію в сітці, виконання дій, відстеження здоров'я та комунікацію через події. Цей всебічний дизайн закладає міцний фундамент для побудови різноманітних і захопливих ігрових сценаріїв у покроковому середовищі.

3.2 Grid система

Після того я ми реалізували базовий рух, нам потрібно реалізувати рух юнітів по клітинках. Створимо клас GridObject який буде фундаментальним елементом для роботи з об'єктами на ігровій сітці. Він відповідає за зберігання та

управління інформацією про кожну окрему клітинку сітки, включаючи її позицію, юнітів, що знаходяться на ній, та інтерактивні елементи.

Основна функціональність класу включає:

– Зберігання інформації про позицію: Кожен об'єкт GridObject зберігає свою позицію на сітці у вигляді координат (рядка та стовпця). Це дозволяє легко визначити місцезнаходження об'єкта на ігровому полі та взаємодіяти з ним відповідно.

– Управління юнітами: GridObject підтримує список юнітів, що знаходяться на клітинці. Це дозволяє відстежувати, які юніти займають конкретну позицію, а також виконувати операції додавання, видалення та отримання списку юнітів.

– Інтерактивність: Клас надає можливість пов'язувати інтерактивні об'єкти з клітинками сітки. Це дозволяє створювати різноманітні інтерактивні елементи на ігровому полі, такі як кнопки, перемикачі, об'єкти, з якими можна взаємодіяти тощо.

– Взаємодія з GridSystem: GridObject тісно взаємодіє з класом GridSystem, який відповідає за створення та управління всією сіткою. Це дозволяє отримувати доступ до властивостей та методів системи сітки, таких як розміри сітки, розмір комірки, перетворення координат тощо.

Використання класу GridObject значно спрощує розробку ігрової логіки, пов'язаної з сіткою. Він дозволяє абстрагуватися від деталей реалізації зберігання даних про клітинки та зосередитися на ігровій механіці. Крім того, GridObject покращує структуру коду, робить його більш зрозумілим та полегшує подальший розвиток та підтримку гри.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class GridObject
```

```
    {

        private GridSystem<GridObject> gridSystem;
        private GridPosition gridPosition;
        private List<Unit> unitList;
        private IInteractable interactable;
        public    GridObject(GridSystem<GridObject>    gridSystem,
GridPosition gridPosition)
        {
            this.gridSystem = gridSystem;
            this.gridPosition = gridPosition;
            unitList = new List<Unit>();

        }

        public override string ToString()

        {

            string unitString = "";

            foreach (Unit unit in unitList)

            {

                unitString += unit + "\n";

            }

            return gridPosition.ToString() + "\n" + unitString;

        }

        public void AddUnit(Unit unit)

        {
```

```
        unitList.Add(unit);
    }
    public void RemoveUnit(Unit unit)
    {
        unitList.Remove(unit);
    }
    public List<Unit> GetUnitList()
    {
        return unitList;
    }
    public bool HasAnyUnit()
    {
        return unitList.Count > 0;
    }
    public Unit GetUnit()
    {
        if (HasAnyUnit())
        {
```

```
            return unitList[0];
        } else
        {
            return null;
        }
    }
    public IInteractable GetInteractable()
    {
        return interactable;
    }
    public void SetInteractable(IInteractable interactable)
    {
        this.interactable = interactable;
    }
    public void ClearInteractable()
    {
        this.interactable = null;
    }
}
```

3.3 Дії та UI

Далі приступимо до реалізації класу TurnSystemUI, відповідального за візуальне представлення та управління елементами інтерфейсу користувача, пов'язаними з системою черговості.

Функціональність класу TurnSystemUI:

– Відображення інформації про поточний хід:

  – turnNumberText: Відображає номер поточного ходу.

  – enemyTurnVisualGameObject: Візуально вказує, чи є поточний хід ходом гравця чи ворога.

– Управління кнопкою завершення ходу:

  – endTurnBtn: Кнопка для завершення ходу гравця, яка відображається лише під час його ходу.

– Реакція на зміну ходу:

  – TurnSystem_OnTurnChanged: Оновлює всі елементи інтерфейсу при зміні ходу.

TurnSystemUI тісно взаємодіє з класом TurnSystem, який відповідає за логіку черговості. Він отримує від TurnSystem інформацію про поточний хід та стан гри, а також може викликати методи TurnSystem для зміни стану гри.

Реалізація класу TurnSystemUI є важливим кроком у створенні покрокової стратегії. Він забезпечує зручний та інформативний інтерфейс користувача, дозволяючи гравцеві бачити, чий зараз хід, який номер ходу, та завершувати свій хід. Це сприяє кращому розумінню ігрової механіки та підвищує задоволення від гри.

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

```
using UnityEngine;
using UnityEngine.UI;
using TMPro;
public class TurnSystemUI : MonoBehaviour
{
    [SerializeField] private Button endTurnBtn;
    [SerializeField] private TextMeshProUGUI turnNumberText;
    [SerializeField] private GameObject
enemyTurnVisualGameObject;

    private void Start()
    {
        endTurnBtn.onClick.AddListener(() =>
        {
            TurnSystem.Instance.NextTurn();
        });

        TurnSystem.Instance.OnTurnChanged +=
TurnSystem_OnTurnChanged;

        UpdateTurnText();
        UpdateEnemyTurnVisual();
        UpdateEndTurnButtonVisibility();
    }

    private void TurnSystem_OnTurnChanged(object sender,
EventArgs e)
    {
        UpdateTurnText();
        UpdateEnemyTurnVisual();
        UpdateEndTurnButtonVisibility();
    }

    private void UpdateTurnText()
    {
        turnNumberText.text = "TURN " +
TurnSystem.Instance.GetTurnNumber();
    }
```

```
        private void UpdateEnemyTurnVisual()
        {

enemyTurnVisualGameObject.SetActive(!TurnSystem.Instance.IsPlayerTu
rn());
        }

        private void UpdateEndTurnButtonVisibility()
        {

endTurnBtn.gameObject.SetActive(TurnSystem.Instance.IsPlayerTurn())
;
        }
    }
```

В основі процесу прийняття рішень лежить метод GetBestEnemyAIAction, який, ймовірно, реалізований унікально для кожного типу дій (наприклад, переміщення, атака, захист). Цей метод, ймовірно, включає специфічну для гри логіку для оцінки тактичної переваги кожної потенційної дії. Наприклад, дія "переміщення" може надавати пріоритет позиціям, які пропонують краще прикриття або можливості для обходу з флангу, тоді як дія "атака" може бути спрямована на найбільш вразливий підрозділ противника.

Якщо для жодного ворога не знайдено підходящої дії, ШІ граціозно завершує свій хід, повертаючи контроль гравцеві. Така покрокова структура забезпечує чітке розмежування дій і підтримує стратегічний потік гри.

Окрім керування ходом ШІ, код також містить обробник подій (TurnSystem_OnTurnChanged), який гарантує, що ШІ негайно реактивується, коли його хід починається знову. Такий плавний перехід між ходами сприяє плавному та цікавому ігровому процесу.

Загалом, цей фрагмент коду формує основу для складної системи штучного інтелекту в покроковій стратегічній грі. Модульна конструкція дозволяє легко

налаштовувати і розширювати систему, оскільки конкретна логіка дій і критерії оцінки можуть бути адаптовані до унікальної механіки і цілей гри.

3.4 Вороги та битва

Клас EnemyAI відповідає за управління поведінкою ворогів у покроковій стратегії, використовуючи кінцевий автомат для визначення поточного стану. Він обирає та виконує дії ворожих юнітів на основі їх оцінки, переходячи між станами очікування ходу, виконання ходу та зайнятості. ШІ взаємодіє з класами TurnSystem, UnitManager, Unit та BaseAction для отримання інформації про стан гри, вибору найкращих дій та їх виконання. Завдяки цьому класу, вороги можуть аналізувати ситуацію на полі бою, приймати рішення та виконувати дії, що відповідають їхнім цілям, роблячи гру більш цікавою та складною для гравця.

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


public class EnemyAI : MonoBehaviour


{


    private enum State


    {


        WaitingForEnemyTurn,


        TakingTurn,


        Busy,


    }
```

```
        private State state;

        private float timer;

        private void Awake()

        {

            state = State.WaitingForEnemyTurn;

        }



        private void Start()

        {

            TurnSystem.Instance.OnTurnChanged                    +=
TurnSystem_OnTurnChanged;

        }

        private void Update()

        {

            if (TurnSystem.Instance.IsPlayerTurn())

            {

                return;

            }

            switch (state)
```

```
            {
                case State.WaitingForEnemyTurn:
                    break;
                case State.TakingTurn:
                    timer -= Time.deltaTime;
                    if (timer <= 0f)
                    {
                        if
(TryTakeEnemyAIAction(SetStateTakingTurn))
                        {
                            state = State.Busy;
                        } else
                        {
                            // No more enemies have actions they can
take, end enemy turn
                            TurnSystem.Instance.NextTurn();
                        }
                    }
                    break;
                case State.Busy:
```

```
                break;

            }

        }

        private void SetStateTakingTurn()

        {

            timer = 0.5f;

            state = State.TakingTurn;

        }

        private void TurnSystem_OnTurnChanged(object sender,
EventArgs e)

        {

            if (!TurnSystem.Instance.IsPlayerTurn())

            {

                state = State.TakingTurn;

                timer = 2f;

            }

        }

        private bool TryTakeEnemyAIAction(Action
onEnemyAIActionComplete)

        {
```

```
            foreach (Unit enemyUnit in
UnitManager.Instance.GetEnemyUnitList())

            {

                if (TryTakeEnemyAIAction(enemyUnit,
onEnemyAIActionComplete))

                {

                    return true;

                }

            }

            return false;

        }

        private bool TryTakeEnemyAIAction(Unit enemyUnit, Action
onEnemyAIActionComplete)

        {

            EnemyAIAction bestEnemyAIAction = null;

            BaseAction bestBaseAction = null;



            foreach (BaseAction baseAction in
enemyUnit.GetBaseActionArray())

            {

                if
(!enemyUnit.CanSpendActionPointsToTakeAction(baseAction))
```

```
                {

                    // Enemy cannot afford this action

                    continue;

                }

                if (bestEnemyAIAction == null)

                {

            bestEnemyAIAction = baseAction.GetBestEnemyAIAction();

                    bestBaseAction = baseAction;

                }

                else

                {

                    EnemyAIAction testEnemyAIAction = baseAction.GetBestEnemyAIAction();

                    if (testEnemyAIAction != null && testEnemyAIAction.actionValue > bestEnemyAIAction.actionValue)

                    {

                        bestEnemyAIAction = testEnemyAIAction;

                        bestBaseAction = baseAction;

                    }

                }
```

```
            }

   if          (bestEnemyAIAction          !=          null          &&
enemyUnit.TrySpendActionPointsToTakeAction(bestBaseAction))

            {


bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition,
onEnemyAIActionComplete);

                return true;

            }

            else
            {
                return false;
            }
        }
```

Цей код встановлює основні принципи управління поведінкою ворожого ШІ в покроковій стратегічній грі. Дії ШІ регулюються окремими станами: WaitingForEnemyTurn (бездіяльність під час ходу гравця), TakingTurn (активна оцінка та виконання дій) та Busy (під час виконання дій).

Коли починається хід ШІ, він переходить у стан TakingTurn, де робить коротку паузу перед тим, як пройтись по кожній ворожій одиниці, що знаходиться під його контролем. Для кожної одиниці викликається метод TryTakeEnemyAIAction для оцінки доступних дій. Цей метод визначає оптимальну дію, враховуючи такі фактори, як залишок очок дій юніта і "цінність" кожної потенційної дії. Дія з найвищою цінністю потім виконується, якщо вона є доступною.

В основі процесу прийняття рішень лежить метод GetBestEnemyAIAction, який, ймовірно, реалізований унікально для кожного типу дій (наприклад, переміщення, атака, захист). Цей метод, ймовірно, включає специфічну для гри

логіку для оцінки тактичної переваги кожної потенційної дії. Наприклад, дія "переміщення" може надавати пріоритет позиціям, які пропонують краще прикриття або можливості для обходу з флангу, тоді як дія "атака" може бути спрямована на найбільш вразливий підрозділ противника.

Якщо для жодного ворога не знайдено підходящої дії, ШІ граціозно завершує свій хід, повертаючи контроль гравцеві. Така покрокова структура забезпечує чітке розмежування дій і підтримує стратегічний потік гри.

Окрім керування ходом ШІ, код також містить обробник подій (TurnSystem_OnTurnChanged), який гарантує, що ШІ негайно реактивується, коли його хід починається знову. Такий плавний перехід між ходами сприяє плавному та цікавому ігровому процесу.

Загалом, цей фрагмент коду формує основу для складної системи штучного інтелекту в покроковій стратегічній грі. Модульна конструкція дозволяє легко налаштовувати і розширювати систему, оскільки конкретна логіка дій і критерії оцінки можуть бути адаптовані до унікальної механіки і цілей гри.

3.5 Юніт–тестування

Юніт–тестування в Unity є особливо важливим через складність ігрових проектів. Воно допомагає забезпечити коректну роботу ігрової логіки, поведінки персонажів, взаємодії з об'єктами, фізичних симуляцій та інших компонентів гри. Враховуючи, що ігри часто мають велику кількість взаємопов'язаних систем, юніт–тести дозволяють ізолювати та перевірити кожну з них окремо, що спрощує пошук та виправлення помилок. Крім того, Unity має власний інструментарій для юніт–тестування, що робить його інтеграцію в процес розробки ще більш зручним та ефективним.

Юніт–тест призначений для тестування окремого «юніта» коду. Склад «юніта» може варіюватися, але важливо пам'ятати, що юніт–тестування має тестувати рівно один «елемент» за раз.

Юніт–тести необхідно створювати для перевірки того, що невеликий логічний фрагмент коду в конкретному сценарії виконується саме так, як ви очікуєте. Розглянемо такий приклад.

Було реалізовано метод, що дозволяє користувачеві вводити ім'я. Метод написаний так, що в імені не допускаються цифри, а саме ім'я може складатися тільки з десяти або менше символів. Наш метод перехоплює натискання кожної клавіші й додає відповідний символ у поле name:

```
public string name = ""
public void UpdateNameWithCharacter(char: character)
{
    // 1
    if (!Char.IsLetter(char))
    {
        return;
    }

    // 2
    if (name.Length >= 10)
    {
        return;
    }

    // 3
    name += character;
}
```

Рисунок 3.1 – Код класу UpdateNameWithCharacter

Розглянемо, що відбувається у данному фрагменті коду:

– Якщо символ не є літерою, то код виконує попередній вихід із функції і не додає символ у рядок.

– Якщо довжина імені становить десять або більше символів, то код не дозволяє користувачеві додати ще один символ.

– Якщо ці дві перевірки пройдено, то код додає в кінець імені символ.

Цей юніт можна протестувати, тому що він являє собою «модуль» виконуваної роботи. Юніт–тести примусово виконують логіку методу.

3.6 Робота з юніт–тестами в середовищі Unity

Щоб писати тести, спочатку відкриємо Unity Test Runner. Test Runner дає змогу виконувати тести та перевіряти, чи проходять вони успішно. Щоб відкрити Unity Test Runner, виберіть Window ▸ General ▸ Test Runner.

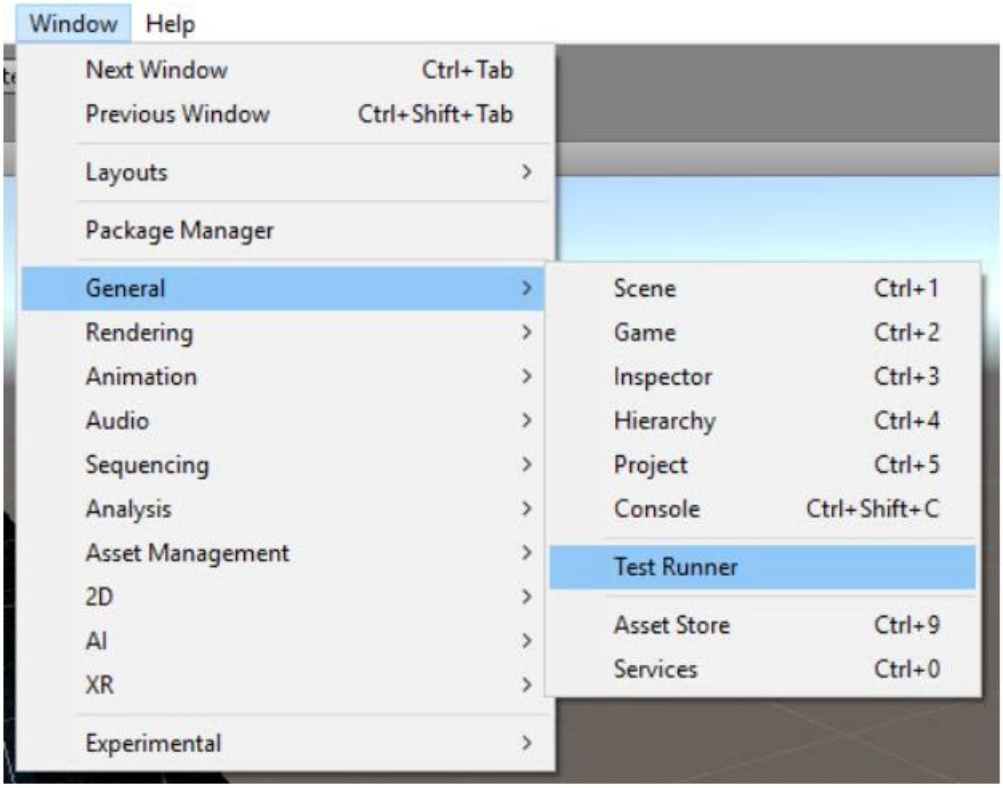


Рисунок 3.2 – Меню вибору Test runner

Після того, як у новому вікні відкриється Test Runner, можна буде спростити собі життя, натиснувши на вікно Test Runner і перетягнувши його на місце поруч із вікном Scene:

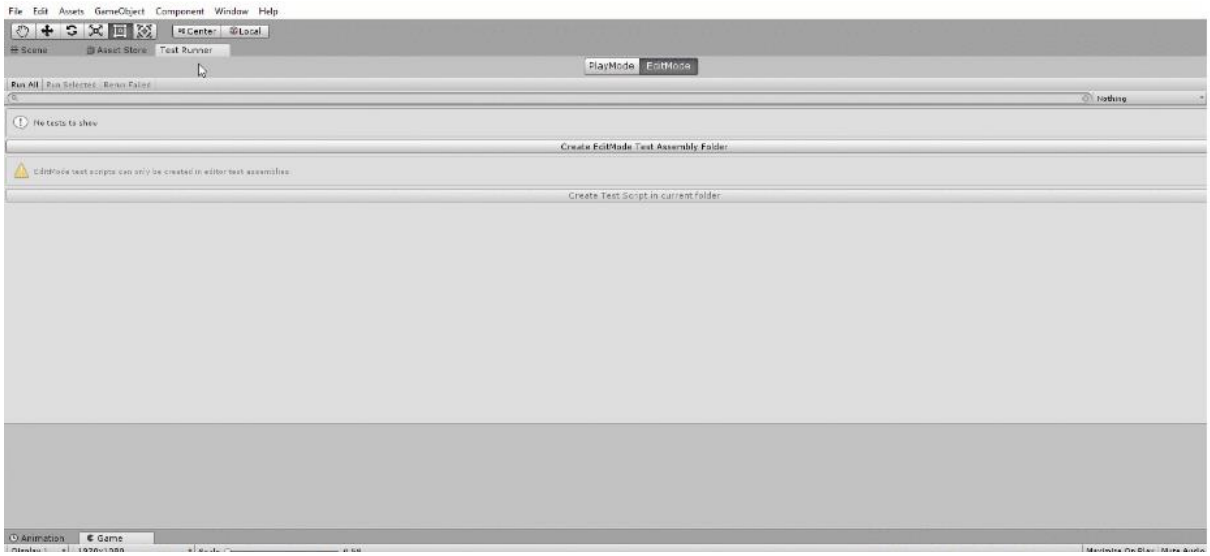


Рисунок 3.3 – Інтерфейс вікна TestRunner

Далі підготуємо NUnit та папки тестів. Test Runner – це функція юніт–тестування, що надається Unity, але вона використовує фреймворк NUnit.

Для запуску тестів нам спочатку потрібно створити папку тестів, у якій зберігатимуться класи тестів.

У вікні Project вибираємо папку RW. Дивимося на вікно Test Runner і переконуємоя, що обрано PlayMode.

Натискаємо кнопку з назвою Create PlayMode Test Assembly Folder. Можна побачите, що в папці RW з'явиться нова папка. Нас влаштує стандартна назва Tests, тому можна просто натиснути Enter.

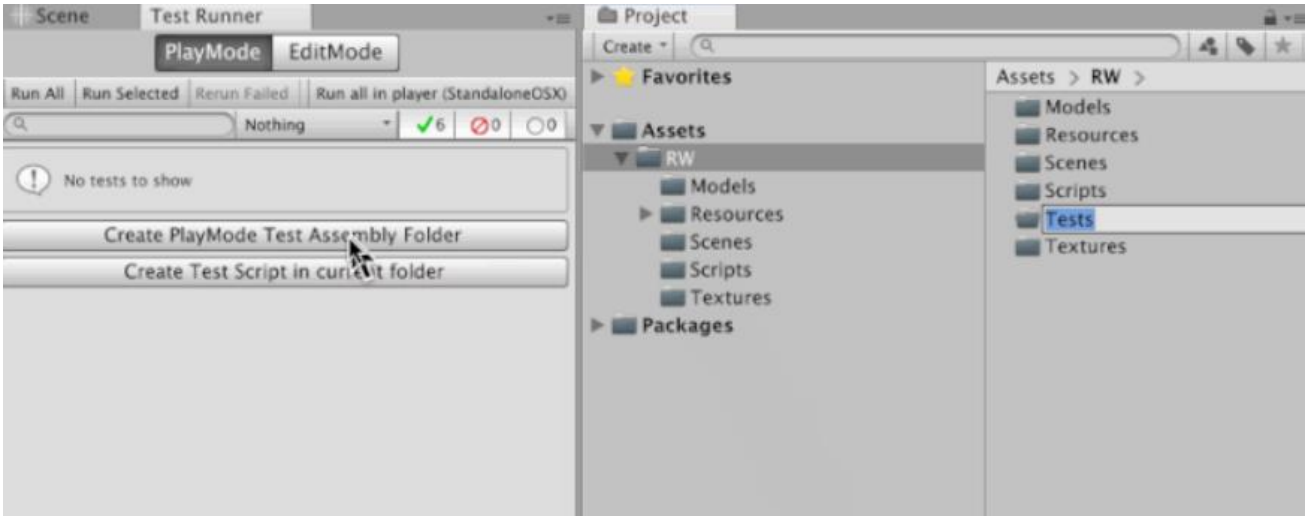


Рисунок 3.4 – Створення папки для тестів

Вкладка PlayMode використовується для тестів, що виконуються в режимі Play (коли гра виконується в реальному часі). Тести вкладки EditMode виконуються поза режимом Play, що зручно для тестування таких речей, як користувацькі behaviors в Inspector.

Як ми дізналися вище, юніт–тест – це функція, що тестує поведінку невеликого конкретного фрагмента коду. Оскільки юніт–тест є методом, для його запуску він має знаходитися у файлі класу.

Test Runner обходить усі файли класів тестів і виконує юніт–тести з них. Файл класу, що містить юніт–тести, називається комплектом тестів (test suite).

У комплекті тестів ми логічно поділяємо наші тести. Ми повинні розділяти код тестів на окремі логічні комплекти (наприклад, комплект тестів для фізики та окремий комплект для бою). Нам знадобиться лише один комплект тестів, і зараз перейдемо до його створення.

Вибераємо папку Tests і у вікні Test Runner натискаємо на кнопку Create Test Script in current folder. Назвемо новий файл TestSuite.

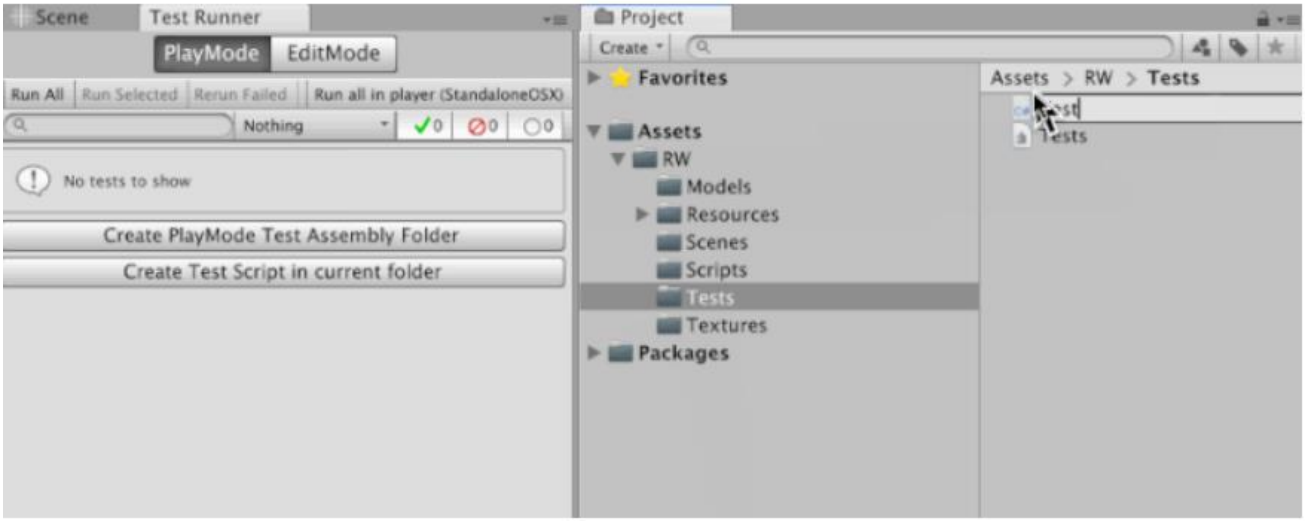


Рисунок 3.5 – Створення скрипта тесту

Крім нового файлу C# рушій Unity також створює ще один файл під назвою Tests.asmdef. Це файл визначення збірки (assembly definition file), який використовується для того, щоб показати Unity, де знаходяться залежності файлу

тесту. Це потрібно, тому що код готового застосунку міститься окремо від тестового коду.

Щоб код тестів мав доступ до класів гри, ми створимо збірку коду класів і задамо посилання в збірці Tests. Натисніть на папку Scripts, щоб вибрати її. Натисніть правою клавішею на цю папку і виберіть Create ▸ Assembly Definition.

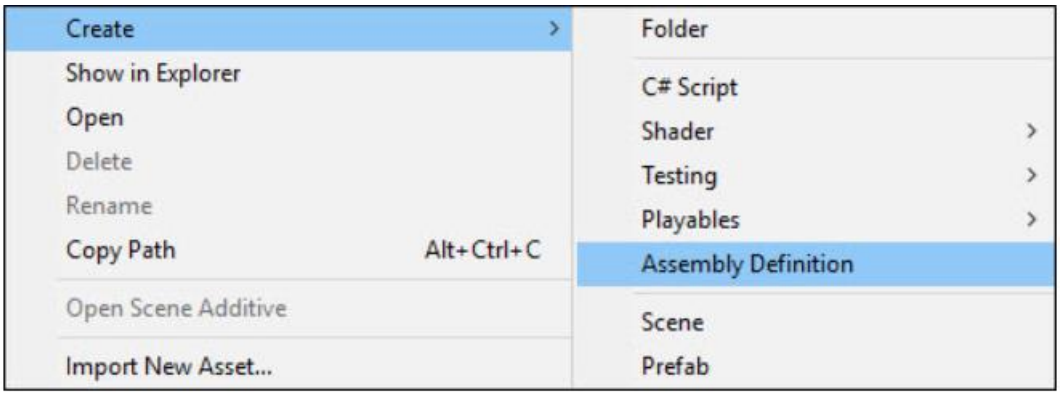


Рисунок 3.6 – Створення Assemby Definition 1

Назвемо файл GameAssembly

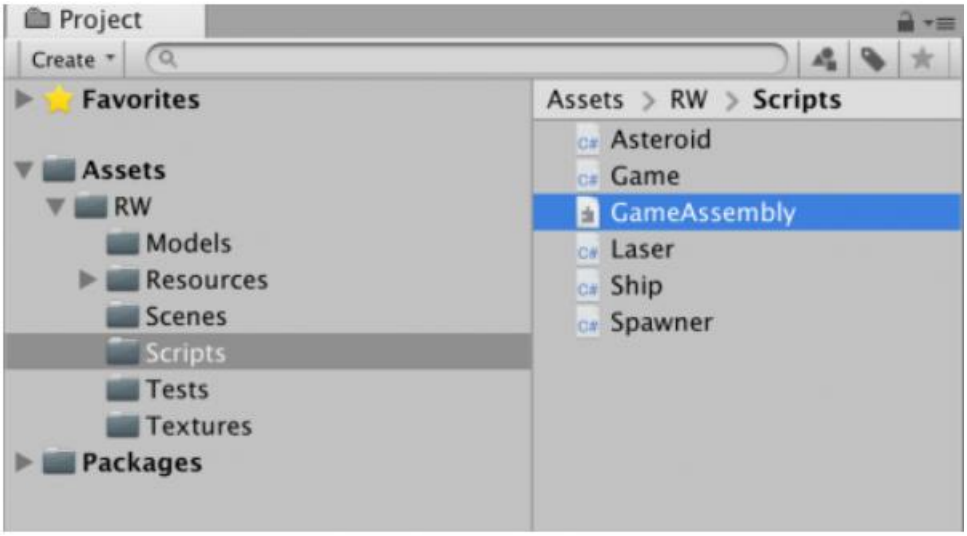


Рисунок 3.7 – Створення Assemby Definition 2

Натискаємо на папку Tests, а потім на файл визначення збірки Tests. В Inspector натискаємо на кнопку плюс під заголовком Assembly Definition References.

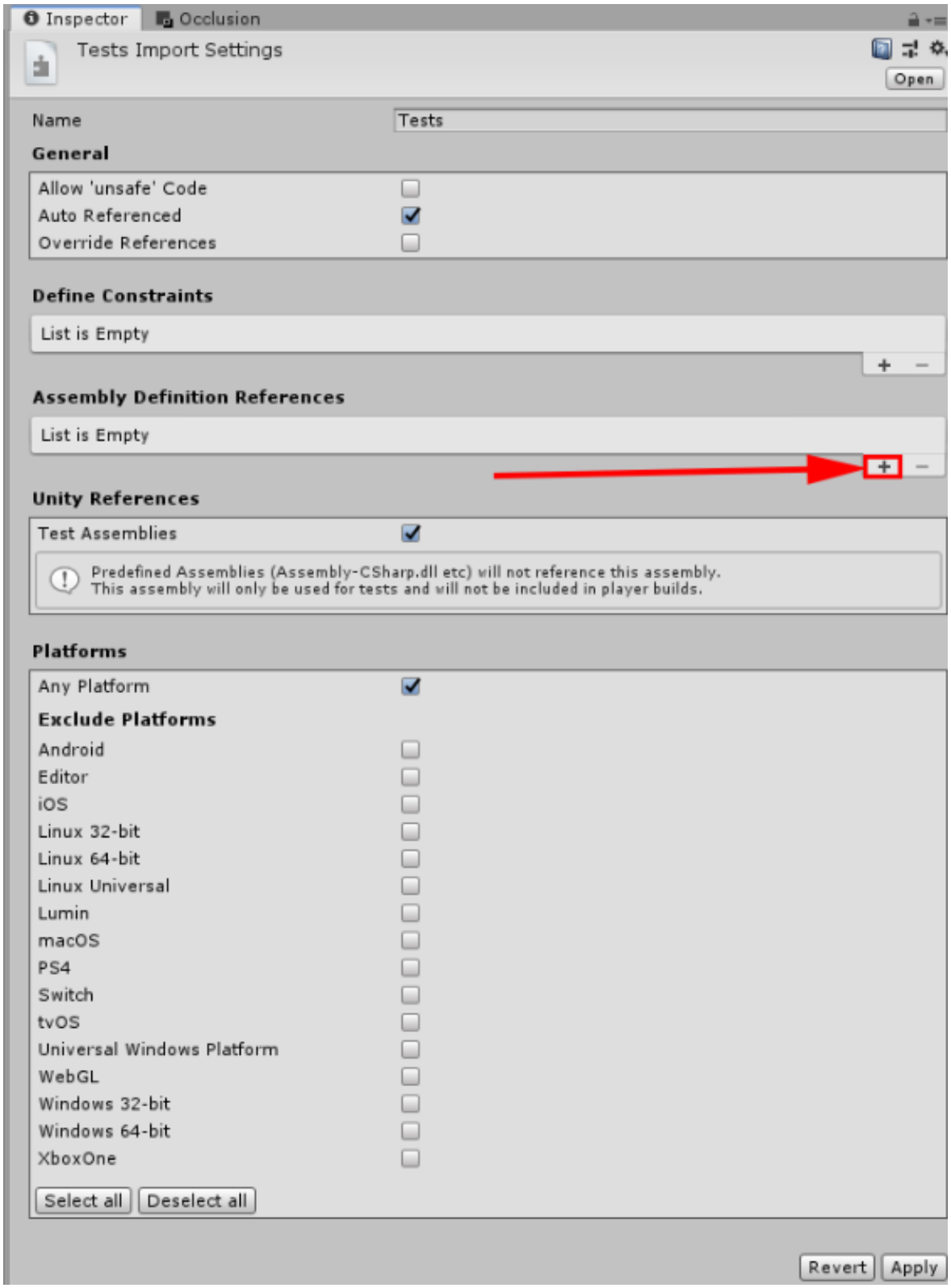


Рисунок 3.8 – Додавання reference до Assemby Definition 1

Ми побачимо поле Missing Reference. Натиснемо на крапку поруч із цим полем, щоб відкрити вікно вибору. Потрібно вибрати файл GameAssembly.

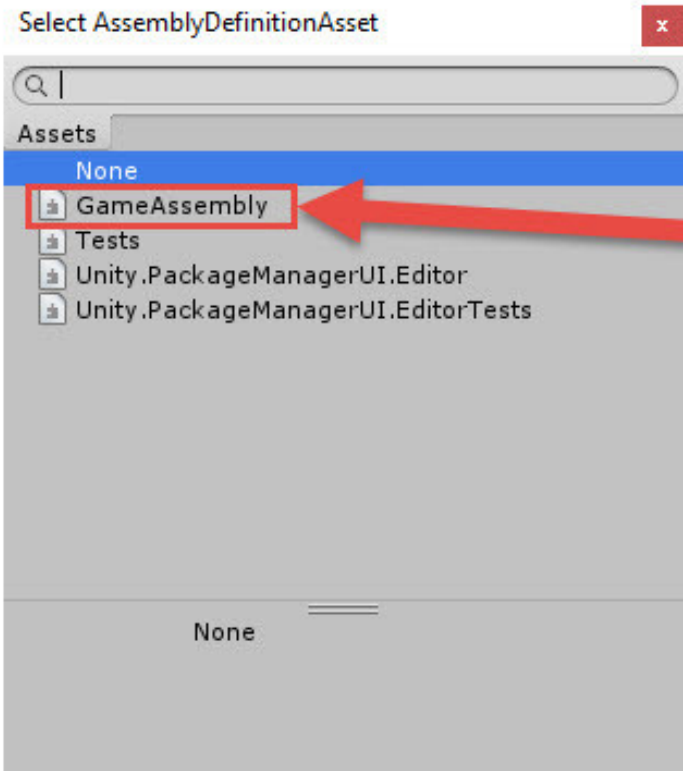


Рисунок 3.9 – Додавання reference до Assemby Definition 2

Далі ми побачимо файл збірки GameAssembly у розділі посилань. Натиснемо на кнопку Apply, щоб зберегти ці зміни.

Рисунок 3.10 – Додавання reference до Assemby Definition 3

3.7 Написання юніт–тесту

Створимо скрипт UnitMovementTests, та відкрити його в редакторі коду. Замінимо весь код на такий:

```
using NUnit.Framework;
using UnityEngine;

public class UnitMovementTests
{
    private GameObject unitGameObject;
    private Unit unit;

    [SetUp]
    public void Setup()
    {
        // Створюємо ігровий об'єкт юніта та отримуємо його компонент Unit
        unitGameObject = new GameObject("TestUnit");
        unit = unitGameObject.AddComponent<Unit>();

        // Ініціалізуємо інші необхідні компоненти (наприклад, GridPosition)
        // ...
    }

    [TearDown]
    public void Teardown()
    {
        // Видаляємо ігровий об'єкт після кожного тесту
        Object.Destroy(unitGameObject);
    }

    [Test]
    public void TestMoveUnitToValidPosition()
    {
        // Визначаємо початкову та цільову позиції на полі
        GridPosition startPosition = new GridPosition(0, 0);
        GridPosition targetPosition = new GridPosition(2, 1);

        // Встановлюємо початкову позицію юніта
        unit.SetGridPosition(startPosition);

        // Викликаємо метод руху юніта
        bool result = unit.Move(targetPosition);

        // Перевіряємо, чи рух був успішним
        Assert.IsTrue(result, "Рух до допустимої позиції повинен бути успішним.");
```

```
            // Перевіряємо, чи юніт знаходиться в цільовій позиції
            Assert.AreEqual(targetPosition, unit.GetGridPosition(),
"Юніт повинен бути в цільовій позиції.");
        }

        [Test]
        public void TestMoveUnitToOccupiedPosition()
        {
            // ... (аналогічно до попереднього тесту, але цільова
позиція вже зайнята)

            // Перевіряємо, чи рух був неуспішним
            Assert.IsFalse(result, "Рух до зайнятої позиції повинен
бути неуспішним.");

            // Перевіряємо, чи юніт залишився на початковій позиції
            Assert.AreEqual(startPosition,  unit.GetGridPosition(),
"Юніт повинен залишитися на початковій позиції.");
        }

        // Інші тести для перевірки руху за межі поля, руху з
перешкодами тощо
        // ...
    }
```

Розглянемо детальніше кожну частину коду:

– SetUp: Створює ігровий об'єкт юніта та ініціалізує його перед кожним тестом.

– TearDown: Видаляє ігровий об'єкт після кожного тесту, щоб уникнути впливу на інші тести.

– TestMoveUnitToValidPosition: Перевіряє, чи юніт може успішно переміститися на вільну позицію на полі.

– TestMoveUnitToOccupiedPosition: Перевіряє, чи юніт не може переміститися на позицію, яка вже зайнята іншим юнітом.

Ми написали перший юніт–тест, тепер слід дізнатися, чи він працює. Це можна зробити зрозуміло, за допомогою Test Runner. У вікні Test Runner розкриваємо всі рядки зі стрілками. Ми побачимо тест UnitMovementTests у списку з сірими кружками.

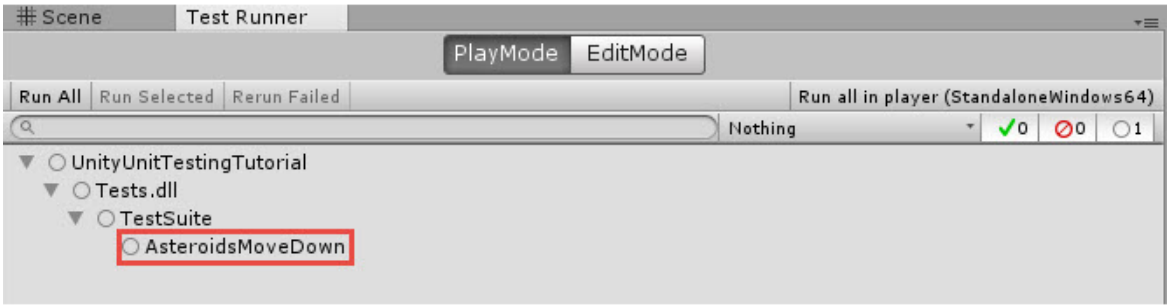


Рисунок 3.11 – Запуск тесту 1

Сірий кружок означає, що тест поки що не виконувався. Якщо тест був запущений і пройдений, то поруч показується зелена стрілка. Якщо тест завершився з помилкою, то поруч із ним буде відображено червоний X. Запустимо тест, натиснувши на кнопку RunAll.

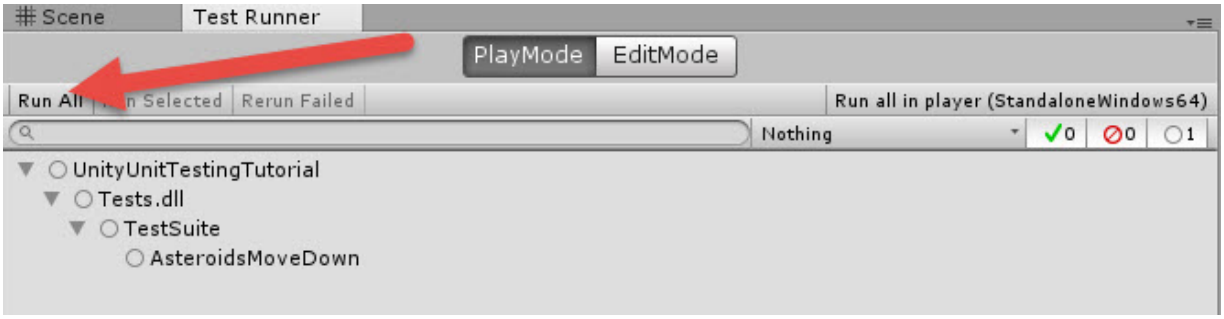


Рисунок 3.12 – Запуск тесту 2

При цьому створиться тимчасова сцена і буде запущено тест. Після завершення ви маєте побачити, що тест пройдено.

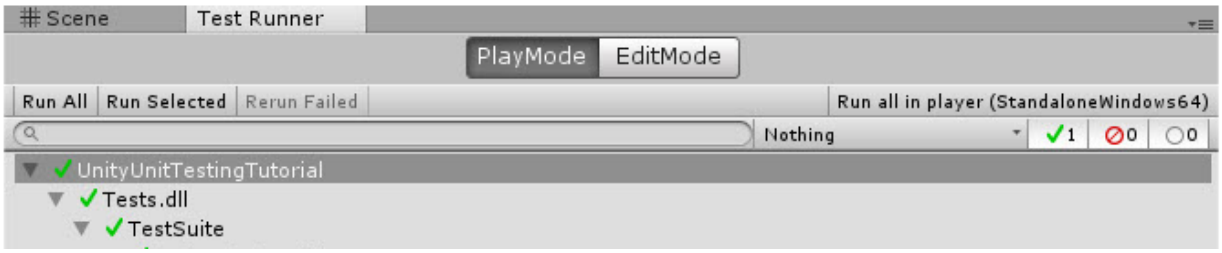


Рисунок 3.12 – Запуск тесту 3

ВИСНОВКИ

У рамках кваліфікаційної роботи було досягнуто поставленої мети – розроблено покрокову гру-стратегію з використанням алгоритму Мінімакс для прийняття рішень комп'ютерним опонентом.

В ході дослідження предметної області було проведено аналіз існуючих покрокових ігор-стратегій, виявлено їх основні особливості та вимоги до програмного забезпечення. Було досліджено алгоритм Мінімакс, його переваги та недоліки, а також способи його оптимізації та адаптації до конкретних ігрових ситуацій.

На етапі проектування було розроблено детальну архітектуру програмного забезпечення, визначено основні компоненти та їх взаємодію. Було обрано ігровий рушій Unity та мову програмування C# як найбільш підходящі інструменти для реалізації проєкту.

В результаті програмної реалізації було створено покрокову гру–стратегію, в якій комп'ютерний опонент використовує алгоритм Мінімакс для прийняття рішень. Гра має зручний інтерфейс користувача, дозволяє гравцеві вибирати рівень складності та налаштовувати інші параметри гри.

Проведене тестування підтвердило працездатність та ефективність розробленого програмного забезпечення. Алгоритм Мінімакс забезпечує високий рівень інтелекту комп'ютерного опонента, що робить гру цікавою та складною для гравця.

В цілому, дипломний проєкт успішно виконано. Розроблена покрокова гра–стратегія з використанням алгоритму Мінімакс може бути використана як для розваги, так і для навчання основам стратегічного мислення.

Подальший розвиток проєкту може включати в себе:

- Додавання нових типів юнітів та будівель.
- Розширення можливостей штучного інтелекту
- Вдосконалення інтерфейсу користувача.

– Оптимізацію продуктивності гри.

– Портування гри на інші платформи (мобільні пристрої, веб).

Результати кваліфікаційної роботи можуть бути використані для подальших досліджень в області розробки ігор та штучного інтелекту.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Game Programming Patterns. URL: https://gameprogrammingpatterns.com/ (дата звернення: 20.04.2024).

2. Game Engines. URL: https://www.gamedeveloper.com/business/the-top-10-game-engines-for-2023 (дата звернення: 20.04.2024).

3. Game AI Pro 3: Collected Wisdom of Game AI Professionals. URL: https://www.amazon.com/Game-AI-Pro-Collected-Professionals/dp/148225472X (дата звернення: 21.04.2024).

4. Artificial Intelligence and Games. URL: https://www.sciencedirect.com/journal/artificial-intelligence-and-games (дата звернення: 21.04.2024).

5. Game Development Stack Exchange. URL: https://gamedev.stackexchange.com/ (дата звернення: 22.04.2024)

6. Unity Asset Store. URL: https://assetstore.unity.com/ (дата звернення: 22.04.2024).

7. C# Corner. URL: https://www.c-sharpcorner.com/ (дата звернення: 23.04.2024).

8. Microsoft Learn C#. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 23.04.2024).

9. The 10 Most Popular Programming Languages to Learn in 2023. URL: https://www.gamedesigning.org/career/programming-languages/ (дата звернення: 24.04.2024).

10. C# in Depth. URL: https://www.manning.com/books/c-sharp-in-depth-fourth-edition (дата звернення: 24.04.2024).

11. Mastering Unity 2022 Game Development with C#. URL: https://www.packtpub.com/product/mastering-unity-2022-game-development-with-c/9781803231698 (дата звернення: 25.04.2024).

12. Unity Game Development Cookbook. URL: https://www.packtpub.com/product/unity-game-development-cookbook-third-edition/9781838829073 (дата звернення: 25.04.2024).

13. The Art of Game Design: A Book of Lenses. URL: https://www.amazon.com/Art-Game-Design-Book-Lenses/dp/0123694965 (дата звернення: 26.04.2024).

14. A Theory of Fun for Game Design. URL: https://www.amazon.com/Theory-Fun-Game-Design/dp/0596009654 (дата звернення: 26.04.2024).

15. Level Up!: The Guide to Great Video Game Design. URL: https://www.amazon.com/Level-Up-Guide-Great-Video/dp/047068867X (дата звернення: 27.04.2024).

16. Rules of Play: Game Design Fundamentals. URL: https://www.amazon.com/Rules-Play-Game-Design-Fundamentals/dp/0262122817 (дата звернення: 27.04.2024).

17. The Gamification of Learning and Instruction: Game-based Methods and Strategies for Training and Education. URL: https://www.amazon.com/Gamification-Learning-Instruction-Game-based-Strategies/dp/1118334845 (дата звернення: 28.04.2024).

18. Flow: The Psychology of Optimal Experience. URL: https://www.amazon.com/Flow-Psychology-Experience-Perennial-Classics/dp/0061339202 (дата звернення: 28.04.2024).

19. Reality is Broken: Why Games Make Us Better and How They Can Change the World. URL: https://www.amazon.com/Reality-Broken-Games-Better-Change/dp/0143120611 (дата звернення: 29.04.2024).

20. The Design of Everyday Things. URL: https://www.amazon.com/Design-Everyday-Things-Revised-Expanded/dp/0465050654 (дата звернення: 29.04.2024).

21. Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability. URL: https://www.amazon.com/Dont-Make-Me-Think-Revisited/dp/0321965515 (дата звернення: 30.04.2024).

22. The Humane Interface: New Directions for Designing Interactive Systems. URL: https://www.amazon.com/Humane-Interface-Directions-Designing-Interactive/dp/0201379376 (дата звернення: 30.04.2024).

23. Designing Games: A Guide to Engineering Experiences. URL: https://www.amazon.com/Designing-Games-Guide-Engineering-Experiences/dp/1449337937 (дата звернення: 01.05.2024).

24. Game Engine Architecture. URL: https://www.amazon.com/Game-Engine-Architecture-Second-Edition/dp/1482211802 (дата звернення: 01.05.2024).

25. Офіційний сайт ігрового рушія Unity. URL: https://unity.com/ (дата звернення: 02.05.2024).

26. C# Documentation. URL: https://learn.microsoft.com/dotnet/csharp/ (дата звернення: 02.05.2024).

27. Unity Learn. URL: https://learn.unity.com/ (дата звернення: 03.05.2024).

28. Unity Forum. URL: https://forum.unity.com/ (дата звернення: 03.05.2024).

29. Unity Answers. URL: https://answers.unity.com/ (дата звернення: 04.05.2024).

30. Game Development Stack Exchange. URL: https://gamedev.stackexchange.com/ (дата звернення: 04.05.2024)

ДОДАТОК А

(обов'язковий)

ПРОГРАМНИЙ КОД

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

public class BarrelInteract : MonoBehaviour, IInteractable
{

    [SerializeField] private Transform barrelDestroyedPrefab;
    [SerializeField] private GameObject visualGameObject;


    private GridPosition gridPosition;
    private Action onInteractionComplete;
    private bool isActive;
    private float timer;

    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        LevelGrid.Instance.SetInteractableAtGridPosition(gridPosition, this);
        Pathfinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
    }

    private void Update()
    {
        if (!isActive)
        {
            return;
        }

        timer -= Time.deltaTime;

        if (timer <= 0f)
        {
            isActive = false;

            LevelGrid.Instance.ClearInteractableAtGridPosition(gridPosition);
            Pathfinding.Instance.SetIsWalkableGridPosition(gridPosition, true);
```

```
            Destroy(gameObject);
            onInteractionComplete();
        }
    }

    public void Interact(Action onInteractionComplete)
    {
        this.onInteractionComplete = onInteractionComplete;
        isActive = true;
        timer = .5f;

        visualGameObject.SetActive(false);

        Transform barrelDestroyedTransform = Instantiate(barrelDestroyedPrefab,
transform.position, transform.rotation);
        ApplyExplosionToChildren(barrelDestroyedTransform, 250f,
transform.position, 10f);
    }

    private void ApplyExplosionToChildren(Transform root, float explosionForce,
Vector3 explosionPosition, float explosionRange)
    {
        foreach (Transform child in root)
        {
            if (child.TryGetComponent<Rigidbody>(out Rigidbody childRigidbody))
            {
                childRigidbody.AddExplosionForce(explosionForce,
explosionPosition, explosionRange);
            }

            ApplyExplosionToChildren(child, explosionForce, explosionPosition,
explosionRange);
        }
    }

}
```

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

public class BulletProjectile : MonoBehaviour
{

    [SerializeField] private TrailRenderer trailRenderer;
    [SerializeField] private Transform bulletHitVfxPrefab;

    private Vector3 targetPosition;
```

```
    public void Setup(Vector3 targetPosition)
    {
        this.targetPosition = targetPosition;
    }

    private void Update()
    {
        Vector3 moveDir = (targetPosition - transform.position).normalized;

        float distanceBeforeMoving = Vector3.Distance(transform.position,
targetPosition);

        float moveSpeed = 200f;
        transform.position += moveDir * moveSpeed * Time.deltaTime;

        float distanceAfterMoving = Vector3.Distance(transform.position,
targetPosition);

        if (distanceBeforeMoving < distanceAfterMoving)
        {
            transform.position = targetPosition;

            trailRenderer.transform.parent = null;

            Destroy(gameObject);

            Instantiate(bulletHitVfxPrefab, targetPosition, Quaternion.identity);
        }
    }

}using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class CameraController : MonoBehaviour
{

    private const float MIN_FOLLOW_Y_OFFSET = 2f;
    private const float MAX_FOLLOW_Y_OFFSET = 12f;

    [SerializeField] private CinemachineVirtualCamera cinemachineVirtualCamera;

    private CinemachineTransposer cinemachineTransposer;
    private Vector3 targetFollowOffset;

    private void Start()
    {
```

```
        cinemachineTransposer =
cinemachineVirtualCamera.GetCinemachineComponent<CinemachineTransposer>();
        targetFollowOffset = cinemachineTransposer.m_FollowOffset;
    }

    private void Update()
    {
        HandleMovement();
        HandleRotation();
        HandleZoom();
    }

    private void HandleMovement()
    {
        Vector2 inputMoveDir = InputManager.Instance.GetCameraMoveVector();

        float moveSpeed = 10f;

        Vector3 moveVector = transform.forward * inputMoveDir.y + transform.right
* inputMoveDir.x;
        transform.position += moveVector * moveSpeed * Time.deltaTime;
    }

    private void HandleRotation()
    {
        Vector3 rotationVector = new Vector3(0, 0, 0);

        rotationVector.y = InputManager.Instance.GetCameraRotateAmount();

        float rotationSpeed = 100f;
        transform.eulerAngles += rotationVector * rotationSpeed * Time.deltaTime;
    }

    private void HandleZoom()
    {
        float zoomIncreaseAmount = 1f;
        targetFollowOffset.y += InputManager.Instance.GetCameraZoomAmount() *
zoomIncreaseAmount;

        targetFollowOffset.y = Mathf.Clamp(targetFollowOffset.y,
MIN_FOLLOW_Y_OFFSET, MAX_FOLLOW_Y_OFFSET);

        float zoomSpeed = 5f;
        cinemachineTransposer.m_FollowOffset =
            Vector3.Lerp(cinemachineTransposer.m_FollowOffset,
targetFollowOffset, Time.deltaTime * zoomSpeed);
    }

}
```

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

public class CameraManager : MonoBehaviour
{

    [SerializeField] private GameObject actionCameraGameObject;

    private void Start()
    {
        BaseAction.OnAnyActionStarted += BaseAction_OnAnyActionStarted;
        BaseAction.OnAnyActionCompleted += BaseAction_OnAnyActionCompleted;

        HideActionCamera();
    }

    private void ShowActionCamera()
    {
        actionCameraGameObject.SetActive(true);
    }

    private void HideActionCamera()
    {
        actionCameraGameObject.SetActive(false);
    }

    private void BaseAction_OnAnyActionStarted(object sender, EventArgs e)
    {
        switch (sender)
        {
            case ShootAction shootAction:
                Unit shooterUnit = shootAction.GetUnit();
                Unit targetUnit = shootAction.GetTargetUnit();

                Vector3 cameraCharacterHeight = Vector3.up * 1.7f;

                Vector3 shootDir = (targetUnit.GetWorldPosition() -
shooterUnit.GetWorldPosition()).normalized;

                float shoulderOffsetAmount = 0.5f;
                Vector3 shoulderOffset = Quaternion.Euler(0, 90, 0) * shootDir *
shoulderOffsetAmount;

                Vector3 actionCameraPosition =
                    shooterUnit.GetWorldPosition() +
                    cameraCharacterHeight +
                    shoulderOffset +
```

```
                    (shootDir * -1);

                actionCameraGameObject.transform.position = actionCameraPosition;
                actionCameraGameObject.transform.LookAt(targetUnit.GetWorldPositi
on() + cameraCharacterHeight);

                ShowActionCamera();
                break;
            }
        }

        private void BaseAction_OnAnyActionCompleted(object sender, EventArgs e)
        {
            switch (sender)
            {
                case ShootAction shootAction:
                    HideActionCamera();
                    break;
            }
        }
}

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

public class DestructibleCrate : MonoBehaviour
{

    public static event EventHandler OnAnyDestroyed;

    [SerializeField] private Transform crateDestroyedPrefab;


    private GridPosition gridPosition;

    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public void Damage()
    {
```

```
        Transform crateDestroyedTransform = Instantiate(crateDestroyedPrefab,
transform.position, transform.rotation);

        ApplyExplosionToChildren(crateDestroyedTransform, 150f,
transform.position, 10f);

        Destroy(gameObject);

        OnAnyDestroyed?.Invoke(this, EventArgs.Empty);
    }

    private void ApplyExplosionToChildren(Transform root, float explosionForce,
Vector3 explosionPosition, float explosionRange)
    {
        foreach (Transform child in root)
        {
            if (child.TryGetComponent<Rigidbody>(out Rigidbody childRigidbody))
            {
                childRigidbody.AddExplosionForce(explosionForce,
explosionPosition, explosionRange);
            }

            ApplyExplosionToChildren(child, explosionForce, explosionPosition,
explosionRange);
        }
    }

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

public class Door : MonoBehaviour, IInteractable
{

    public static event EventHandler OnAnyDoorOpened;
    public event EventHandler OnDoorOpened;


    [SerializeField] private bool isOpen;

    private GridPosition gridPosition;
    private Animator animator;
    private Action onInteractionComplete;
    private bool isActive;
    private float timer;


    private void Awake()
```

```
{
    animator = GetComponent<Animator>();
}

private void Start()
{
    gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
    LevelGrid.Instance.SetInteractableAtGridPosition(gridPosition, this);

    if (isOpen)
    {
        OpenDoor();
    } else
    {
        CloseDoor();
    }
}

private void Update()
{
    if (!isActive)
    {
        return;
    }

    timer -= Time.deltaTime;

    if (timer <= 0f)
    {
        isActive = false;
        onInteractionComplete();
    }
}



public void Interact(Action onInteractionComplete)
{
    this.onInteractionComplete = onInteractionComplete;
    isActive = true;
    timer = .5f;

    if (isOpen)
    {
        CloseDoor();
    }
    else
    {
        OpenDoor();
```

```
        }
    }

    private void OpenDoor()
    {
        isOpen = true;
        animator.SetBool("IsOpen", isOpen);
        Pathfinding.Instance.SetIsWalkableGridPosition(gridPosition, true);

        OnDoorOpened?.Invoke(this, EventArgs.Empty);
        OnAnyDoorOpened?.Invoke(this, EventArgs.Empty);
    }

    private void CloseDoor()
    {
        isOpen = false;
        animator.SetBool("IsOpen", isOpen);
        Pathfinding.Instance.SetIsWalkableGridPosition(gridPosition, false);
    }

}

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

public class EnemyAI : MonoBehaviour
{

    private enum State
    {
        WaitingForEnemyTurn,
        TakingTurn,
        Busy,
    }

    private State state;
    private float timer;

    private void Awake()
    {
        state = State.WaitingForEnemyTurn;
    }

    private void Start()
    {
        TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;
    }
```

```
    private void Update()
    {
        if (TurnSystem.Instance.IsPlayerTurn())
        {
            return;
        }

        switch (state)
        {
            case State.WaitingForEnemyTurn:
                break;
            case State.TakingTurn:
                timer -= Time.deltaTime;
                if (timer <= 0f)
                {
                    if (TryTakeEnemyAIAction(SetStateTakingTurn))
                    {
                        state = State.Busy;
                    } else
                    {
                        // No more enemies have actions they can take, end enemy
turn
                        TurnSystem.Instance.NextTurn();
                    }
                }
                break;
            case State.Busy:
                break;
        }
    }

    private void SetStateTakingTurn()
    {
        timer = 0.5f;
        state = State.TakingTurn;
    }

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

    private bool TryTakeEnemyAIAction(Action onEnemyAIActionComplete)
    {
```

```
        foreach (Unit enemyUnit in UnitManager.Instance.GetEnemyUnitList())
        {
            if (TryTakeEnemyAIAction(enemyUnit, onEnemyAIActionComplete))
            {
                return true;
            }
        }

        return false;
    }

    private bool TryTakeEnemyAIAction(Unit enemyUnit, Action
onEnemyAIActionComplete)
    {
        EnemyAIAction bestEnemyAIAction = null;
        BaseAction bestBaseAction = null;

        foreach (BaseAction baseAction in enemyUnit.GetBaseActionArray())
        {
            if (!enemyUnit.CanSpendActionPointsToTakeAction(baseAction))
            {
                // Enemy cannot afford this action
                continue;
            }

            if (bestEnemyAIAction == null)
            {
                bestEnemyAIAction = baseAction.GetBestEnemyAIAction();
                bestBaseAction = baseAction;
            }
            else
            {
                EnemyAIAction testEnemyAIAction =
baseAction.GetBestEnemyAIAction();
                if (testEnemyAIAction != null && testEnemyAIAction.actionValue >
bestEnemyAIAction.actionValue)
                {
                    bestEnemyAIAction = testEnemyAIAction;
                    bestBaseAction = baseAction;
                }
            }

        }

        if (bestEnemyAIAction != null &&
enemyUnit.TrySpendActionPointsToTakeAction(bestBaseAction))
        {
            bestBaseAction.TakeAction(bestEnemyAIAction.gridPosition,
onEnemyAIActionComplete);
```

```
            return true;
        }
        else
        {
            return false;
        }
    }

}

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

public class EnemyAIAction
{

    public GridPosition gridPosition;
    public int actionValue;

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

public class GrenadeProjectile : MonoBehaviour
{
    public static event EventHandler OnAnyGrenadeExploded;


    [SerializeField] private Transform grenadeExplodeVfxPrefab;
    [SerializeField] private TrailRenderer trailRenderer;
    [SerializeField] private AnimationCurve arcYAnimationCurve;


    private Vector3 targetPosition;
    private Action onGrenadeBehaviourComplete;
    private float totalDistance;
    private Vector3 positionXZ;

    private void Update()
    {
        Vector3 moveDir = (targetPosition - positionXZ).normalized;

        float moveSpeed = 15f;
        positionXZ += moveDir * moveSpeed * Time.deltaTime;

        float distance = Vector3.Distance(positionXZ, targetPosition);
        float distanceNormalized = 1 - distance / totalDistance;
```

```
        float maxHeight = totalDistance / 4f;
        float positionY = arcYAnimationCurve.Evaluate(distanceNormalized) *
maxHeight;
        transform.position = new Vector3(positionXZ.x, positionY, positionXZ.z);

        float reachedTargetDistance = .2f;
        if (Vector3.Distance(positionXZ, targetPosition) < reachedTargetDistance)
        {
            float damageRadius = 4f;
            Collider[] colliderArray = Physics.OverlapSphere(targetPosition,
damageRadius);

            foreach (Collider collider in colliderArray)
            {
                if (collider.TryGetComponent<Unit>(out Unit targetUnit))
                {
                    targetUnit.Damage(30);
                }
                if (collider.TryGetComponent<DestructibleCrate>(out
DestructibleCrate destructibleCrate))
                {
                    destructibleCrate.Damage();
                }
            }

            OnAnyGrenadeExploded?.Invoke(this, EventArgs.Empty);

            trailRenderer.transform.parent = null;

            Instantiate(grenadeExplodeVfxPrefab, targetPosition + Vector3.up *
1f, Quaternion.identity);

            Destroy(gameObject);

            onGrenadeBehaviourComplete();
        }
    }


    public void Setup(GridPosition targetGridPosition, Action
onGrenadeBehaviourComplete)
    {
        this.onGrenadeBehaviourComplete = onGrenadeBehaviourComplete;
        targetPosition = LevelGrid.Instance.GetWorldPosition(targetGridPosition);

        positionXZ = transform.position;
        positionXZ.y = 0;
        totalDistance = Vector3.Distance(positionXZ, targetPosition);
```

```
    }

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

public class HealthSystem : MonoBehaviour
{

    public event EventHandler OnDead;
    public event EventHandler OnDamaged;


    [SerializeField] private int health = 100;
    private int healthMax;


    private void Awake()
    {
        healthMax = health;
    }

    public void Damage(int damageAmount)
    {
        health -= damageAmount;

        if (health < 0)
        {
            health = 0;
        }

        OnDamaged?.Invoke(this, EventArgs.Empty);

        if (health == 0)
        {
            Die();
        }
    }

    private void Die()
    {
        OnDead?.Invoke(this, EventArgs.Empty);
    }

    public float GetHealthNormalized()
    {
        return (float)health / healthMax;
    }
```

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

public interface IInteractable
{

    void Interact(Action onInteractionComplete);

}

}

#define USE_NEW_INPUT_SYSTEM
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

public class InputManager : MonoBehaviour
{

    public static InputManager Instance { get; private set; }

    private PlayerInputActions playerInputActions;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one InputManager! " + transform + "
- " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        playerInputActions = new PlayerInputActions();
        playerInputActions.Player.Enable();
    }


    public Vector2 GetMouseScreenPosition()
    {
#if USE_NEW_INPUT_SYSTEM
        return Mouse.current.position.ReadValue();
#else
```

```
        return Input.mousePosition;
#endif
    }

    public bool IsMouseButtonDownThisFrame()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.Click.WasPressedThisFrame();
#else
        return Input.GetMouseButtonDown(0);
#endif
    }

    public Vector2 GetCameraMoveVector()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraMovement.ReadValue<Vector2>();
#else
        Vector2 inputMoveDir = new Vector2(0, 0);

        if (Input.GetKey(KeyCode.W))
        {
            inputMoveDir.y = +1f;
        }
        if (Input.GetKey(KeyCode.S))
        {
            inputMoveDir.y = -1f;
        }
        if (Input.GetKey(KeyCode.A))
        {
            inputMoveDir.x = -1f;
        }
        if (Input.GetKey(KeyCode.D))
        {
            inputMoveDir.x = +1f;
        }

        return inputMoveDir;
#endif
    }

    public float GetCameraRotateAmount()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraRotate.ReadValue<float>();
#else
        float rotateAmount = 0f;

        if (Input.GetKey(KeyCode.Q))
```

```
        {
            rotateAmount = +1f;
        }
        if (Input.GetKey(KeyCode.E))
        {
            rotateAmount = -1f;
        }

        return rotateAmount;
#endif
    }

    public float GetCameraZoomAmount()
    {
#if USE_NEW_INPUT_SYSTEM
        return playerInputActions.Player.CameraZoom.ReadValue<float>();
#else
        float zoomAmount = 0f;

        if (Input.mouseScrollDelta.y > 0)
        {
            zoomAmount = -1f;
        }
        if (Input.mouseScrollDelta.y < 0)
        {
            zoomAmount = +1f;
        }

        return zoomAmount;
#endif
    }


}

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

public class InteractSphere : MonoBehaviour, IInteractable
{

    [SerializeField] private Material greenMaterial;
    [SerializeField] private Material redMaterial;
    [SerializeField] private MeshRenderer meshRenderer;

    private GridPosition gridPosition;
    private bool isGreen;
```

```
private Action onInteractionComplete;
private bool isActive;
private float timer;

private void Start()
{
    gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
    LevelGrid.Instance.SetInteractableAtGridPosition(gridPosition, this);

    SetColorGreen();
}

private void Update()
{
    if (!isActive)
    {
        return;
    }

    timer -= Time.deltaTime;

    if (timer <= 0f)
    {
        isActive = false;
        onInteractionComplete();
    }
}

private void SetColorGreen()
{
    isGreen = true;
    meshRenderer.material = greenMaterial;
}

private void SetColorRed()
{
    isGreen = false;
    meshRenderer.material = redMaterial;
}

public void Interact(Action onInteractionComplete)
{
    this.onInteractionComplete = onInteractionComplete;
    isActive = true;
    timer = .5f;

    if (isGreen)
    {
        SetColorRed();
```

```
        } else
        {
            SetColorGreen();
        }
    }

}

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

public class LevelGrid : MonoBehaviour
{

    public static LevelGrid Instance { get; private set; }


    public event EventHandler<OnAnyUnitMovedGridPositionEventArgs>
OnAnyUnitMovedGridPosition;
    public class OnAnyUnitMovedGridPositionEventArgs : EventArgs
    {
        public Unit unit;
        public GridPosition fromGridPosition;
        public GridPosition toGridPosition;
    }


    [SerializeField] private Transform gridDebugObjectPrefab;
    [SerializeField] private int width;
    [SerializeField] private int height;
    [SerializeField] private float cellSize;

    private GridSystem<GridObject> gridSystem;


    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one LevelGrid! " + transform + " -
" + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        gridSystem = new GridSystem<GridObject>(width, height, cellSize,
```

```
                (GridSystem<GridObject> g, GridPosition gridPosition) => new
GridObject(g, gridPosition));
        //gridSystem.CreateDebugObjects(gridDebugObjectPrefab);
    }

    private void Start()
    {
        Pathfinding.Instance.Setup(width, height, cellSize);
    }


    public void AddUnitAtGridPosition(GridPosition gridPosition, Unit unit)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        gridObject.AddUnit(unit);
    }

    public List<Unit> GetUnitListAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        return gridObject.GetUnitList();
    }

    public void RemoveUnitAtGridPosition(GridPosition gridPosition, Unit unit)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        gridObject.RemoveUnit(unit);
    }

    public void UnitMovedGridPosition(Unit unit, GridPosition fromGridPosition,
GridPosition toGridPosition)
    {
        RemoveUnitAtGridPosition(fromGridPosition, unit);

        AddUnitAtGridPosition(toGridPosition, unit);

        OnAnyUnitMovedGridPosition?.Invoke(this, new
OnAnyUnitMovedGridPositionEventArgs {
            unit = unit,
            fromGridPosition = fromGridPosition,
            toGridPosition = toGridPosition,
        });
    }

    public GridPosition GetGridPosition(Vector3 worldPosition) =>
gridSystem.GetGridPosition(worldPosition);

    public Vector3 GetWorldPosition(GridPosition gridPosition) =>
gridSystem.GetWorldPosition(gridPosition);
```

```
    public bool IsValidGridPosition(GridPosition gridPosition) =>
gridSystem.IsValidGridPosition(gridPosition);

    public int GetWidth() => gridSystem.GetWidth();

    public int GetHeight() => gridSystem.GetHeight();

    public bool HasAnyUnitOnGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        return gridObject.HasAnyUnit();
    }

    public Unit GetUnitAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        return gridObject.GetUnit();
    }

    public IInteractable GetInteractableAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        return gridObject.GetInteractable();
    }

    public void SetInteractableAtGridPosition(GridPosition gridPosition,
IInteractable interactable)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        gridObject.SetInteractable(interactable);
    }

    public void ClearInteractableAtGridPosition(GridPosition gridPosition)
    {
        GridObject gridObject = gridSystem.GetGridObject(gridPosition);
        gridObject.ClearInteractable();
    }


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

public class LevelScripting : MonoBehaviour
{
```

```
    [SerializeField] private List<GameObject> hider1List;
    [SerializeField] private List<GameObject> hider2List;
    [SerializeField] private List<GameObject> hider3List;
    [SerializeField] private List<GameObject> enemy1List;
    [SerializeField] private List<GameObject> enemy2List;
    [SerializeField] private Door door1;
    [SerializeField] private Door door2;

    private bool hasShownFirstHider = false;

    private void Start()
    {
        LevelGrid.Instance.OnAnyUnitMovedGridPosition +=
LevelGrid_OnAnyUnitMovedGridPosition;
        door1.OnDoorOpened += (object sender, EventArgs e) =>
        {
            SetActiveGameObjectList(hider2List, false);
        };
        door2.OnDoorOpened += (object sender, EventArgs e) =>
        {
            SetActiveGameObjectList(hider3List, false);
            SetActiveGameObjectList(enemy2List, true);
        };
    }

    private void LevelGrid_OnAnyUnitMovedGridPosition(object sender,
LevelGrid.OnAnyUnitMovedGridPositionEventArgs e)
    {
        if (e.toGridPosition.z == 5 && !hasShownFirstHider)
        {
            hasShownFirstHider = true;
            SetActiveGameObjectList(hider1List, false);
            SetActiveGameObjectList(enemy1List, true);
        }
    }

    private void SetActiveGameObjectList(List<GameObject> gameObjectList, bool
isActive)
    {
        foreach (GameObject gameObject in gameObjectList)
        {
            gameObject.SetActive(isActive);
        }
    }

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

```
public class LookAtCamera : MonoBehaviour
{

    [SerializeField] private bool invert;


    private Transform cameraTransform;


    private void Awake()
    {
        cameraTransform = Camera.main.transform;
    }

    private void LateUpdate()
    {
        if (invert)
        {
            Vector3 dirToCamera = (cameraTransform.position –
transform.position).normalized;
            transform.LookAt(transform.position + dirToCamera * –1);
        } else
        {
            transform.LookAt(cameraTransform);
        }
    }
}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MouseWorld : MonoBehaviour
{

    private static MouseWorld instance;


    [SerializeField] private LayerMask mousePlaneLayerMask;

    private void Awake()
    {
        instance = this;
    }

    public static Vector3 GetPosition()
    {
        Ray ray =
Camera.main.ScreenPointToRay(InputManager.Instance.GetMouseScreenPosition());
```

```
        Physics.Raycast(ray, out RaycastHit raycastHit, float.MaxValue,
instance.mousePlaneLayerMask);
        return raycastHit.point;
    }

}


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

public class Pathfinding : MonoBehaviour
{

    public static Pathfinding Instance { get; private set; }


    private const int MOVE_STRAIGHT_COST = 10;
    private const int MOVE_DIAGONAL_COST = 14;

    [SerializeField] private Transform gridDebugObjectPrefab;
    [SerializeField] private LayerMask obstaclesLayerMask;


    private int width;
    private int height;
    private float cellSize;
    private GridSystem<PathNode> gridSystem;

    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one Pathfinding! " + transform + "
- " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

    }

    public void Setup(int width, int height, float cellSize)
    {
        this.width = width;
        this.height = height;
        this.cellSize = cellSize;
```

```
        gridSystem = new GridSystem<PathNode>(width, height, cellSize,
            (GridSystem<PathNode> g, GridPosition gridPosition) => new
PathNode(gridPosition));

        //gridSystem.CreateDebugObjects(gridDebugObjectPrefab);

        for (int x = 0; x < width; x++)
        {
            for (int z = 0; z < height; z++)
            {
                GridPosition gridPosition = new GridPosition(x, z);
                Vector3 worldPosition =
LevelGrid.Instance.GetWorldPosition(gridPosition);
                float raycastOffsetDistance = 5f;
                if (Physics.Raycast(
                    worldPosition + Vector3.down * raycastOffsetDistance,
                    Vector3.up,
                    raycastOffsetDistance * 2,
                    obstaclesLayerMask))
                {
                    GetNode(x, z).SetIsWalkable(false);
                }
            }
        }
    }

    public List<GridPosition> FindPath(GridPosition startGridPosition,
GridPosition endGridPosition, out int pathLength)
    {
        List<PathNode> openList = new List<PathNode>();
        List<PathNode> closedList = new List<PathNode>();

        PathNode startNode = gridSystem.GetGridObject(startGridPosition);
        PathNode endNode = gridSystem.GetGridObject(endGridPosition);
        openList.Add(startNode);

        for (int x = 0; x < gridSystem.GetWidth(); x++)
        {
            for (int z = 0; z < gridSystem.GetHeight(); z++)
            {
                GridPosition gridPosition = new GridPosition(x, z);
                PathNode pathNode = gridSystem.GetGridObject(gridPosition);

                pathNode.SetGCost(int.MaxValue);
                pathNode.SetHCost(0);
                pathNode.CalculateFCost();
                pathNode.ResetCameFromPathNode();
            }
        }
```

```
        startNode.SetGCost(0);
        startNode.SetHCost(CalculateDistance(startGridPosition,
endGridPosition));
        startNode.CalculateFCost();

        while (openList.Count > 0)
        {
            PathNode currentNode = GetLowestFCostPathNode(openList);

            if (currentNode == endNode)
            {
                // Reached final node
                pathLength = endNode.GetFCost();
                return CalculatePath(endNode);
            }

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

            foreach (PathNode neighbourNode in GetNeighbourList(currentNode))
            {
                if (closedList.Contains(neighbourNode))
                {
                    continue;
                }

                if (!neighbourNode.IsWalkable())
                {
                    closedList.Add(neighbourNode);
                    continue;
                }

                int tentativeGCost =
                    currentNode.GetGCost() +
CalculateDistance(currentNode.GetGridPosition(),
neighbourNode.GetGridPosition());

                if (tentativeGCost < neighbourNode.GetGCost())
                {
                    neighbourNode.SetCameFromPathNode(currentNode);
                    neighbourNode.SetGCost(tentativeGCost);
                    neighbourNode.SetHCost(CalculateDistance(neighbourNode.GetGri
dPosition(), endGridPosition));
                    neighbourNode.CalculateFCost();

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

```
                    }
                }
            }
        }

        // No path found
        pathLength = 0;
        return null;
    }

    public int CalculateDistance(GridPosition gridPositionA, GridPosition
gridPositionB)
    {
        GridPosition gridPositionDistance = gridPositionA - gridPositionB;
        int xDistance = Mathf.Abs(gridPositionDistance.x);
        int zDistance = Mathf.Abs(gridPositionDistance.z);
        int remaining = Mathf.Abs(xDistance - zDistance);
        return MOVE_DIAGONAL_COST * Mathf.Min(xDistance, zDistance) +
MOVE_STRAIGHT_COST * remaining;
    }

    private PathNode GetLowestFCostPathNode(List<PathNode> pathNodeList)
    {
        PathNode lowestFCostPathNode = pathNodeList[0];
        for (int i = 0; i < pathNodeList.Count; i++)
        {
            if (pathNodeList[i].GetFCost() < lowestFCostPathNode.GetFCost())
            {
                lowestFCostPathNode = pathNodeList[i];
            }
        }
        return lowestFCostPathNode;
    }

    private PathNode GetNode(int x, int z)
    {
        return gridSystem.GetGridObject(new GridPosition(x, z));
    }

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

        GridPosition gridPosition = currentNode.GetGridPosition();

        if (gridPosition.x - 1 >= 0)
        {
            // Left
            neighbourList.Add(GetNode(gridPosition.x - 1, gridPosition.z + 0));
```

```
            if (gridPosition.z - 1 >= 0)
            {
                // Left Down
                neighbourList.Add(GetNode(gridPosition.x - 1, gridPosition.z -
1));
            }

            if (gridPosition.z + 1 < gridSystem.GetHeight())
            {
                // Left Up
                neighbourList.Add(GetNode(gridPosition.x - 1, gridPosition.z +
1));
            }
        }

        if (gridPosition.x + 1 < gridSystem.GetWidth())
        {
            // Right
            neighbourList.Add(GetNode(gridPosition.x + 1, gridPosition.z + 0));
            if (gridPosition.z - 1 >= 0)
            {
                // Right Down
                neighbourList.Add(GetNode(gridPosition.x + 1, gridPosition.z -
1));
            }
            if (gridPosition.z + 1 < gridSystem.GetHeight())
            {
                // Right Up
                neighbourList.Add(GetNode(gridPosition.x + 1, gridPosition.z +
1));
            }
        }

        if (gridPosition.z - 1 >= 0)
        {
            // Down
            neighbourList.Add(GetNode(gridPosition.x + 0, gridPosition.z - 1));
        }
        if (gridPosition.z + 1 < gridSystem.GetHeight())
        {
            // Up
            neighbourList.Add(GetNode(gridPosition.x + 0, gridPosition.z + 1));
        }

        return neighbourList;
    }

    private List<GridPosition> CalculatePath(PathNode endNode)
    {
```

```
        List<PathNode> pathNodeList = new List<PathNode>();
        pathNodeList.Add(endNode);
        PathNode currentNode = endNode;
        while (currentNode.GetCameFromPathNode() != null)
        {
            pathNodeList.Add(currentNode.GetCameFromPathNode());
            currentNode = currentNode.GetCameFromPathNode();
        }

        pathNodeList.Reverse();

        List<GridPosition> gridPositionList = new List<GridPosition>();
        foreach (PathNode pathNode in pathNodeList)
        {
            gridPositionList.Add(pathNode.GetGridPosition());
        }

        return gridPositionList;
    }

    public void SetIsWalkableGridPosition(GridPosition gridPosition, bool
isWalkable)
    {
        gridSystem.GetGridObject(gridPosition).SetIsWalkable(isWalkable);
    }

    public bool IsWalkableGridPosition(GridPosition gridPosition)
    {
        return gridSystem.GetGridObject(gridPosition).IsWalkable();
    }

    public bool HasPath(GridPosition startGridPosition, GridPosition
endGridPosition)
    {
        return FindPath(startGridPosition, endGridPosition, out int pathLength)
!= null;
    }

    public int GetPathLength(GridPosition startGridPosition, GridPosition
endGridPosition)
    {
        FindPath(startGridPosition, endGridPosition, out int pathLength);
        return pathLength;
    }

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

```
using TMPro;

public class PathfindingGridDebugObject : GridDebugObject
{

    [SerializeField] private TextMeshPro gCostText;
    [SerializeField] private TextMeshPro hCostText;
    [SerializeField] private TextMeshPro fCostText;
    [SerializeField] private SpriteRenderer isWalkableSpriteRenderer;

    private PathNode pathNode;

    public override void SetGridObject(object gridObject)
    {
        base.SetGridObject(gridObject);
        pathNode = (PathNode)gridObject;
    }

    protected override void Update()
    {
        base.Update();
        gCostText.text = pathNode.GetGCost().ToString();
        hCostText.text = pathNode.GetHCost().ToString();
        fCostText.text = pathNode.GetFCost().ToString();
        isWalkableSpriteRenderer.color = pathNode.IsWalkable() ? Color.green :
Color.red;
    }

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

public class PathfindingUpdater : MonoBehaviour
{


    private void Start()
    {
        DestructibleCrate.OnAnyDestroyed += DestructibleCrate_OnAnyDestroyed;
    }

    private void DestructibleCrate_OnAnyDestroyed(object sender, EventArgs e)
    {
        DestructibleCrate destructibleCrate = sender as DestructibleCrate;
        Pathfinding.Instance.SetIsWalkableGridPosition(destructibleCrate.GetGridP
osition(), true);
    }
```

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

public class PathNode
{

    private GridPosition gridPosition;
    private int gCost;
    private int hCost;
    private int fCost;
    private PathNode cameFromPathNode;
    private bool isWalkable = true;

    public PathNode(GridPosition gridPosition)
    {
        this.gridPosition = gridPosition;
    }

    public override string ToString()
    {
        return gridPosition.ToString();
    }

    public int GetGCost()
    {
        return gCost;
    }

    public int GetHCost()
    {
        return hCost;
    }

    public int GetFCost()
    {
        return fCost;
    }

    public void SetGCost(int gCost)
    {
        this.gCost = gCost;
    }

    public void SetHCost(int hCost)
    {
        this.hCost = hCost;
    }
```

```
    public void CalculateFCost()
    {
        fCost = gCost + hCost;
    }

    public void ResetCameFromPathNode()
    {
        cameFromPathNode = null;
    }

    public void SetCameFromPathNode(PathNode pathNode)
    {
        cameFromPathNode = pathNode;
    }

    public PathNode GetCameFromPathNode()
    {
        return cameFromPathNode;
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public bool IsWalkable()
    {
        return isWalkable;
    }

    public void SetIsWalkable(bool isWalkable)
    {
        this.isWalkable = isWalkable;
    }

}
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Cinemachine;

public class ScreenShake : MonoBehaviour
{

    public static ScreenShake Instance { get; private set; }


    private CinemachineImpulseSource cinemachineImpulseSource;
```

```
    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one ScreenShake! " + transform + "
- " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        cinemachineImpulseSource = GetComponent<CinemachineImpulseSource>();
    }

    public void Shake(float intensity = 1f)
    {
        cinemachineImpulseSource.GenerateImpulse(intensity);
    }

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

public class ScreenShakeActions : MonoBehaviour
{


    private void Start()
    {
        ShootAction.OnAnyShoot += ShootAction_OnAnyShoot;
        GrenadeProjectile.OnAnyGrenadeExploded +=
GrenadeProjectile_OnAnyGrenadeExploded;
        SwordAction.OnAnySwordHit += SwordAction_OnAnySwordHit;
    }

    private void SwordAction_OnAnySwordHit(object sender, EventArgs e)
    {
        ScreenShake.Instance.Shake(2f);
    }

    private void GrenadeProjectile_OnAnyGrenadeExploded(object sender, EventArgs
e)
    {
        ScreenShake.Instance.Shake(5f);
    }
```

```
    private void ShootAction_OnAnyShoot(object sender,
ShootAction.OnShootEventArgs e)
    {
        ScreenShake.Instance.Shake();
    }

}

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

public class Testing : MonoBehaviour
{

    [SerializeField] private Unit unit;

    private void Start()
    {
    }

    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.T))
        {
        }
    }

}

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

public class TurnSystem : MonoBehaviour
{

    public static TurnSystem Instance { get; private set; }


    public event EventHandler OnTurnChanged;


    private int turnNumber = 1;
    private bool isPlayerTurn = true;
```

```
    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one TurnSystem! " + transform + " -
" + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }


    public void NextTurn()
    {
        turnNumber++;
        isPlayerTurn = !isPlayerTurn;

        OnTurnChanged?.Invoke(this, EventArgs.Empty);
    }

    public int GetTurnNumber()
    {
        return turnNumber;
    }

    public bool IsPlayerTurn()
    {
        return isPlayerTurn;
    }

}

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

public class Unit : MonoBehaviour
{
    private const int ACTION_POINTS_MAX = 3;


    public static event EventHandler OnAnyActionPointsChanged;
    public static event EventHandler OnAnyUnitSpawned;
    public static event EventHandler OnAnyUnitDead;


    [SerializeField] private bool isEnemy;
```

```
    private GridPosition gridPosition;
    private HealthSystem healthSystem;
    private BaseAction[] baseActionArray;
    private int actionPoints = ACTION_POINTS_MAX;

    private void Awake()
    {
        healthSystem = GetComponent<HealthSystem>();
        baseActionArray = GetComponents<BaseAction>();
    }

    private void Start()
    {
        gridPosition = LevelGrid.Instance.GetGridPosition(transform.position);
        LevelGrid.Instance.AddUnitAtGridPosition(gridPosition, this);

        TurnSystem.Instance.OnTurnChanged += TurnSystem_OnTurnChanged;

        healthSystem.OnDead += HealthSystem_OnDead;

        OnAnyUnitSpawned?.Invoke(this, EventArgs.Empty);
    }

    private void Update()
    {
        GridPosition newGridPosition =
LevelGrid.Instance.GetGridPosition(transform.position);
        if (newGridPosition != gridPosition)
        {
            // Unit changed Grid Position
            GridPosition oldGridPosition = gridPosition;
            gridPosition = newGridPosition;

            LevelGrid.Instance.UnitMovedGridPosition(this, oldGridPosition,
newGridPosition);
        }
    }

    public T GetAction<T>() where T : BaseAction
    {
        foreach (BaseAction baseAction in baseActionArray)
        {
            if (baseAction is T)
            {
                return (T)baseAction;
            }
        }
```

```
        return null;
    }

    public GridPosition GetGridPosition()
    {
        return gridPosition;
    }

    public Vector3 GetWorldPosition()
    {
        return transform.position;
    }

    public BaseAction[] GetBaseActionArray()
    {
        return baseActionArray;
    }

    public bool TrySpendActionPointsToTakeAction(BaseAction baseAction)
    {
        if (CanSpendActionPointsToTakeAction(baseAction))
        {
            SpendActionPoints(baseAction.GetActionPointsCost());
            return true;
        } else
        {
            return false;
        }
    }

    public bool CanSpendActionPointsToTakeAction(BaseAction baseAction)
    {
        if (actionPoints >= baseAction.GetActionPointsCost())
        {
            return true;
        } else
        {
            return false;
        }
    }

    private void SpendActionPoints(int amount)
    {
        actionPoints -= amount;

        OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
    }

    public int GetActionPoints()
```

```
    {
        return actionPoints;
    }

    private void TurnSystem_OnTurnChanged(object sender, EventArgs e)
    {
        if ((IsEnemy() && !TurnSystem.Instance.IsPlayerTurn()) ||
            (!IsEnemy() && TurnSystem.Instance.IsPlayerTurn()))
        {
            actionPoints = ACTION_POINTS_MAX;

            OnAnyActionPointsChanged?.Invoke(this, EventArgs.Empty);
        }
    }

    public bool IsEnemy()
    {
        return isEnemy;
    }

    public void Damage(int damageAmount)
    {
        healthSystem.Damage(damageAmount);
    }

    private void HealthSystem_OnDead(object sender, EventArgs e)
    {
        LevelGrid.Instance.RemoveUnitAtGridPosition(gridPosition, this);

        Destroy(gameObject);

        OnAnyUnitDead?.Invoke(this, EventArgs.Empty);
    }

    public float GetHealthNormalized()
    {
        return healthSystem.GetHealthNormalized();
    }

}



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

```
public class UnitActionSystem : MonoBehaviour
{

    public static UnitActionSystem Instance { get; private set; }


    public event EventHandler OnSelectedUnitChanged;
    public event EventHandler OnSelectedActionChanged;
    public event EventHandler<bool> OnBusyChanged;
    public event EventHandler OnActionStarted;


    [SerializeField] private Unit selectedUnit;
    [SerializeField] private LayerMask unitLayerMask;

    private BaseAction selectedAction;
    private bool isBusy;


    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one UnitActionSystem! " + transform
+ " - " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;
    }

    private void Start()
    {
        SetSelectedUnit(selectedUnit);
    }

    private void Update()
    {
        if (isBusy)
        {
            return;
        }

        if (!TurnSystem.Instance.IsPlayerTurn())
        {
            return;
        }

        if (EventSystem.current.IsPointerOverGameObject())
```

```
        {
            return;
        }

        if (TryHandleUnitSelection())
        {
            return;
        }

        HandleSelectedAction();
    }

    private void HandleSelectedAction()
    {
        if (InputManager.Instance.IsMouseButtonDownThisFrame())
        {
            GridPosition mouseGridPosition =
LevelGrid.Instance.GetGridPosition(MouseWorld.GetPosition());

            if (!selectedAction.IsValidActionGridPosition(mouseGridPosition))
            {
                return;
            }

            if (!selectedUnit.TrySpendActionPointsToTakeAction(selectedAction))
            {
                return;
            }

            SetBusy();
            selectedAction.TakeAction(mouseGridPosition, ClearBusy);

            OnActionStarted?.Invoke(this, EventArgs.Empty);
        }
    }


    private void SetBusy()
    {
        isBusy = true;

        OnBusyChanged?.Invoke(this, isBusy);
    }

    private void ClearBusy()
    {
        isBusy = false;

        OnBusyChanged?.Invoke(this, isBusy);
```

```
    }

    private bool TryHandleUnitSelection()
    {
        if (InputManager.Instance.IsMouseButtonDownThisFrame())
        {
            Ray ray =
Camera.main.ScreenPointToRay(InputManager.Instance.GetMouseScreenPosition());
            if (Physics.Raycast(ray, out RaycastHit raycastHit, float.MaxValue,
unitLayerMask))
            {
                if (raycastHit.transform.TryGetComponent<Unit>(out Unit unit))
                {
                    if (unit == selectedUnit)
                    {
                        // Unit is already selected
                        return false;
                    }

                    if (unit.IsEnemy())
                    {
                        // Clicked on an Enemy
                        return false;
                    }

                    SetSelectedUnit(unit);
                    return true;
                }
            }
        }

        return false;
    }

    private void SetSelectedUnit(Unit unit)
    {
        selectedUnit = unit;

        SetSelectedAction(unit.GetAction<MoveAction>());

        OnSelectedUnitChanged?.Invoke(this, EventArgs.Empty);
    }

    public void SetSelectedAction(BaseAction baseAction)
    {
        selectedAction = baseAction;

        OnSelectedActionChanged?.Invoke(this, EventArgs.Empty);
    }
```

```
    public Unit GetSelectedUnit()
    {
        return selectedUnit;
    }

    public BaseAction GetSelectedAction()
    {
        return selectedAction;
    }

}

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

public class UnitAnimator : MonoBehaviour
{

    [SerializeField] private Animator animator;
    [SerializeField] private Transform bulletProjectilePrefab;
    [SerializeField] private Transform shootPointTransform;
    [SerializeField] private Transform rifleTransform;
    [SerializeField] private Transform swordTransform;


    private void Awake()
    {
        if (TryGetComponent<MoveAction>(out MoveAction moveAction))
        {
            moveAction.OnStartMoving += MoveAction_OnStartMoving;
            moveAction.OnStopMoving += MoveAction_OnStopMoving;
        }

        if (TryGetComponent<ShootAction>(out ShootAction shootAction))
        {
            shootAction.OnShoot += ShootAction_OnShoot;
        }

        if (TryGetComponent<SwordAction>(out SwordAction swordAction))
        {
            swordAction.OnSwordActionStarted += SwordAction_OnSwordActionStarted;
            swordAction.OnSwordActionCompleted +=
SwordAction_OnSwordActionCompleted;
        }
    }
```

```
    private void Start()
    {
        EquipRifle();
    }

    private void SwordAction_OnSwordActionCompleted(object sender, EventArgs e)
    {
        EquipRifle();
    }

    private void SwordAction_OnSwordActionStarted(object sender, EventArgs e)
    {
        EquipSword();
        animator.SetTrigger("SwordSlash");
    }

    private void MoveAction_OnStartMoving(object sender, EventArgs e)
    {
        animator.SetBool("IsWalking", true);
    }

    private void MoveAction_OnStopMoving(object sender, EventArgs e)
    {
        animator.SetBool("IsWalking", false);
    }

    private void ShootAction_OnShoot(object sender, ShootAction.OnShootEventArgs
e)
    {
        animator.SetTrigger("Shoot");

        Transform bulletProjectileTransform =
            Instantiate(bulletProjectilePrefab, shootPointTransform.position,
Quaternion.identity);

        BulletProjectile bulletProjectile =
bulletProjectileTransform.GetComponent<BulletProjectile>();

        Vector3 targetUnitShootAtPosition = e.targetUnit.GetWorldPosition();

        targetUnitShootAtPosition.y = shootPointTransform.position.y;

        bulletProjectile.Setup(targetUnitShootAtPosition);
    }

    private void EquipSword()
    {
        swordTransform.gameObject.SetActive(true);
        rifleTransform.gameObject.SetActive(false);
```

```
    }

    private void EquipRifle()
    {
        swordTransform.gameObject.SetActive(false);
        rifleTransform.gameObject.SetActive(true);
    }

}


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

public class UnitManager : MonoBehaviour
{

    public static UnitManager Instance { get; private set; }


    private List<Unit> unitList;
    private List<Unit> friendlyUnitList;
    private List<Unit> enemyUnitList;


    private void Awake()
    {
        if (Instance != null)
        {
            Debug.LogError("There's more than one UnitManager! " + transform + "
- " + Instance);
            Destroy(gameObject);
            return;
        }
        Instance = this;

        unitList = new List<Unit>();
        friendlyUnitList = new List<Unit>();
        enemyUnitList = new List<Unit>();
    }

    private void Start()
    {
        Unit.OnAnyUnitSpawned += Unit_OnAnyUnitSpawned;
        Unit.OnAnyUnitDead += Unit_OnAnyUnitDead;
    }
```

```
    private void Unit_OnAnyUnitSpawned(object sender, EventArgs e)
    {
        Unit unit = sender as Unit;

        unitList.Add(unit);

        if (unit.IsEnemy())
        {
            enemyUnitList.Add(unit);
        } else
        {
            friendlyUnitList.Add(unit);
        }
    }

    private void Unit_OnAnyUnitDead(object sender, EventArgs e)
    {
        Unit unit = sender as Unit;

        unitList.Remove(unit);

        if (unit.IsEnemy())
        {
            enemyUnitList.Remove(unit);
        }
        else
        {
            friendlyUnitList.Remove(unit);
        }
    }

    public List<Unit> GetUnitList()
    {
        return unitList;
    }

    public List<Unit> GetFriendlyUnitList()
    {
        return friendlyUnitList;
    }

    public List<Unit> GetEnemyUnitList()
    {
        return enemyUnitList;
    }

}
```

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

public class UnitRagdoll : MonoBehaviour
{

    [SerializeField] private Transform ragdollRootBone;

    public void Setup(Transform originalRootBone)
    {
        MatchAllChildTransforms(originalRootBone, ragdollRootBone);

        Vector3 randomDir = new Vector3(Random.Range(-1f, +1f), 0, Random.Range(-
1f, +1f));
        ApplyExplosionToRagdoll(ragdollRootBone, 300f, transform.position +
randomDir, 10f);
    }

    private void MatchAllChildTransforms(Transform root, Transform clone)
    {
        foreach (Transform child in root)
        {
            Transform cloneChild = clone.Find(child.name);
            if (cloneChild != null)
            {
                cloneChild.position = child.position;
                cloneChild.rotation = child.rotation;

                MatchAllChildTransforms(child, cloneChild);
            }
        }
    }

    private void ApplyExplosionToRagdoll(Transform root, float explosionForce,
Vector3 explosionPosition, float explosionRange)
    {
        foreach (Transform child in root)
        {
            if (child.TryGetComponent<Rigidbody>(out Rigidbody childRigidbody))
            {
                childRigidbody.AddExplosionForce(explosionForce,
explosionPosition, explosionRange);
            }

            ApplyExplosionToRagdoll(child, explosionForce, explosionPosition,
explosionRange);
        }
    }
```

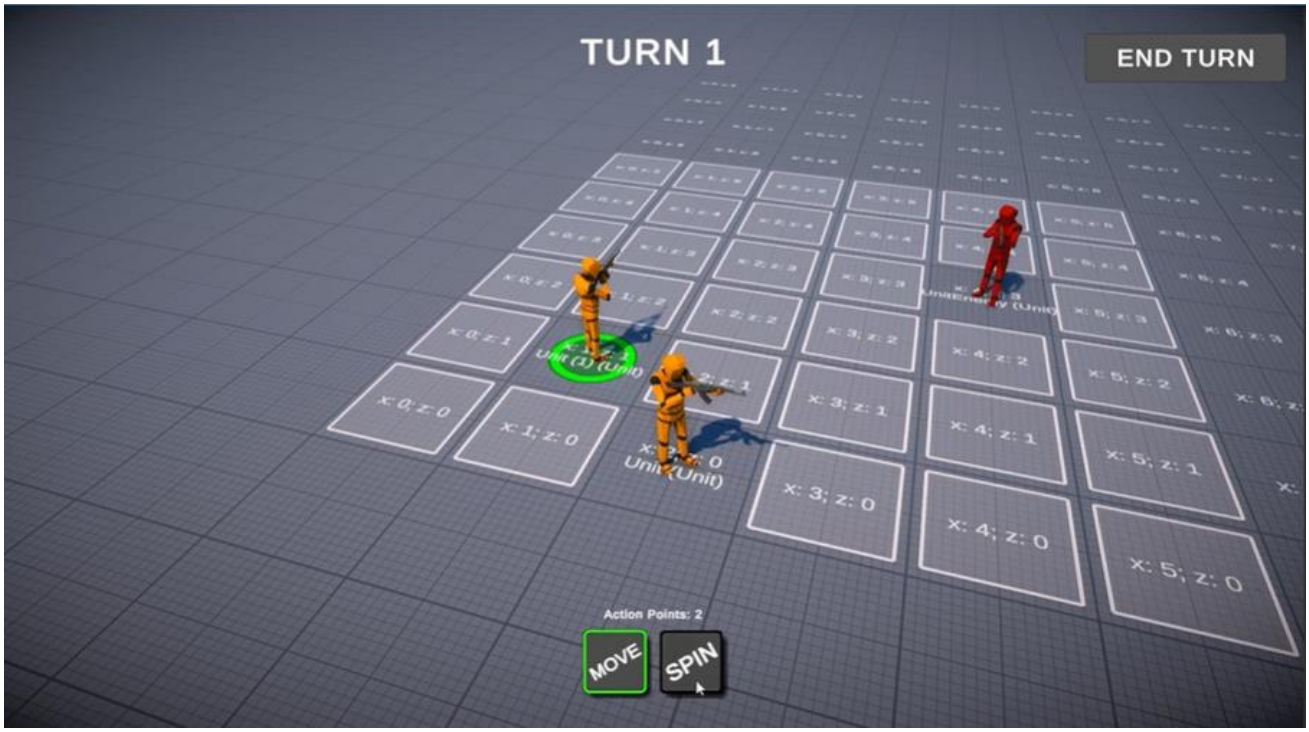
```
}



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

public class UnitRagdollSpawner : MonoBehaviour
{


    [SerializeField] private Transform ragdollPrefab;
    [SerializeField] private Transform originalRootBone;


    private HealthSystem healthSystem;

    private void Awake()
    {
        healthSystem = GetComponent<HealthSystem>();

        healthSystem.OnDead += HealthSystem_OnDead;
    }

    private void HealthSystem_OnDead(object sender, EventArgs e)
    {
        Transform ragdollTransform = Instantiate(ragdollPrefab,
transform.position, transform.rotation);
        UnitRagdoll unitRagdoll = ragdollTransform.GetComponent<UnitRagdoll>();
        unitRagdoll.Setup(originalRootBone);
    }


}




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

public class UnitSelectedVisual : MonoBehaviour
{
```

```
    [SerializeField] private Unit unit;

    private MeshRenderer meshRenderer;

    private void Awake()
    {
        meshRenderer = GetComponent<MeshRenderer>();
    }

    private void Start()
    {
        UnitActionSystem.Instance.OnSelectedUnitChanged +=
UnitActionSystem_OnSelectedUnitChanged;

        UpdateVisual();
    }

    private void UnitActionSystem_OnSelectedUnitChanged(object sender, EventArgs
empty)
    {
        UpdateVisual();
    }

    private void UpdateVisual()
    {
        if (UnitActionSystem.Instance.GetSelectedUnit() == unit)
        {
            meshRenderer.enabled = true;
        }
        else
        {
            meshRenderer.enabled = false;
        }
    }

    private void OnDestroy()
    {
        UnitActionSystem.Instance.OnSelectedUnitChanged -=
UnitActionSystem_OnSelectedUnitChanged;
    }
```

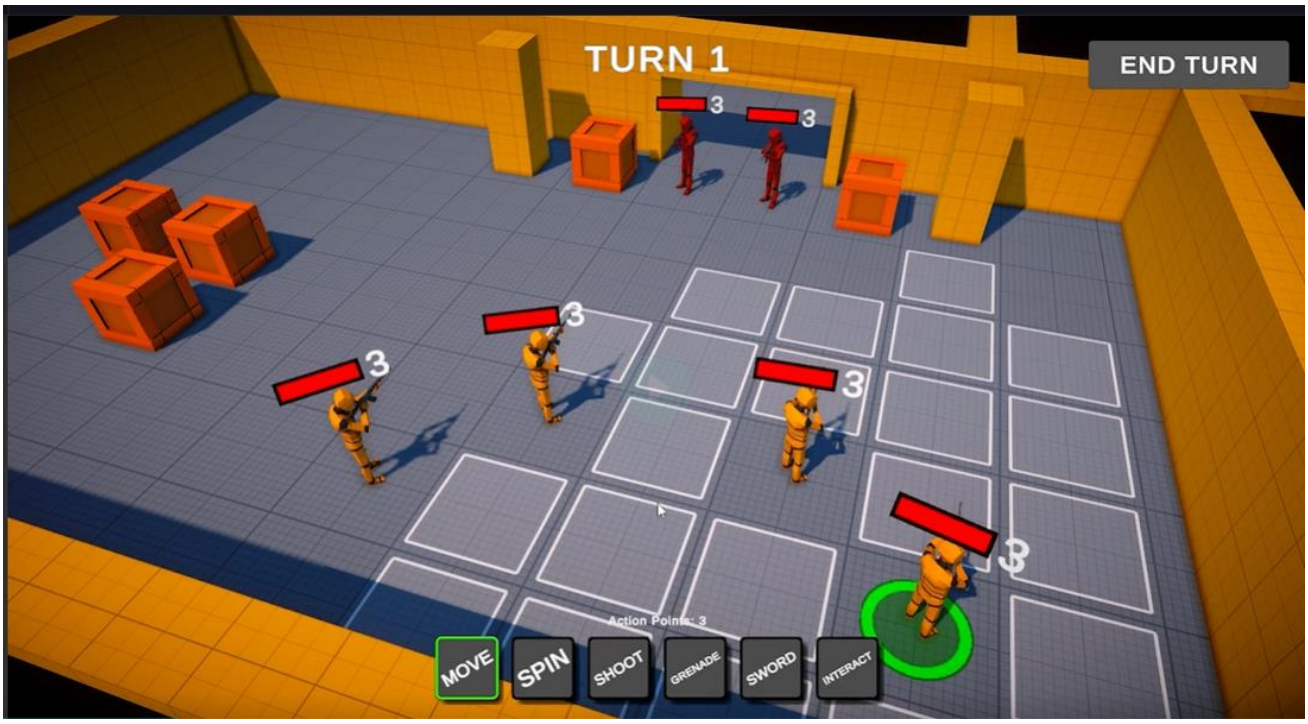
ДОДАТОК Б

(обов’язковий)

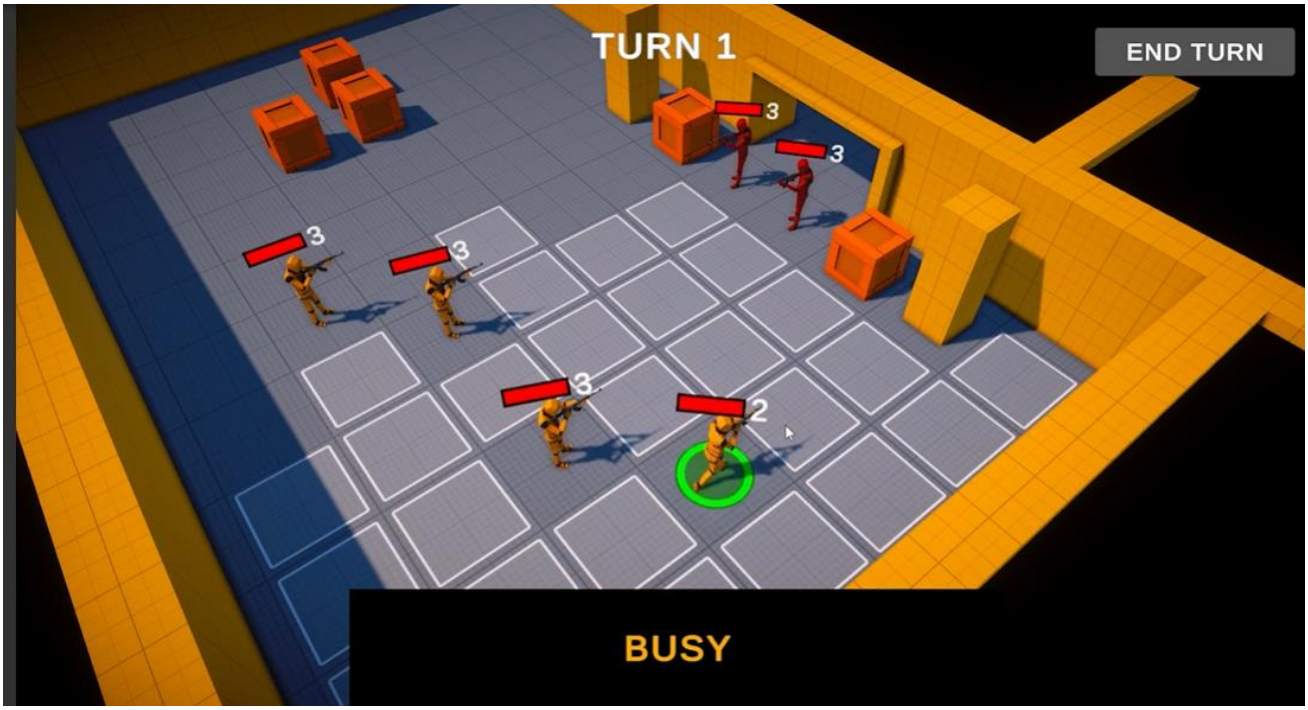
ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ



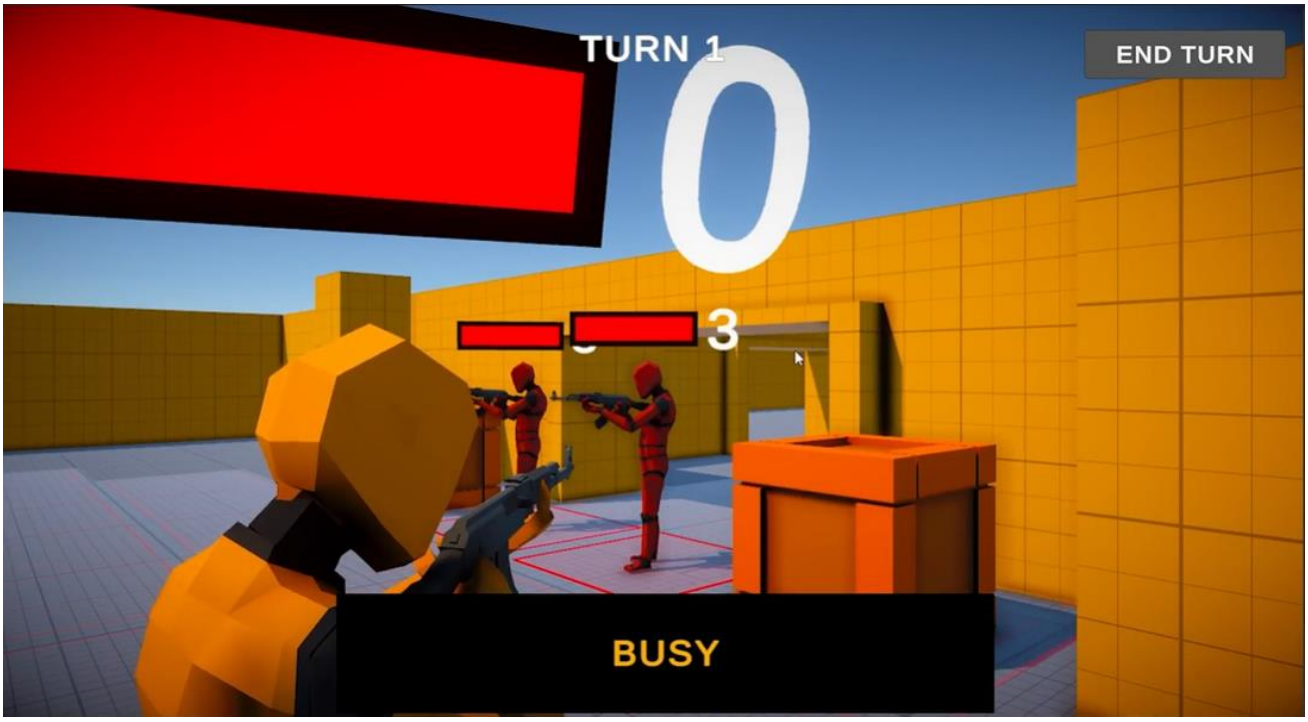
Грід система



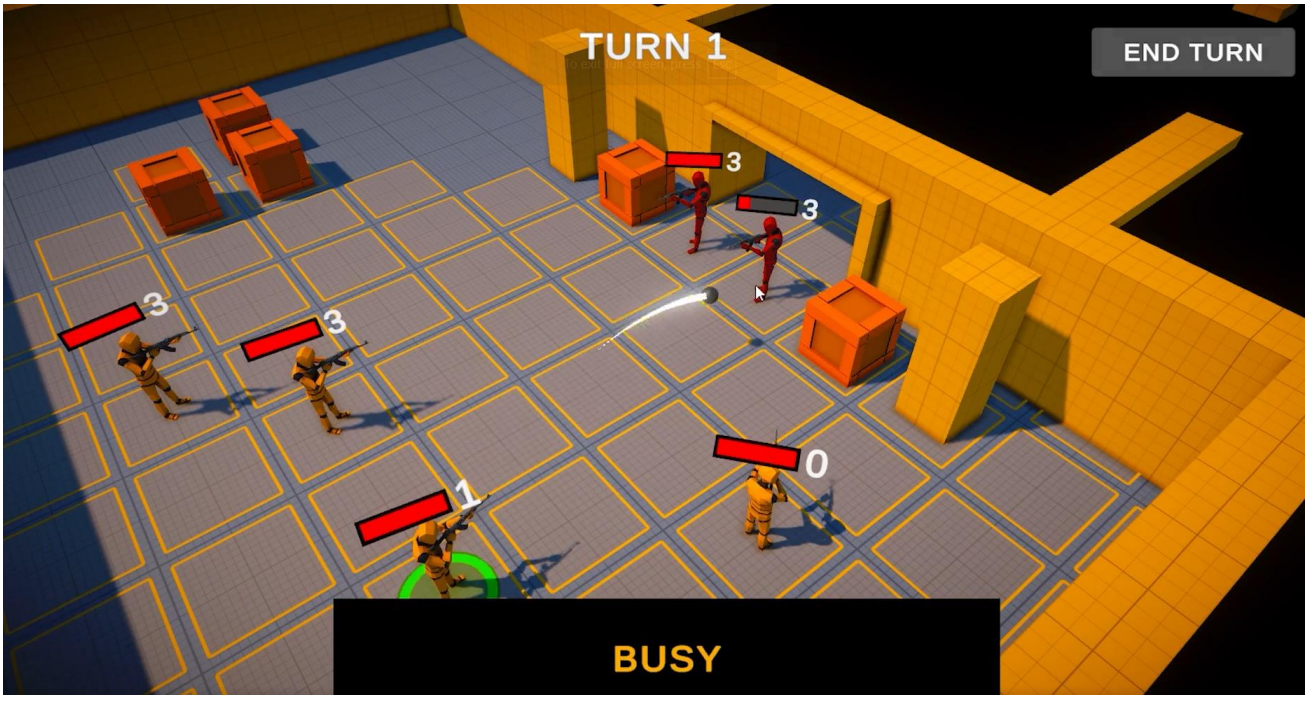
Стартове поле



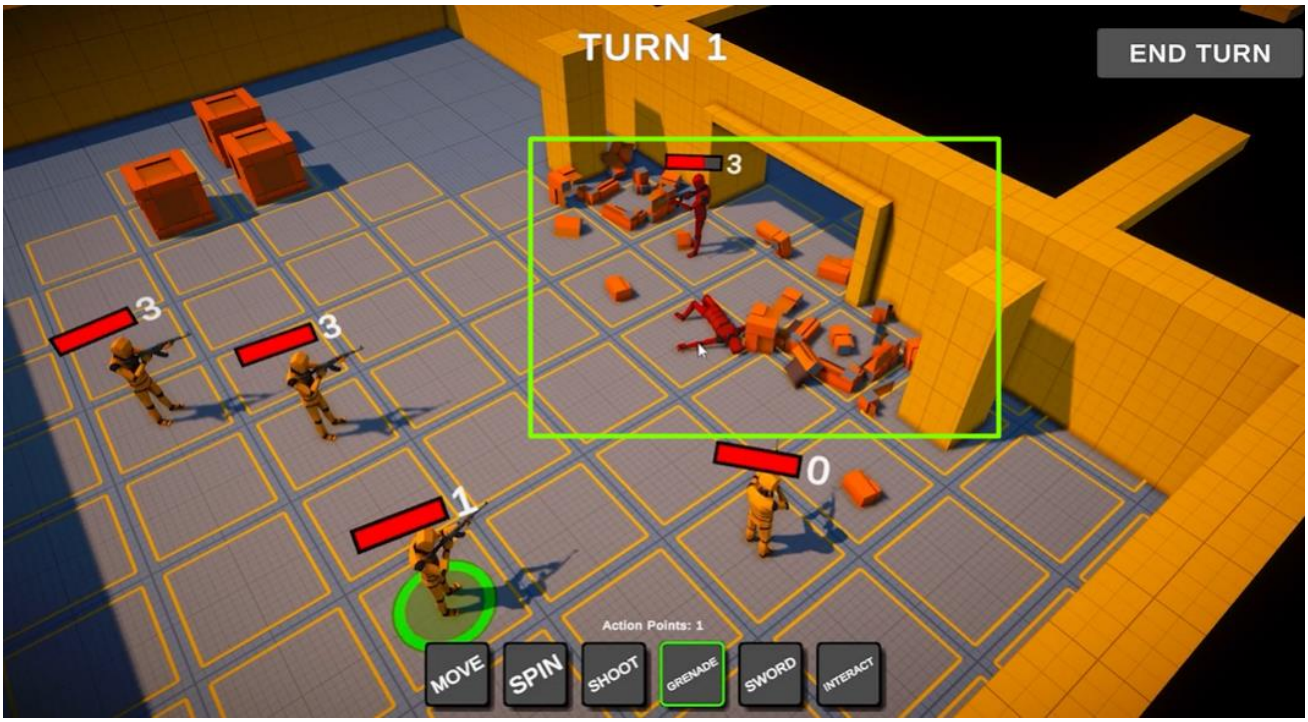
Рух юнітів



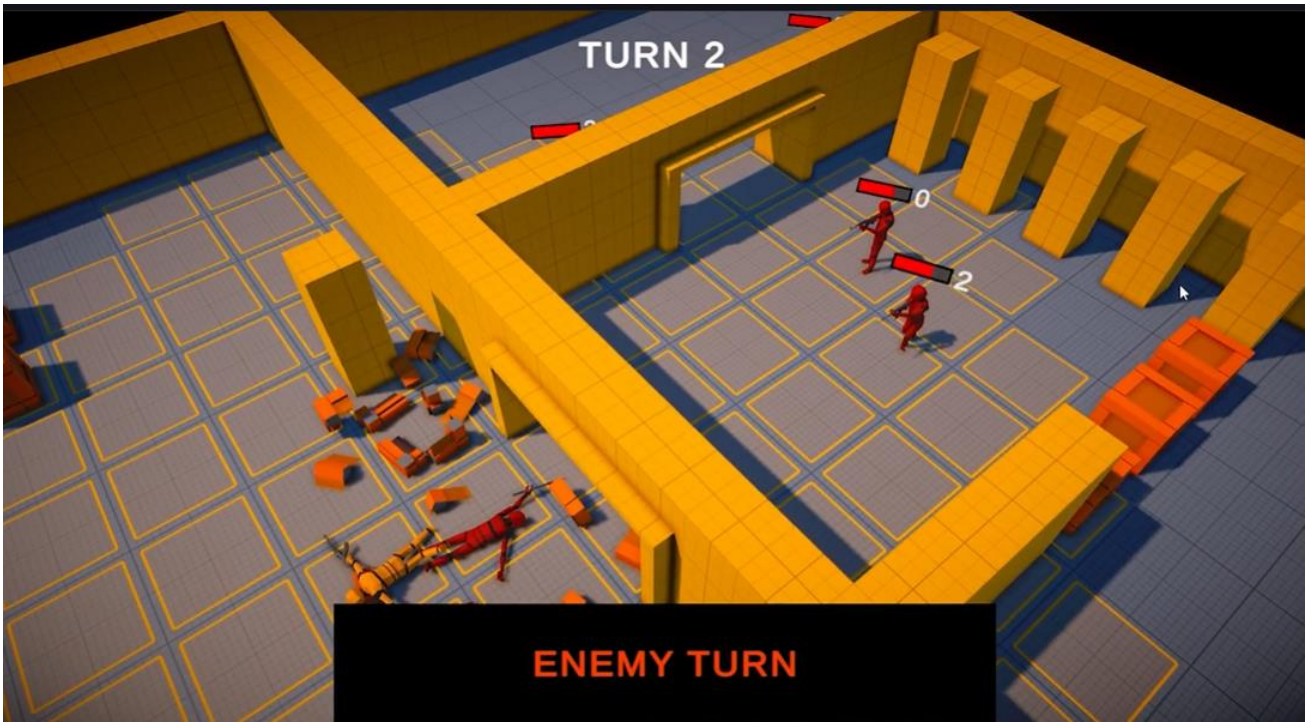
Стрільба



Кидок гранати

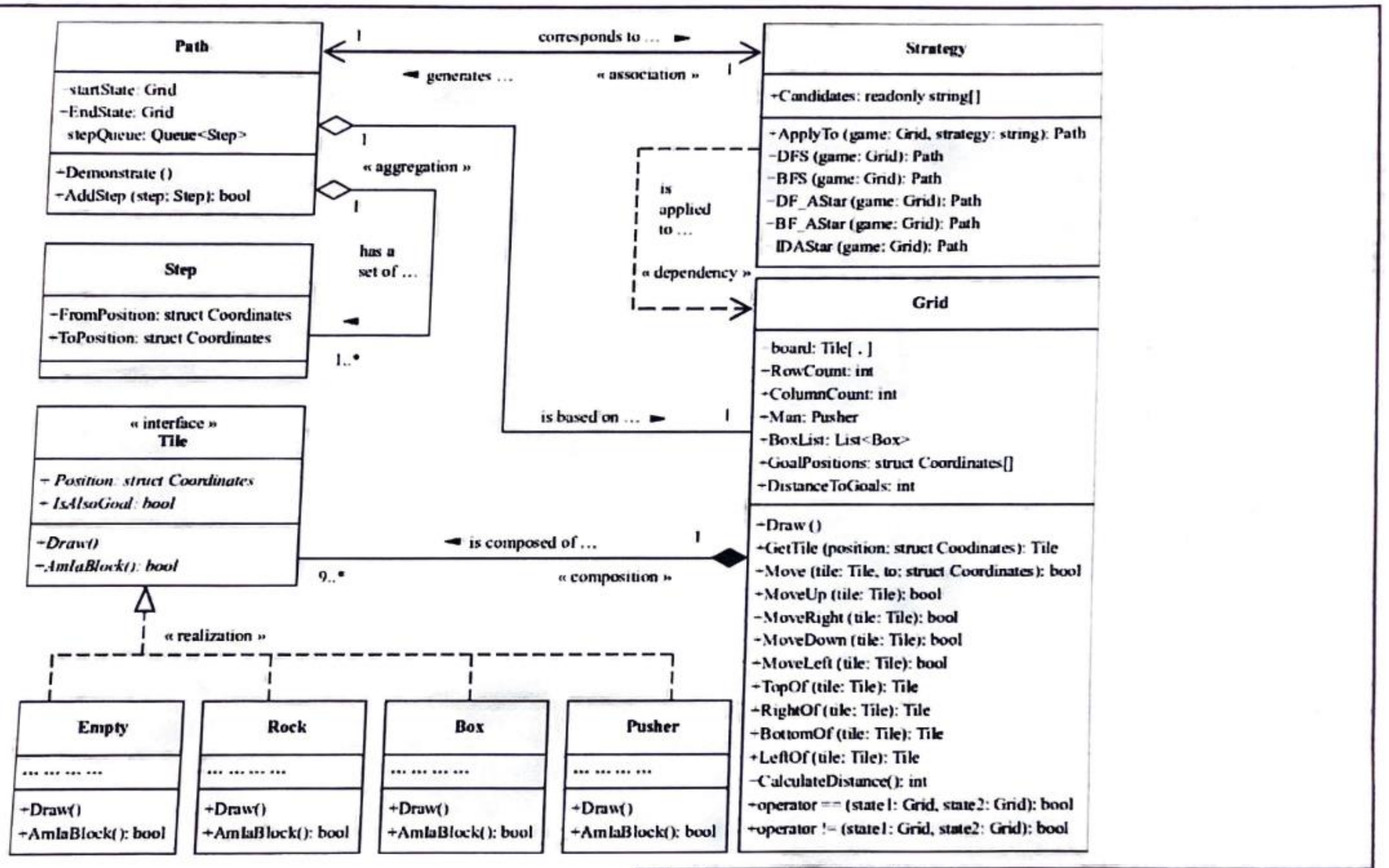


Руйнування, смерть ворогів

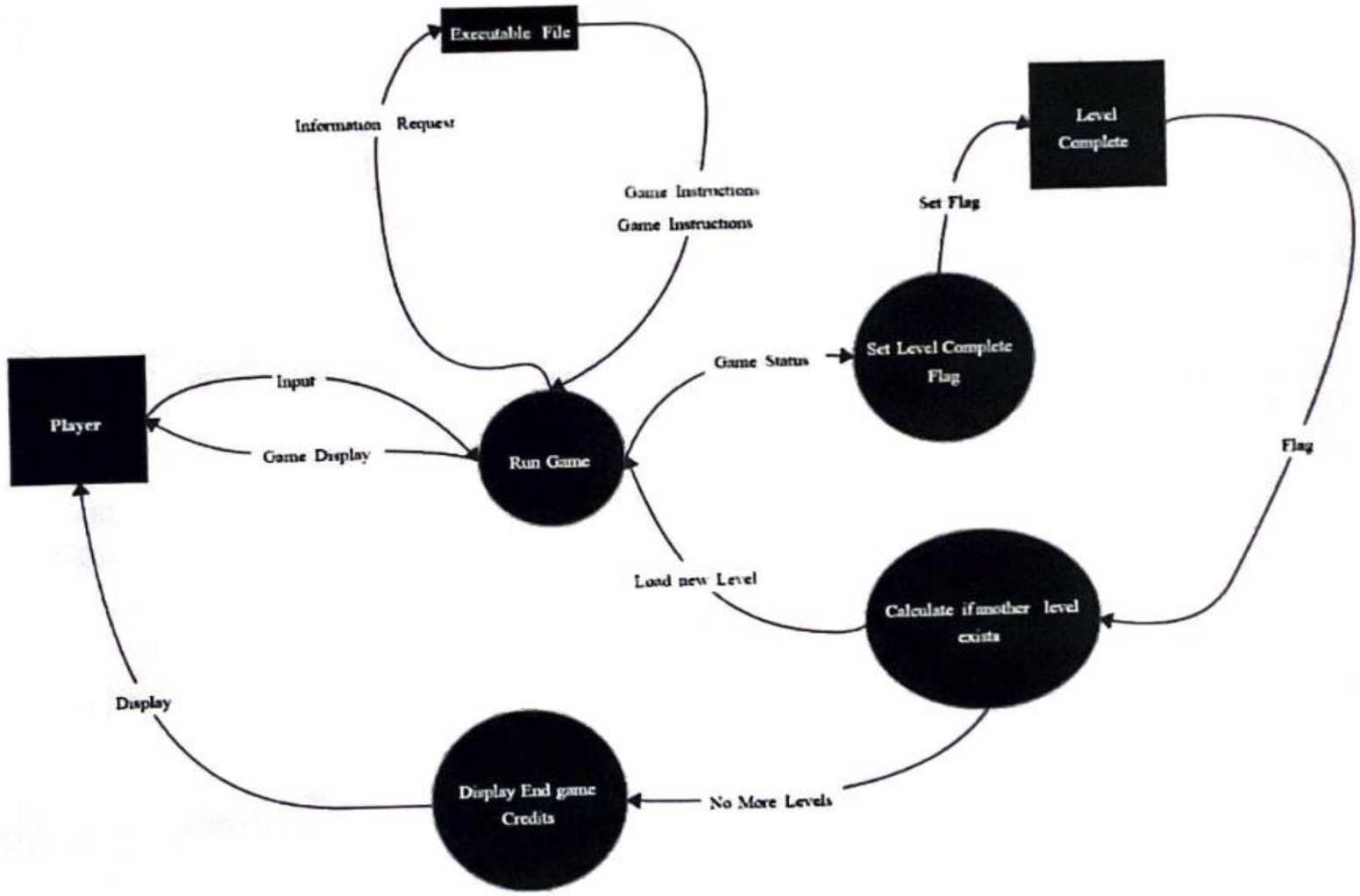


ШІ ворога яки грає проти гравця

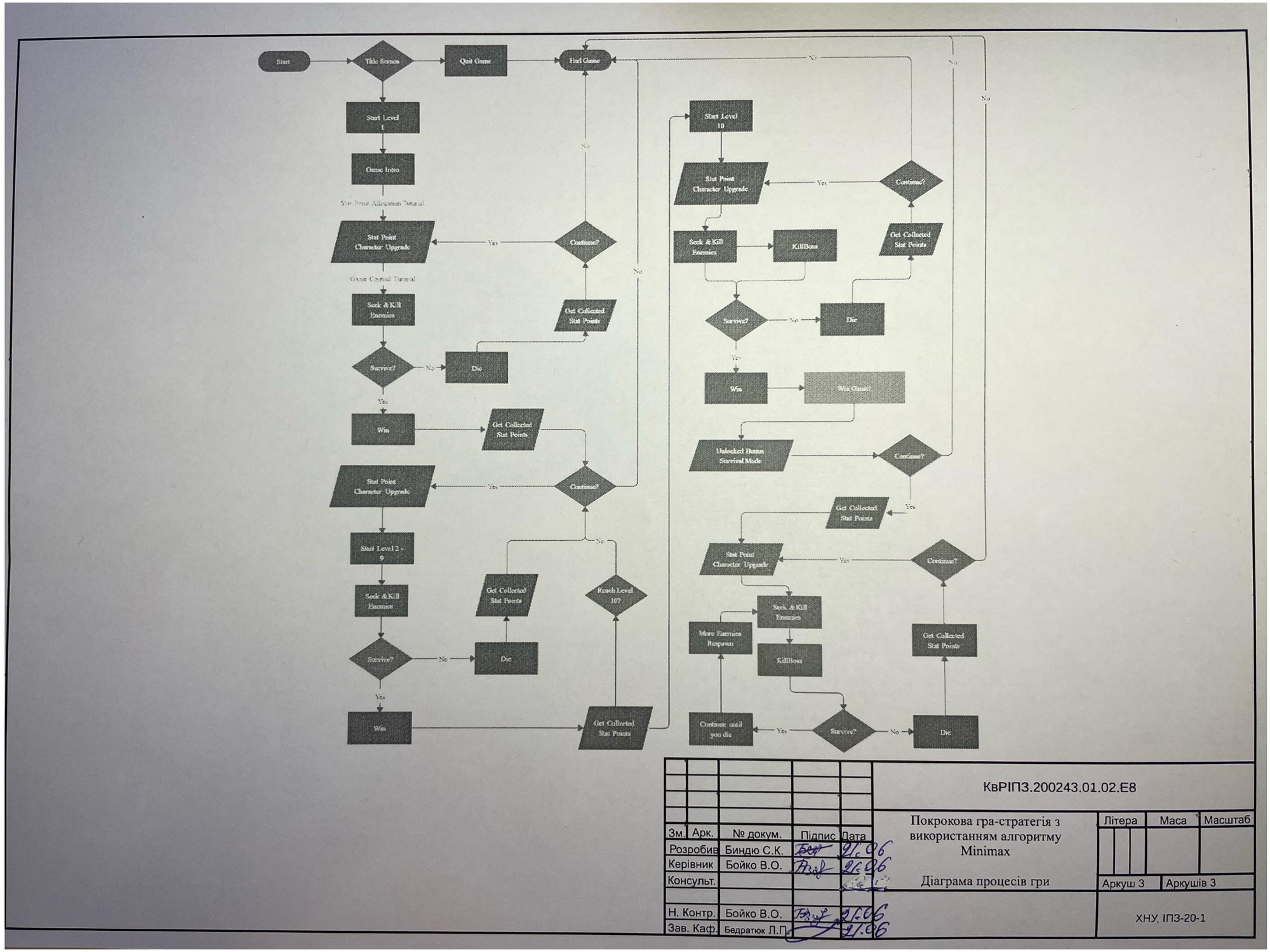
ГРАФІЧНА ЧАСТИНА



					КвРІПЗ.200243.01.02.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Покрокова гра-стратегія з використанням алгоритму Minimax	Літера / Маса / Масштаб
Розробив		Биндю С.К.		21.06		
Керівник		Бойко В.О.		21.06	Діаграма класів	Аркуш 1 / Аркушів 3
Консульт.						
Н. Контр.		Бойко В.О.		21.06		ХНУ, ІПЗ-20-1
Зав. Каф.		Бедратюк Л.П.		21.06		



					КвРІПЗ.200243.01.02.Е8	
Зм.	Арк.	№ докум.	Підпис	Дата	Покрокова гра-стратегія з використанням алгоритму Minimax	Літера / Маса / Масштаб
Розробив		Биндю С.К.		11.06		
Керівник		Бойко В.О.		11.06	DFD-діаграма	Аркуш 2 / Аркушів 3
Консульт.						
Н. Контр.		Бойко В.О.		11.06		ХНУ, ІПЗ-20-1
Зав. Каф.		Бездатюк Л.П.		11.06		



КвРІПЗ.200243.01.02.Е8

Зм.	Арк.	№ докум.	Підпис	Дата
Розробив		Биндю С.К.		21.06
Керівник		Бойко В.О.		21.06
Консульт.				
Н. Контр.		Бойко В.О.		21.06
Зав. Каф.		Бедратюк Л.П.		21.06

Покрокова гра-стратегія з використанням алгоритму Minimax

Діаграма процесів гри

Літера	Маса	Масштаб
Аркуш 3	Аркушів 3	

ХНУ, ІПЗ-20-1

СУПРОВІДНІ ДОКУМЕНТИ

Завідувачу кафедри інженерії програмного забезпечення проф. Бедратюку Л. П.

здобувача вищої освіти
Бидю С. К.

Прізвище, ініціали

факультет ІТ, 4 курс, група ІПЗ-20-1

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності в Хмельницькому національному університеті», згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії Хмельницького національного університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та/або Anti-Plagiarism) і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення текстових збігів у роботах.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

19.06.24
дата

підпис

UNICHECK
by Turnitin

Ім я користувача:
ІПЗ

Дата перевірки:
20.06.2024 23:42:42 EEST

Дата звіту:
21.06.2024 07:17:14 EEST

ID перевірки:
1016379431

Тип перевірки:
Doc vs Internet + Library

ID користувача:
100012953

Назва документа: БКР_Покрокова гра-стратегія з використанням алгоритму Minimax_Биндю_

Кількість сторінок: 118 Кількість слів: 14546 Кількість символів: 127749 Розмір файлу: 7.45 MB ID файлу: 1016188239

9.93%
Схожість

Найбільша схожість: 4.19% з джерелом з Бібліотеки (ID файлу: 1016114512)

4.19% Джерела з Інтернету 667 Сторінка 120

3.93% Джерела з Бібліотеки 184 Сторінка 125

8.75% Цитат

Цитати 58 Сторінка 126

Не знайдено жодних посилань

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи 6

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 3.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 19%**

	Документ		Сумарний збіг по Базі Даних	
ID: 131892 Назва: БКР_Покрокова гра-стратегія з використанням алгоритму Minimax_Биндю_Бойко Додано в БД: 2024-06-21 Автора: Биндю С. К. Керівники: Бойко В. О. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	76441	773	4021 (5%)	54 (7%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

освітнього ступеня «бакалавр»

Бакалавр Биндю Семен Костянтинович

Тема Покрокова гра-стратегія з використанням алгоритму Minimax

Спеціальність 121 Інженерія програмного забезпечення

Обсяг дипломної роботи освітнього ступеня «бакалавр»:

Кількість листів креслень 3 ; кількість сторінок записки 64

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі проведено аналіз основних підходів до розроблення покрокових ігор-стратегій та алгоритмів штучного інтелекту. Розроблено програмне забезпечення для покрокової гри-стратегії з використанням алгоритму Мінімакс. На його основі реалізовано ігровий процес на прикладі створеного ігрового світу з використанням Unity та мови програмування C#. Проведено тестування реалізованого застосунку за отриманими результатами.

2. Висновок про відповідність кваліфікаційної роботи завданню Кваліфікаційна робота освітнього ступеня «бакалавр» відповідає поставленому завданню як у теоретичній, так і в практичній її частині

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи
У вступі обґрунтовано актуальність теми роботи, проаналізовано проблему розроблення покрокових стратегій та обґрунтовано використання алгоритму Мінімакс, сформульовано мету та завдання роботи. У першому розділі проведено аналіз предметної області покрокових ігор-стратегій та алгоритмів штучного інтелекту. Наступні розділи присвячені проєктуванню та реалізації покрокової гри-стратегії з використанням сучасних інструментів розробки ігрових застосунків, таких як ігровий рушій Unity та мова програмування C#.

4. Позитивні сторони роботи Кваліфікайна робота демонструє базове розуміння принципів розроблення покрокових ігор-стратегій та алгоритмів штучного інтелекту. Реалізовано базовий функціонал покрокової гри-стратегії з використанням алгоритму Мінімакс, що включає рух юнітів, систему черговості та елементи бойової системи.

5. Негативні сторони роботи В роботі відсутній аналіз методів оптимізації алгоритму Мінімакс, що могло б покращити продуктивність гри та дозволити використовувати його у більш складних ігрових сценаріях. Також відсутня інформація про тестування продуктивності гри та аналіз її масштабованості. Крім того, можна було б розглянути можливість використання інших методів штучного інтелекту, таких як Монте-Карло, дерево пошуку або навчання з підкріпленням, для порівняння їх ефективності з алгоритмом Мінімакс.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми кваліфікаційної роботи з дотриманням вимог стандартів. У загальному графічне оформлення виконане на достатньому рівні. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

6. Відгук про роботу в цілому Кваліфікаційна робота відповідає основним вимогам, поставленим до робіт такого типу. Структура роботи загалом логічна, проте окремі розділи потребують уточнення та доопрацювання для забезпечення більшої зв'язності. Представлений матеріал охоплює ключові аспекти обраної теми, хоча глибина аналізу та обгрунтування окремих висновків можуть бути посилені. Графічний матеріал доцільно доповнити для більш наочної ілюстрації результатів дослідження.

8.Інші зауваження

9. Оцінка кваліфікаційної роботи Розглянувши позитивні та негативні сторони представленої роботи, можна зробити висновок, що вона заслуговує оцінки «задовільно».

РЕЦЕНЗЕНТ (прізвище. ім'я, по батькові, посада. місце роботи)
Ніченчук Андрій Олександрович, канд. техн. наук, доцент кафедри КІТС ХНУ.

« 21 » 06 2024 р.

(підпис)

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів перевірки роботи, продукованими програмно-технічним засобом (ами), на наявність текстових збігів:

Назва кваліфікаційної роботи: «Покрокова гра-стратегія з використанням алгоритму Minimax»
Автор: Биндю Семен Костянтинович
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Спеціальність: 121 – Інженерія програмного забезпечення
Науковий керівник: Бойко В'ячеслав Олександрович, асистент
Після аналізу звіту/звітів зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована.	
3	Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнуті. Робота може бути допущена до захисту після того, як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системою перевірки на плагіат Unicheck виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, відомість документів), у структурі змісту, назвах розділів/підрозділів, у рамках основних написів, у назвах публікацій переліку джерел посилання;

2) в якості запозичень системою Unicheck було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) запозичення, виявлені в тексті роботи, є фрагментарними.

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 3.0%. Обсяг запозичень, визначений системою Unicheck виявлення збігів ідентичності/схожості, складає 9.93% і адресується до 667 джерел з Інтернету і 184 джерела з бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 21.06.2024

Завідувач кафедри _____ Леонід БЕДРАТЮК

Гарант освітньої програми _____ Леонід БЕДРАТЮК

Керівник кваліфікаційної роботи _____ В'ячеслав БОЙКО