**Міністерство освіти і науки України**
**Національний технічний університет**
**«Дніпровська політехніка»**

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Інформаційних технологій та комп'ютерної інженерії
(повна назва)

# ПОЯСНЮВАЛЬНА ЗАПИСКА

**кваліфікаційна робота ступеня бакалавра**
(бакалавра, спеціаліста, магістра)

**Здобувача** Мовлик Дмитро Дмитрович
(ПІБ)

**академічної групи** 126-21-1
(шифр)

**спеціальності** 126 «Інформаційні системи та технології»
(код і назва спеціальності)

**за освітньо-професійною програмою** «Інформаційні системи та технології»
(офіційна назва)

**на тему** Розробка 2D-RPG ігрового застосунку з процедурною генерацією світу та динамічною системою поведінки ворогів на рушії Unity
(назва за наказом ректора)

| **Керівник** | **Прізвище, ініціали** | **Оцінка за шкалою** | | **Підпис** |
|---|---|---|---|---|
| | | **рейтинговою** | **інституційною** | |
| кваліфікаційної роботи | доц. Соколова Н. О. | | | |
| розділів: | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

| | | | | |
|---|---|---|---|---|
| **Рецензент** | | | | |

| | | | | |
|---|---|---|---|---|
| **Нормоконтролер** | проф. Коротенко Г.М. | | | |

**Дніпро**
**2025**

**ЗАТВЕРДЖЕНО:**
завідувач кафедри

інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ Гнатушенко В.В.
(підпис) (прізвище)

«___» _____ 2025 року

**ЗАВДАННЯ**
**на кваліфікаційну роботу**
**ступеня ___бакалавра___**
(бакалавра, спеціаліста, магістра)

**здобувачу** Мовлик.Д.Д. **академічної групи** 126-21-1
(прізвище та ініціали) (шифр)

**спеціальності** 126 «Інформаційні системи та технології»

**за освітньо-професійною програмою** _____

**на тему** Розробка 2D-RPG ігрового застосунку з процедурною генерацією світу та динамічною системою поведінки ворогів на рушії Unity

затверджену наказом ректора НТУ «Дніпровська політехніка» від 05.05.2025 № 336-с

| Розділ | Зміст | Термін виконання |
|---|---|---|
| Розділ 1 | Аналіз стану області рішення завдання | 03.02.2025 – 30.04.2025 |
| Розділ 2 | Проектування та розробка ігрового застосунку | 03.02.2025 – 03.06.2025 |

**Завдання видано** _____ Соколова Н. О.
(підпис керівника) (прізвище, ініціали)

**Дата видачі** 03.02.2025

**Дата подання до екзаменаційної комісії** _____

**Прийнято до виконання** _____ _____
(підпис студента) (прізвище, ініціали)

# РЕФЕРАТ

Пояснювальна записка: 90 с., 38 рис., 3 табл., 1 додатків, 13 джерел. UNITY, 2D-RPG, C#, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, ШТУЧНИЙ ІНТЕЛЕКТ, NAVMESH, TILEMAP, CINEMACHINE.

***Об'єкт кваліфікаційної роботи:*** 2D-RPG ігровий застосунок з процедурною генерацією та динамічною поведінкою ворогів.

***Предмет кваліфікаційної роботи:*** методи реалізації процедурної генерації світу та поведінки AI-персонажів у 2D-середовищі на рушії Unity.

***Мета роботи:*** розробка ігрового застосунку жанру 2D-RPG, що поєднує механіки процедурної генерації світу та динамічної системи поведінки ворогів для створення реіграбельного ігрового досвіду.

У вступі обґрунтовано актуальність теми, сформульовано мету та завдання роботи, визначено об'єкт та предмет дослідження.

У першому розділі проведено аналіз сучасних 2D-RPG, досліджено підходи до процедурної генерації контенту та розглянуто архітектурні особливості ігрового рушія Unity для реалізації проєктів даного жанру.

У другому розділі наведено практичну реалізацію проєкту: розроблено логіку головного героя та ворожих AI-агентів, реалізовано бойову систему, адаптовано систему навігації NavMesh для 2D, налаштовано камеру Cinemachine та створено ігровий світ за допомогою Tilemap та процедурної генерації.

У висновках узагальнено результати виконаної роботи, підтверджено досягнення поставленої мети та надано пропозиції щодо подальшого розвитку проєкту.

Практичне значення: розроблений прототип є основою для створення 2D-RPG з високим рівнем реграбельності, а реалізовані системи (AI, генерація світу) можуть бути використані в інших ігрових проєктах.

Розроблене програмне забезпечення може бути запроваджено як у комерційні, так і в незалежні (інді) проєкти, орієнтовані на жанр 2D-RPG.

# ABSTRACT

Explanatory note: 88 p., 38 figures, 3 tables, 1 appendices, 13 sources.

UNITY, 2D-RPG, C#, PROCEDURAL GENERATION, ARTIFICIAL INTELLIGENCE, NAVMESH, TILEMAP, CINEMACHINE.

***Object of the qualification work:*** a 2D-RPG game application with procedural generation and dynamic enemy behavior.

***Subject of the qualification work:*** methods for implementing procedural world generation and the behavior of AI characters in a 2D environment on the Unity engine.

***The goal of the work****:* is the development of a 2D-RPG game application that combines procedural world generation mechanics and a dynamic enemy behavior system to create a replayable gaming experience.

The introduction substantiates the relevance of the topic, formulates the goal and objectives of the work, and defines the object and subject of the research.

The first chapter analyzes modern 2D-RPGs, explores approaches to procedural content generation, and examines the architectural features of the Unity game engine for implementing projects of this genre.

The second chapter presents the practical implementation of the project: the logic for the main character and enemy AI agents was developed, the combat system was implemented, the NavMesh navigation system was adapted for 2D, the Cinemachine camera was configured, and the game world was created using Tilemap and procedural generation.

The conclusions summarize the results of the work performed, confirm the achievement of the set goal, and provide proposals for the further development of the project.

Practical significance: the developed prototype serves as a basis for creating 2D-RPGs with a high level of replayability, and the implemented systems (AI, world generation) can be used in other game projects.

The developed software can be implemented in both commercial and independent (indie) projects focused on the 2D-RPG genre.

# ЗМІСТ

РЕФЕРАТ ..... 3
ЗМІСТ ..... 6
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ..... 8
ВСТУП ..... 9
РОЗДІЛ 1. АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАВДАННЯ ..... 11
1.1 Значення жанру 2D-RPG та сучасні технологічні тренди в ігровій індустрії ..... 11
1.1.1 Сучасний стан та перспективи розвитку 2D-RPG ..... 11
1.1.2 Жанр 2d-RPG ..... 11
1.1.3 Актуальність розробки інтелектуальних ворогів та процедурної генерації в сучасних іграх. ..... 12
1.1.4 Проблематика: обмеження класичних підходів. ..... 13
1.2 Огляд сучасних 2D-RPG ігор ..... 13
1.3 Ігрові рушії та їх можливості для створення 2D-RPG ..... 14
1.4 Методи процедурної генерації в іграх ..... 16
1.5 Висновки та аналітичні підсумки огляду ..... 18
РОЗДІЛ 2 ..... 20
ПРОЕКТУВАННЯ ТА РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ ..... 20
2.1 Загальний задум ігрового проєкту ..... 20
2.2 Програмна реалізація ворога: стани, маршрут, взаємодія ..... 21
2.2.1 Використання графічного пакету Skeleton Sprite Pack ..... 21
2.2.2 Імпорт та нарізка спрайтів ..... 22
2.2.3 Створення об’єкта Skeleton і налаштування Sprite Renderer ..... 23
2.2.4 Створення об’єкта Skeleton і налаштування Sprite Renderer ..... 23
2.2.5 Налаштування Animator та побудова FSM-моделі ..... 24
2.2.6 Реалізація механіки виявлення та переслідування гравця ..... 28
2.2.7 Логіка смерті монстра Skeleton ..... 32
2.3 Створення та функціонал головного героя ..... 34
2.3.1 Створення об’єкта Hero та налаштування фізичної модел ..... 34
2.3.2 Підготовка графічних ресурсів та анімацій ..... 37

2.3.3 Підготовка графічних ресурсів та створення анімацій ..... 38
2.3.4 Реалізація системи керування ..... 41
2.3.5 Реалізація життєвого циклу та бойової системи героя ..... 44
2.4 Налаштування камери ..... 46
2.4.1 Cinemachine камера ..... 46
2.5 ..... 49
2.5.1 Початкове налаштування системи навігації AI у Unity ..... 49
2.5.2 Адаптація NavMesh для 2D та налаштування Tilemap ..... 50
2.5.3 Побудова та результати NavMesh ..... 52
2.6. Створення ігрового світу за допомогою Tilemap ..... 56
2.6.1 Побудова ігрового середовища ..... 56
2.7 Процедурна генерація оточення ..... 57
2.7.1 Реалізація алгоритму випадкового розміщення об'єктів ..... 57
2.8 Висновки по другому розділу ..... 59
ВИСНОВКИ ..... 60
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ..... 61
ДОДАТОК А. ВИХІДНИЙ КОД ОСНОВНИХ СКРИПТІВ ..... 63

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

2D - Two-Dimensional – двовимірна графіка або простір.

3D - Three-Dimensional – тривимірна графіка або простір.

AI - Artificial Intelligence – штучний інтелект; система, що імітує поведінку живих істот.

API - Application Programming Interface – програмний інтерфейс застосунку; набір інструментів для взаємодії програм.

C# - мова програмування, що використовується для написання скриптів у Unity.

FSM - Finite State Machine – скінченний автомат; модель, що використовується для проєктування поведінки об'єктів, зокрема в анімаціях.

GitHub - веб-сервіс для хостингу IT-проєктів та спільної розробки.

NPC - Non-Playable Character – неігровий персонаж; будь-який персонаж, яким не керує гравець (у даній роботі – ворог).

RPG - Role-Playing Game – рольова гра.

UI - User Interface – користувацький інтерфейс.

Unity - кросплатформовий ігровий рушій, розроблений компанією Unity Technologies.

Cinemachine - пакет Unity для створення інтелектуальних та динамічних камер.

NavMesh - Navigation Mesh – навігаційна сітка, що використовується AI для прокладання маршрутів.

Tilemap - система Unity для створення ігрових рівнів на основі сітки з тайлів.

# ВСТУП

**Актуальніть роботи.** Ігрова індустрія є однією з динамічних галузей сучасного цифрового середовища. Значну частку в ній посідають інді-проекти, зокрема це ігри жанру 2D-RPG, які залишаються актуальними завдяки відносній простоті реалізації, гнучкості художнього стилю та широкій підтримці з боку геймерської спільноти. Незважаючи на розвиток інструментів, які полегшують створення контенту, проблеми масштабування, повторюваності ігрового світу та примітивної поведінки ворожих об’єктів залишаються актуальними для розробників.

У сучасних 2D-RPG іграх часто застосовується ручне проектування ігрового світу, що обмежує варіативність ігрового процесу та потребує значних трудовитрат. Аналогічно, в більшості випадків логіка поведінки ворогів реалізується за допомогою статичних шаблонів або простої скриптованої логіки, що не забезпечує належної адаптивності та динамічності в умовах мінливого середовища гри. Ці фактори обмежують реіграбельність та глибину взаємодії гравця з ігровим світом.

З метою подолання вказаних обмежень буде доцільним використання процедурної генерації ігрового середовища та побудова гнучкої системи штучного інтелекту для ворогів, здатної змінювати свою поведінку залежно від ситуації. Для реалізації такого підходу обрано рушій Unity, який надає широкий спектр інструментів для створення 2D-ігор, зокрема системи Tilemap, ScriptableObject і тд., за для підтримки анімаційних станів та можливість інтеграції алгоритмів процедурної генерації.

**Метою роботи** є створення 2D-RPG ігрового застосунку з процедурною генерацією ігрового світу та динамічною системою поведінки ворогів на рушії Unity, що забезпечує підвищену реіграбельність та гнучку адаптацію ігрових ситуацій.

**Об'єктом дослідження** є процес розробки програмного забезпечення жанру 2D-RPG із використанням алгоритмів автоматизованого створення

контенту та штучного інтелекту.

**Предметом дослідження** буде розглянуто алгоритми процедурної генерації ігрового світу, моделі поведінки ворогів та способи їх програмної реалізації в середовищі Unity.

**Завдання дослідження:**

1) Провести аналіз сучасних ігор жанру 2D-RPG з метою виявлення типових рішень щодо побудови ігрового світу та AI-систем.

2) Дослідити наявні методи процедурної генерації ігрового світу та визначити придатні для 2D-середовища.

3) Проаналізувати підходи до моделювання поведінки ворогів у 2D-іграх.

4) Оцінити технічні можливості рушія Unity для реалізації зазначених компонентів.

5) Сформувати загальну архітектуру ігрового застосунку з урахуванням процедурної генерації світу та динамічного AI.

# РОЗДІЛ 1. АНАЛІЗ СТАНУ ОБЛАСТІ РІШЕННЯ ЗАВДАННЯ

## **1.1** Значення жанру 2D-RPG та сучасні технологічні тренди в ігровій індустрії

### 1.1.1 Сучасний стан та перспективи розвитку 2D-RPG

У сучасному світі ігрова індустрія постійно еволюціонує, адаптуючись до нових технологій і потреб гравців. Одним із жанрів, що зберігає свою актуальність, є 2D-RPG - рольові ігри з двовимірною графікою, які поєднують глибокий сюжет, розвиток персонажів і тактичні механіки. Водночас розвиток таких технологій, як процедурна генерація та штучний інтелект, відкриває нові можливості для створення унікального й захопливого ігрового досвіду.

### 1.1.2 Жанр 2d-RPG

Жанр 2D-RPG (Role Playing Game) - це жанр ігор, у яких геймер бере на себе роль персонажа або групи персонажів, що розвиваються через виконання завдань, битви з ворогами та дослідження ігрового світу. Основні риси жанру включають двовимірну графіку, бойові системи, а також механіки розвитку, такі як система рівнів, навичок та інвентарю (передача предметів чи просто зберігання). Важливою складовою є сюжетна лінія, яка часто занурює гравця в глибоку історію.

У XXI столітті  жанр 2D-RPG ще є досить популярним, завдяки ностальгії за класичними іграми, такими як “Cave Story”[5] чи “Disgaea: Hour of Darkness”[6], а також доступністю жанру для незалежних розробників. Завдяки простоті реалізації 2D-графіки інді-студії можуть створювати проєкти з багатими механіками та унікальними історіями, не потребуючи великих бюджетів. Цільова аудиторія цього жанру - гравці, які цінують сюжетну глибину, тактичні бої та розвиток персонажа. Або, як часто зазначають самі гравці: “Фарм усьому Фарм”. Тобто гравцям подобається більш одноманітний

тип геймплею, що дозволяє прокачати свого персонажа до максимально рівня. Частіше в цей жанр грають саме шанувальники ретроігор або любителі індіпроєктів, які шукають автентичний ігровий досвід.

### 1.1.3 Актуальність розробки інтелектуальних ворогів та процедурної генерації в сучасних іграх.

На сьогоднішній день, ігрова індустрія стикається з викликами, пов’язаними зі зростаючою складністю ігор і очікуваннями гравців щодо унікального контенту. Процедурна генерація - це технологія автоматизованого створення ігрових елементів, таких як рівні чи світ, предмети чи квести, за допомогою алгоритмів [2]. Завдяки генерації, розробники економлять час і ресурси, одночасно забезпечуючи високу реіграбельність завдяки унікальності кожного проходження [3]. Наприклад, гравець може досліджувати щоразу новий світ, що робить гру більш цікавою.

Штучний інтелект (AI), у свою чергу, вдосконалює поведінку неігрових персонажів (NPC) і ворогів, роблячи їх адаптивними до дій гравця. Інтелектуальні вороги можуть аналізувати тактику гравця, змінювати стратегії чи реагувати на події в грі, що підвищує складність і занурення [1]. Актуальність цих технологій зумовлена потребою в оптимізації розробки, створенні динамічного контенту та відповідності сучасним трендам, де гравці прагнуть непередбачуваності й інтерактивності.

### 1.1.4 Проблематика: обмеження класичних підходів.

Традиційні методи створення ігор мають суттєві недоліки, які ускладнюють розробку сучасних проєктів. Ручне створення рівнів вважається одним із трудомістких процесів, що вимагає значних зусиль від дизайнерів і художників. Такий підхід обмежує варіативність контенту, адже кожен рівень створюється вручну, що також підвищує витрати на розробку. У результаті гравці отримують статичний світ, який швидко втрачає новизну.

Ще однією проблемою є скриптовані вороги, поведінка яких заздалегідь запрограмована. Такі противники діють передбачувано, не адаптуються до дій гравця і не створюють серйозного виклику. Це знижує інтерес до гри, особливо для досвідчених гравців.

Таким чином, інтеграція процедурної генерації та штучного інтелекту в жанр 2D-RPG є не лише актуальною, але й необхідною для подолання обмежень традиційних методів. Завдяки цим технологіям, можна створювати захопливі ігри, що зацікавлять сучасну аудиторію.

## 1.2 Огляд сучасних 2D-RPG ігор

Сучасні ігри 2D-RPG, такі як Hyper Light Drifter [7], Dead Cells [8], Undertale [9] і Hollow Knight, поєднують двовимірну графіку, розвиток персонажів і тактичні бої. Більшість використовують ручне створення рівнів і шаблонні AI, що забезпечує якість, але знижує реіграбельність. Dead Cells вирізняється від названих ігор тим, що в цій грі є процедурна генерація, яка додає варіативність, хоч і потребує балансу. Порівняння (табл. 1.1):

Таблиця 1.1 – Порівняння 2D-RPG ігор

| **Назва гри** | **Рік** | **Тип AI** | **Генерація рівнів** | **Платформа** | **Рушій** |
|---|---|---|---|---|---|
| Dead Cells | 2018 | Шаблонний | Процедурна | PC, консолі, мобільні | Unity |
| Hyper Light Drifter | 2016 | Шаблонний | Ручна | PC, консолі | GameMaker |
| Undertale | 2015 | Шаблонний | Ручна | PC, консолі | GameMaker |
| Hollow Knight | 2017 | Шаблонний (Боси) | Ручна | PC, консолі | Unity |

### 1.3 Ігрові рушії та їх можливості для створення 2D-RPG

Щоб розробляти 2D-RPG, потрібно вибрати рушій, який дасть змогу користувачеві зручно взаємодіяти з 2D-графікою, у якому легко писати логіку гри, та інтегрувати вже написані складні частини, зокрема навчальний робот і процедурна генерація. Арсенал таких рушіїв обмежується Unity, Godot і Unreal Engine. Щоб обрати, який саме потрібен для реалізації проєкту, потрібно проаналізувати п'ять складових критеріїв, а саме: підтримка 2D, простота реалізації AI, підтримка процедурної генерації, продуктивність та активність спільноти. (табл. 1.2)

Таблиця 1.2 – порівняння характеристик рушіїв для 2D-розробки

| **Рушій** | **Простота AI** | **Підтримка 2D** | **Генерація світу** | **Продуктивність** | **Спільнота та ресурси** |
|---|---|---|---|---|---|
| **Unreal** | Behavior Tree, складний для новачків | Орієнтований переважно на 3D, 2D реалізується через Paper2D | Обмежена підтримка у 2D, складна інтеграція | Високі вимоги до ресурсів, перевага у 3D | Висока якість ресурсів, але менше 2D-контенту |
| **Unity** | Вбудований NavMesh, FSM, розширення через ScriptableObject | Повна підтримка Tilemap, Sprite Renderer, 2D-фізики | Плагіни + C# API, можливість реалізації з нуля | Висока стабільність на PC і мобільних платформах | Одна з найбільших спільнот, маркетплейс, регулярні оновлення |
| **Godot** | Простий AI-сценарій через GDScript, відсутній візуальний AI-редактор | Нормальна підтримка, але менш гнучка, ніж у Unity | Підтримка процедурної генерації через GDScript | Висока на ПК, слабша на мобільних пристроях | Активна спільнота, але менш розвинена інфраструктура |

З перегляду таблиці де порівнюються рушії для 2D-розробки, стало зрозуміло, що Unity пропонує найкращий баланс між можливостями, простотою впровадження і наявністю допоміжних інструментів. Саме тому для реалізації ігрового застосунку було обрано рушій Unity.

Однією з переваг Unity є чітка та розширювана архітектура сцени. У типовій 2D-сцені використовуються основні компоненти, які взаємодіють між собою через ієрархію об’єктів. Сцена містить Tilemap для зберігання карти світу, ігрові об’єкти з фізикою, ворогів зі скриптованою поведінкою, а також камеру, яка слідує за гравцем.

Щоб наочно продемонструвати вигляд бойової сцени та взаємодію персонажа з ворогами, буде представлено скриншот моменту бою, який ілюструє основні елементи геймплею (Рис. 1.1):


**Рисунок 1.1** – момент бою в грі Hollow Knight

### 1.4 Методи процедурної генерації в іграх

Для реалізації процедурної генерації рівнів у 2D-RPG необхідно обрати алгоритми, які забезпечують потрібний баланс між варіативністю контенту та контролем над іграбельністю. Найпоширеніші методами вважаються: градієнтний шум (Perlin Noise), клітинні автомати, BSP-дерева та лінійний варіант Marching Cubes (Marching Squares). (табл. 1.3)

Таблиця 1.3 – порівняння алгоритмів процедурної генерації

| **Метод** | **Зона застосування** | **Складність імплементації** | **Переваги** | **Обмеження** |
|---|---|---|---|---|
| **Клітинний автомат** | Моделювання печер, тунельних систем | Низька | Природні, нерегулярні форми | Потребує тонкого налаштування правил |
| **BSP-дерева** | Планування кімнат і коридорів | Середня | Чітка топологія, контроль масштабу | Статична ієрархія, може давати фрагментацію |
| **Perlin Noise** | Створення висотних карт, ландшафтів | Середня | Плавність форм, швидкість | Потребує нормалізації, повторюваність |
| **Marching Squares** | Генерація контурів, ізоліній у 2D | Висока | Гладкі межі, адаптивність до поля | Обчислювальна вартість, складність коду |

Щоб створити цікавий ігровий рівень за допомогою процедурної генерації, потрібно чітко поєднувати кілька алгоритмів. Кожен з них відповідає за окремий етап, а саме: розбиття простору, формування ландшафту, створення кімнат і фінальне налаштування. Схема нижче показує послідовність цих кроків. (схем. 1.1):


Схема 1.1 – 2D-RPG ефективна комбінована схема

### 1.5 Висновки та аналітичні підсумки огляду

Після проведеного огляду, було підтверджено актуальність обраної теми дипломної роботи. Аналіз сучасних 2D-RPG ігор, таких як Dead Cells, Hyper Light Drifter, Undertale та Hollow Knight, показав, що хоча ці проєкти демонструють високий рівень якості, вони часто мають обмеження, такі як: використання шаблонного AI та ручного створення рівнів, що знижує реіграбельність і варіативність контенту.

Розроблюваний застосунок має заповнити ці прогалини шляхом інтеграції процедурної генерації й адаптивного штучного інтелекту ворогів,

що дозволить створювати різноманітний світ, та робити поведінку противників більш гнучкою й цікавою для гравця.

# РОЗДІЛ 2
# ПРОЕКТУВАННЯ ТА РОЗРОБКА ІГРОВОГО ЗАСТОСУНКУ

## 2.1 Загальний задум ігрового проєкту

Метою розробки даної 2D top-down RPG гри на рушії Unity є створення інтерактивного продукту, який поєднує процедурну генерацію світу на початку кожного запуску та динамічну систему бою з ворогами. Гра розрахована на широке коло гравців, які цінують дослідження, адаптивне управління та можливість розробляти власні стратегії подолання небезпек.

Основні функціональні можливості гри включають:

- **Процедурна генерація світу** - при старті кожного сеансу створюється унікальна карта місцевості з різним розташуванням об’єктів.
- **Дослідження ігрового світу:** Гравець вільно переміщається камерою з видом зверху, відкриваючи нові території.
- **Бойова система:** Монстри одного типу реагують на дії гравця, завдають шкоди та отримують урон у процесі бою.
- **Адаптивне управління:** Налаштування керування клавіатури, з варіативністю.
- **Візуальне оформлення:** Стилізована двовимірна графіка з чіткими контурами та атмосферною палітрою кольорів.

Розроблений продукт забезпечує багаторазове повторне проходження завдяки варіативності світу та адаптивності AI, створюючи захопливий ігровий досвід для різних категорій геймерів.

## 2.2 Програмна реалізація ворога: стани, маршрут, взаємодія

### 2.2.1 Використання графічного пакету Skeleton Sprite Pack

Для реалізації візуального образу ворожого персонажа на початковому етапі розробки проєкту було використано готовий графічний набір Asset Skeleton Sprite Pack. Зазначений пакет включає набір анімованих 2D-спрайтів Скелета, що характеризуються високою якістю промальовування, наявністю деталізованих рухів та відповідністю стилістичним вимогам обраного ігрового середовища. (рис. 2.1)

Використання цього ассету [10] дозволило значно прискорити процес створення персонажа, уникаючи витрат часу на ручну відмальовку, а також забезпечити візуальну цілісність ігрової сцени.


Рисунок 2.1 - Skeleton Sprite Pack

### 2.2.2 Імпорт та нарізка спрайтів

Після інсталяції пакета Skeleton Sprite Pack [10] у середовище Unity, його вміст було скопійовано до директорії Assets/Sprites. У цій папці з’явилися файли з текстурами та атласом спрайтів скелетоподібного монстра, готові до подальшої обробки.

Наступним кроком у роботі з графікою було нарізання спрайтів за допомогою вбудованого Sprite Editor. Інструмент було переключено в режим Multiple, після чого за допомогою автоматичної сітки та ручних коригувань виділено кожен кадр анімації. Це дозволило отримати окремі зображення ходьби, атаки та інших станів персонажа.

Для кожного з отриманих фрагментів було встановлено точку опори (Pivot). Відповідно до стилю анімації й подальшої логіки модуля навігації, Pivot було зміщено до центру нижнього краю спрайта. Такий вибір забезпечує коректне вирівнювання персонажа на сцені під час пересування та зіткнень.

Після завершення нарізки та налаштування Pivot усі спрайти були збережені, що дало змогу перейти до побудови анімаційних кліпів та інтеграції Skeleton із системою штучного інтелекту й навігації. (рис. 2.2)

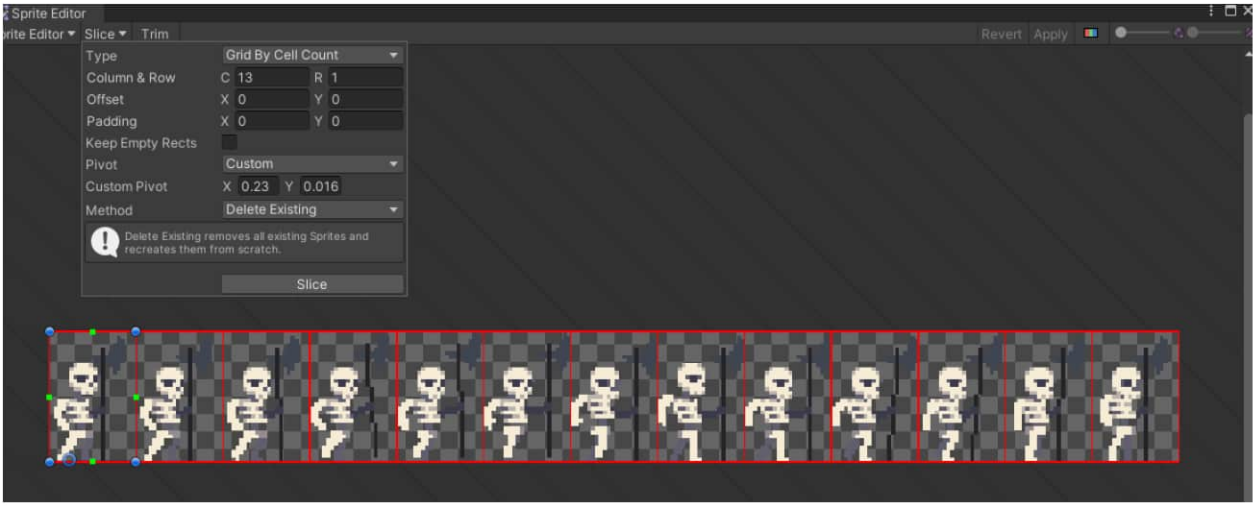



Рисунок 2.2 - Нарізання спрайту і налаштування положення Pivot

### 2.2.3 Створення об’єкта Skeleton і налаштування Sprite Renderer

Як частина підготовки функціоналу персонажа було реалізовано створення базового об’єкта з назвою Skeleton. Для забезпечення його візуального відображення на сцені був доданий дочірній об’єкт SkeletonVisual, який містив компонент Sprite Renderer.

У параметрах Sprite Renderer значення властивості Sprite Sort Point було змінено з Center на Pivot. Це рішення дозволило покращити сортування спрайтів за глибиною, завдяки чому персонаж коректно взаємодіє з іншими об’єктами сцени - без артефактів накладання чи ефекту «проходу крізь текстури». (рис. 2.3)

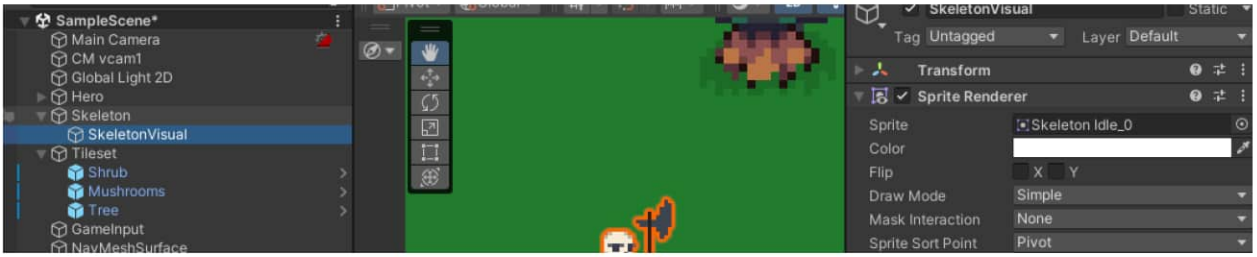



Рисунок 2.3 - Використання дочірнього об’єкта SkeletonVisual для стабільної візуалізації персонажа

### 2.2.4 Створення об’єкта Skeleton і налаштування Sprite Renderer

На етапі реалізації фізичної взаємодії персонажа з навколишнім середовищем до об’єкта Skeleton, після налаштування його візуальної частини, були додані компоненти Box Collider 2D та Rigidbody 2D.

Компонент Box Collider 2D визначає область зіткнень та дозволяє коректно взаємодіяти з іншими об’єктами на сцені, зокрема - уникати проходження крізь перешкоди або застрягання в коллайдерах. Завдяки Rigidbody 2D персонаж починає підпорядковуватись фізичним законам, таким як гравітація та імпульс, що забезпечує його динамічну реакцію на сили та події в грі. (рис. 2.4)

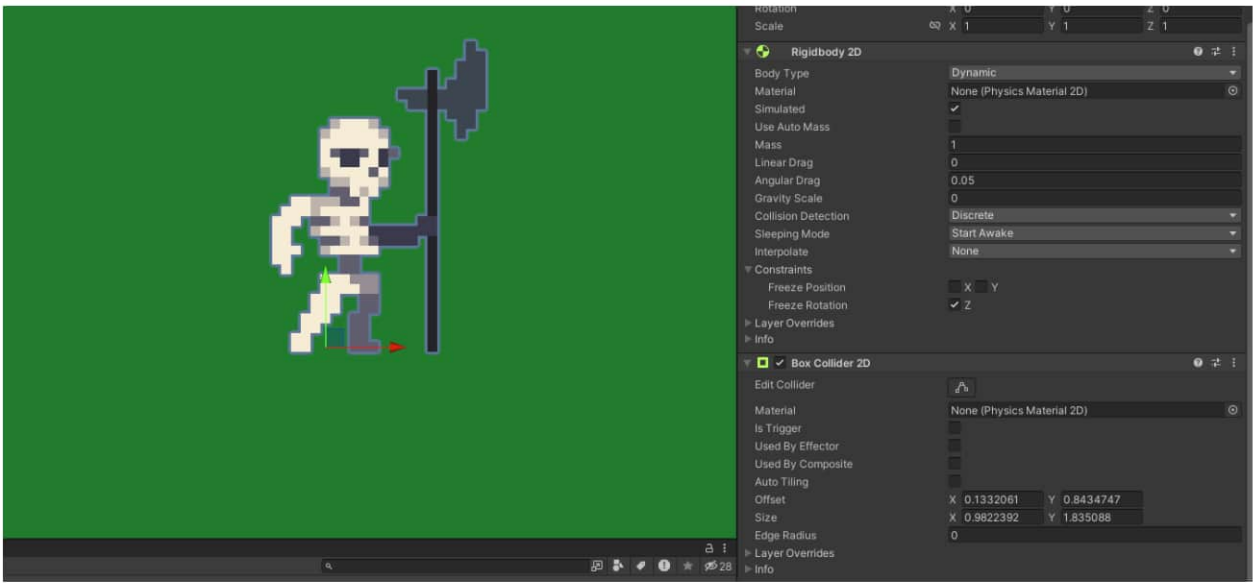



Рисунок 2.4 - Налаштування фізичних компонентів персонажа Skeleton

### 2.2.5 Налаштування Animator та побудова FSM-моделі

Для забезпечення динамічної візуалізації персонажів на сцені, ключовим етапом стало інтегрування системи анімацій. Це передбачало додавання та налаштування компонента Animator до відповідних ігрових об'єктів, що дозволило керувати послідовністю відтворення анімаційних кліпів на основі внутрішньоігрових станів та подій

#### 2.2.5.1 Налаштування анімацій персонажа

Щоб підключити анімації до об'єкта на сцені, до візуальної частини персонажа (об'єкт SkeletonVisual) був доданий компонент Animator. Через інспектор Unity до його поля Controller був призначений Animator Controller, що містить анімаційні стани персонажа. Така прив'язка забезпечує можливість переходів між анімаціями у межах реалізованої FSM-моделі.

У рамках цієї FSM-моделі, вхідна точка Entry пов'язана зі станом Idle, який відповідає за відображення стану спокою. На основі зміни параметра isMoving (що задається з ігрового коду) реалізоване автоматичне перемикання між станами Idle та Walking. Це дозволяє відтворювати відповідні анімації залежно від дій користувача під час гри. (рис. 2.5)

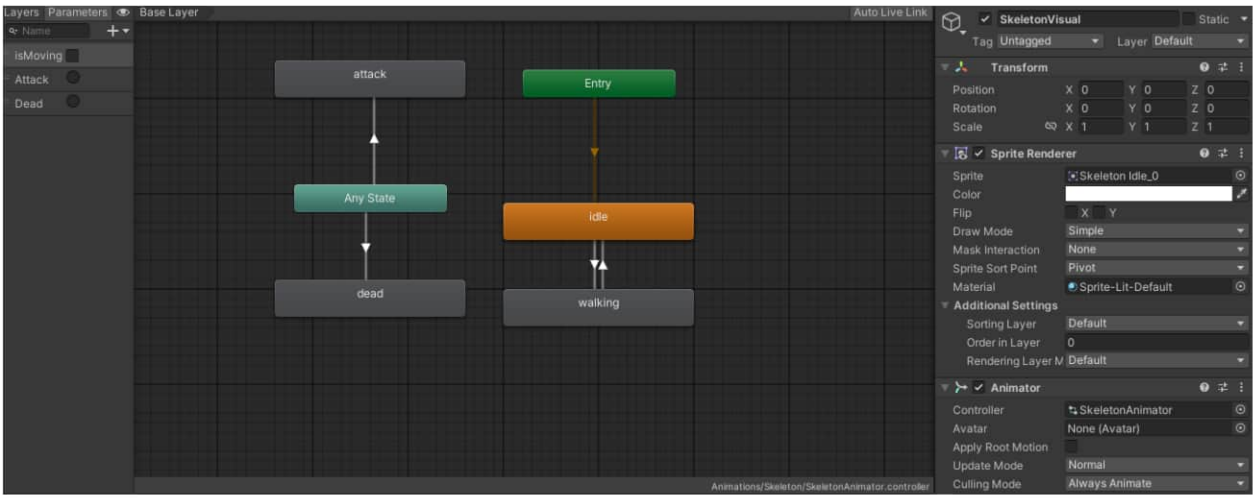



Рисунок 2.5 - Налаштування анімаційних станів персонажа Skeleton

Після того як були підготовлені всі необхідні спрайти для анімацій, вони були використані для створення анімаційних кліпів (наприклад, Idle, Walking, Attack та інших). Ці кліпи були додані в спеціальний компонент Animator Controller, який відповідає за логіку перемикання анімацій залежно від станів персонажа.

На етапі налаштування Animator було реалізовано базову схему станів. Вхідний вузол Entry був логічно пов'язаний зі станом Idle - початковою позою персонажа в спокої. Далі, для реалізації переходів між рухом і нерухомістю, було налаштовано двостороннє перемикання між станами Idle і Walking. Тобто при зміні параметра isMoving анімація переходила від спокою до руху, і навпаки.

Також було додано стани Attack і Dead, перехід до яких відбувався при спрацьовуванні відповідних булевих параметрів Attack і Dead. Ці параметри давали змогу керувати поведінкою анімації на основі внутрішньо-ігрових подій. (рис. 2.6)

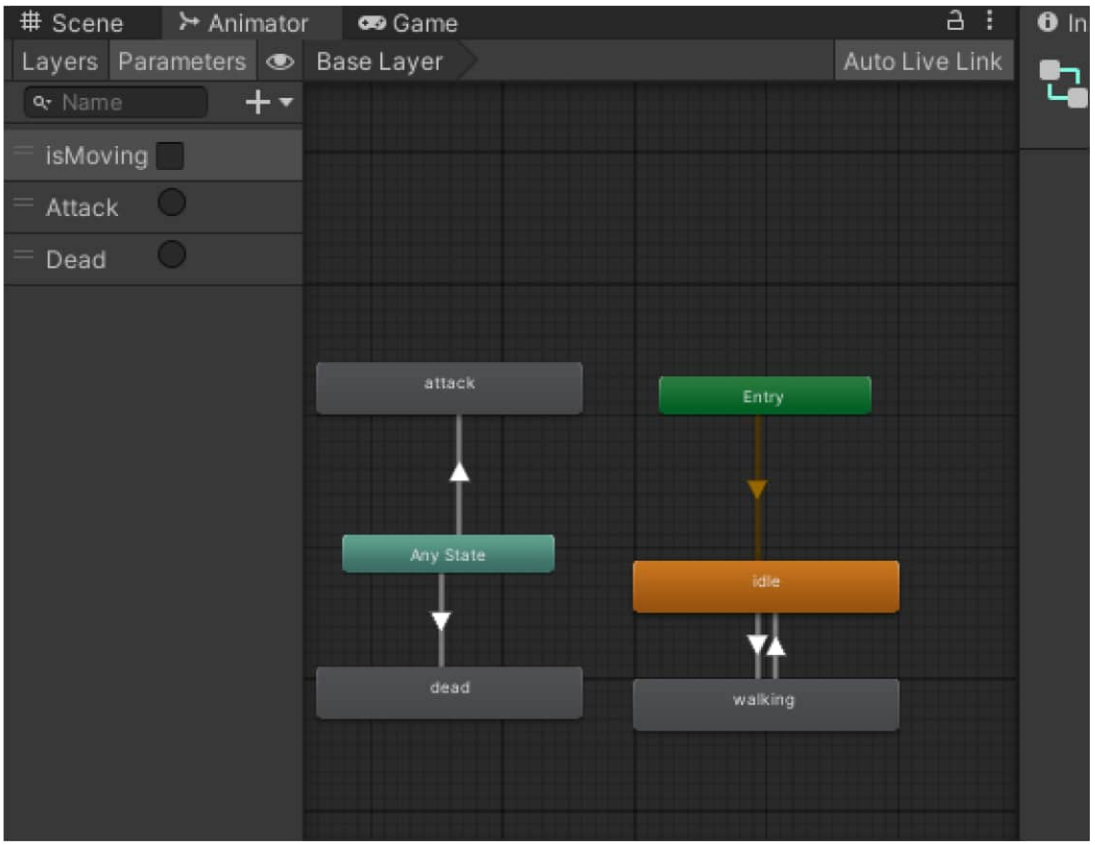



Рисунок 2.6 - Налаштована логіка анімацій

### 2.2.5.2 Реалізація життєвого циклу ворога

Далі для монстра Skeleton було додано скрипт EnemyHealth, який прикріплено безпосередньо до основного об’єкта персонажа. Цей скрипт відповідає за управління здоров’ям ворога, обробку отриманих ушкоджень та станом смерті. Важливою частиною реалізації є застосування патерну State Management, що дозволяє відслідковувати поточний стан ворога (живий чи мертвий) і відповідно реагувати на зміни. Метод TakeDamage контролює зменшення здоров’я, а також виконує фізичне відштовхування ворога після отримання удару. Програмна реалізація цієї логіки наведена в Додатку А. (рис. 2.7)

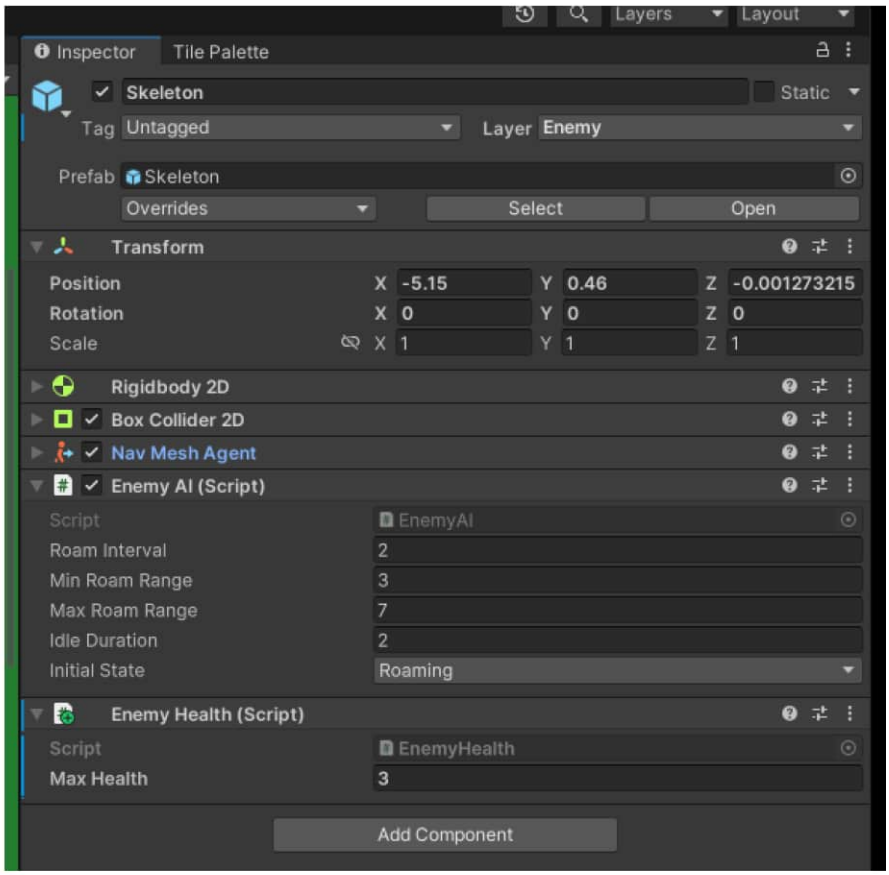



Рисунок 2.7 - скрипт життя Skeleton

Для коректної візуалізації смерті ворога в аніматорі було додано тригер Dead, який активується при зменшенні здоров’я ворога до нуля або нижче. Цей тригер зв’язаний із переходом до стану анімації загибелі в Animator Controller, де використовується патерн State Machine, що дозволяє чітко розмежувати логіку життєвих станів ворога - живий, мертвий. Завдяки цьому при активації тригера відбувається плавний перехід анімації на смерть, що покращує візуальну якість та послідовність ігрового процесу.

Після запуску анімації смерті скрипт EnemyHealth забезпечує повне завершення життєвого циклу ворога: деактивує колайдер для уникнення подальших взаємодій, припиняє рух через зміну властивостей Rigidbody, та виконує затримку перед знищенням об’єкта, враховуючи тривалість анімації. Такий підхід дозволив гармонійно поєднати анімаційні та ігрові механіки, підвищуючи загальну якість взаємодії з ворогом.

Відповідні зміни у контролері анімацій та доповнення у коді реалізовані за допомогою патерну State Management, що забезпечує керованість станами та реакціями ворога на події ігрового світу. Детальна програмна реалізація цих механізмів наведена у Додатку А. (рис.2.8)

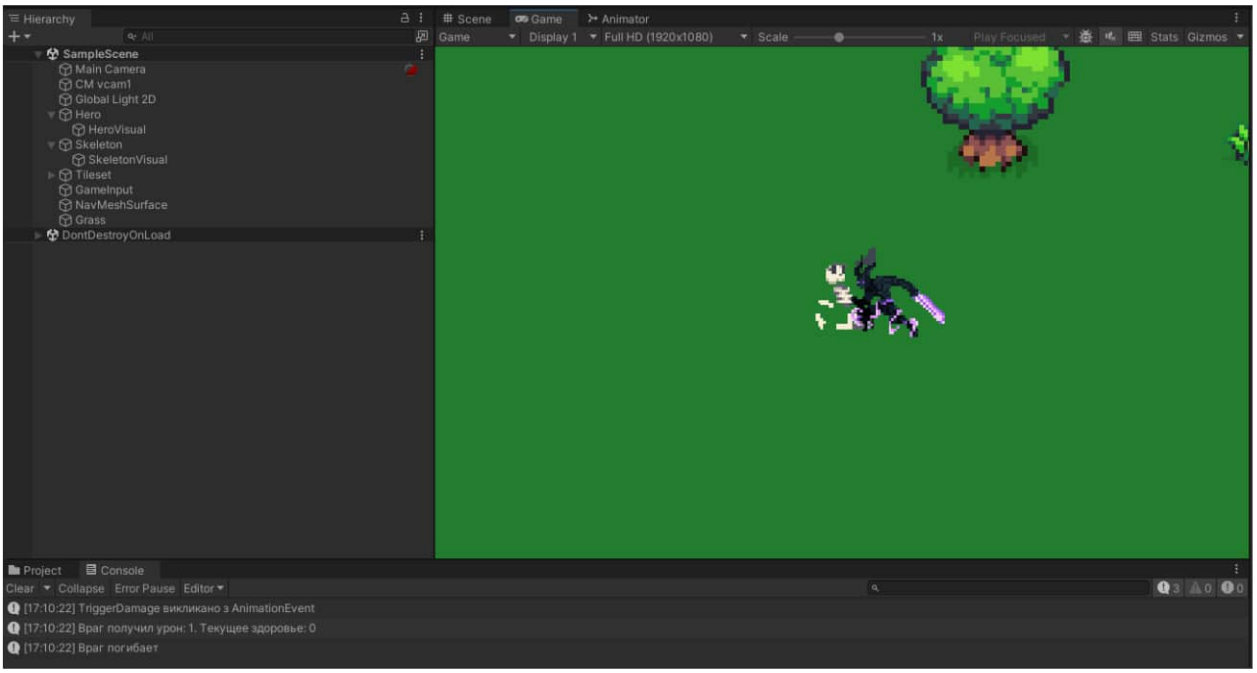



Рисунок 2.8 - Результат спрацьовування анімації смерті ворога та його знищення

### 2.2.6 Реалізація механіки виявлення та переслідування гравця

Щоб забезпечити можливості ворогів реагувати на присутність гравця в ігровому світі, було впроваджено механізм виявлення на основі тригерних коллайдерів. Цей підхід дозволяє ідентифікувати гравця при його входженні в певну зону навколо ворога, ініціюючи зміну поведінки останнього (наприклад, перехід до стану переслідування).

Для цього було додано компонент Circle Collider 2D до об’єкта Skeleton (монстр). Це забезпечує кругову форму зони виявлення, що є оптимальною для визначення радіусу агру. Ключовою особливістю цього коллайдера є активація опції Is Trigger в його налаштуваннях, що дозволяє йому виступати як сенсор, який фіксує перетин меж іншими коллайдерами без фізичного зіткнення.

Програмна логіка обробки входу/виходу з тригерної зони представлена у Додатку А. (рис. 2.9)

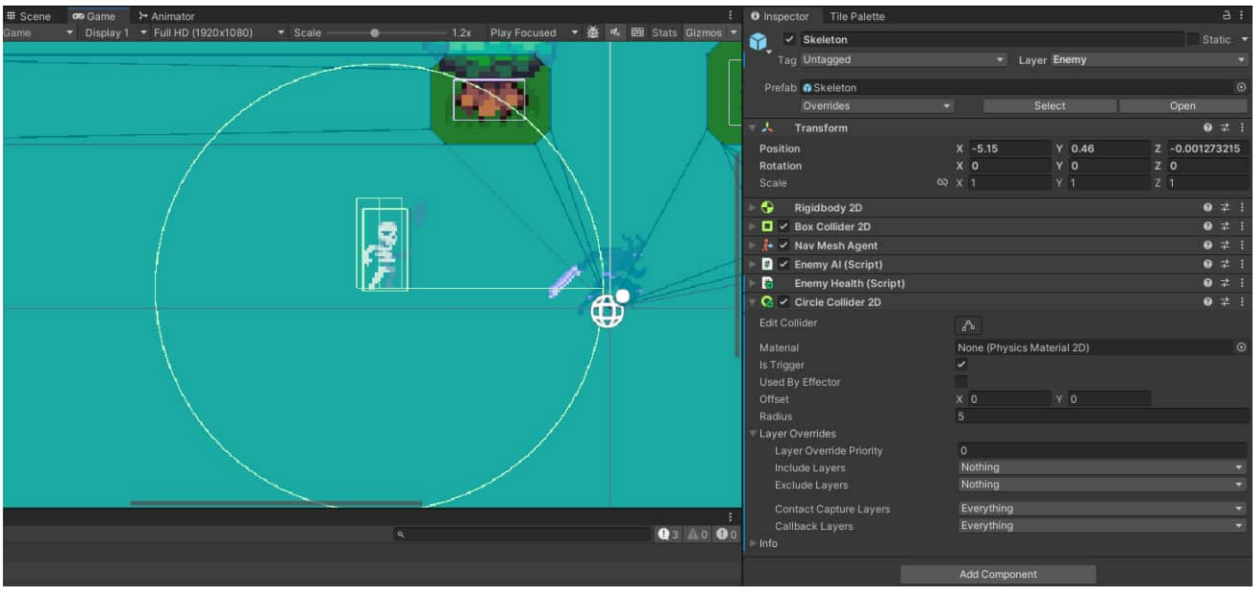


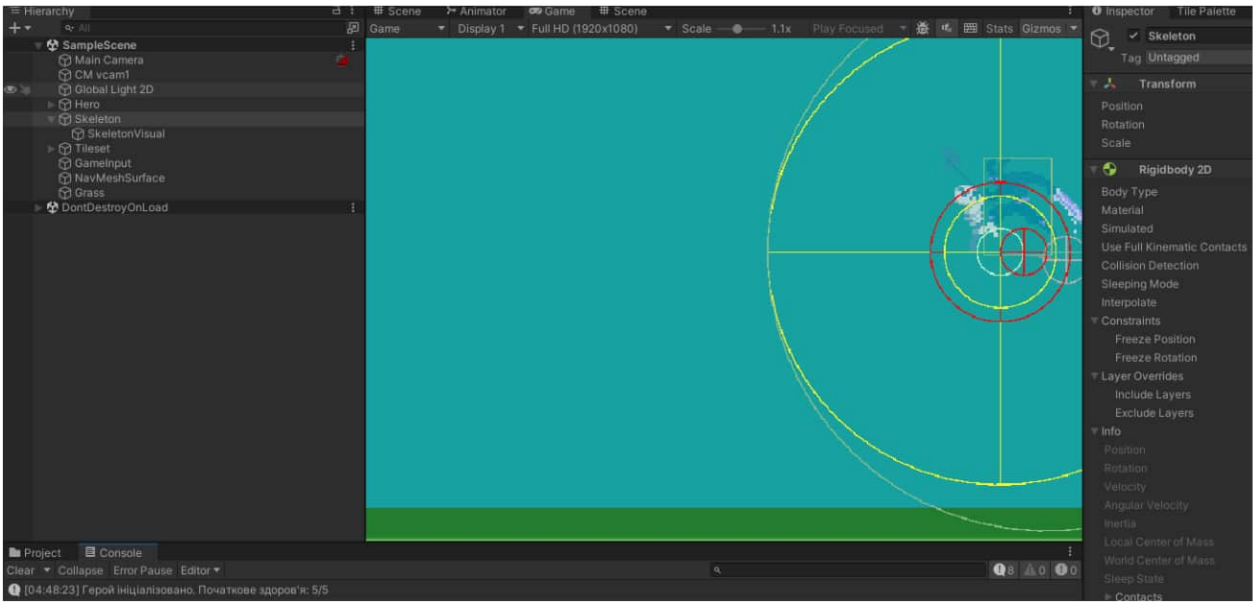

Рисунок 2.9 – Радіус агру противника

### 2.2.6.1 Виявлення агру та початок переслідування

На цьому етапі було забезпечено, щоб ворог реагував на появу гравця в його зоні видимості. Коли герой потрапляє в поле виявлення ворога, останній змінює свою поведінку, переходячи від патрулювання до переслідування гравця.

Коли коллайдер гравця перетинає межі цього тригерного коллайдера ворога, у скрипті EnemyAI спрацьовують методи OnTriggerEnter2D та OnTriggerExit2D. Це дозволяє програмно відслідковувати появу гравця в зоні виявлення та його подальший вихід, відповідно змінюючи внутрішній стан ворожого штучного інтелекту. При виявленні гравця, EnemyAI переводить ворога у стан Chasing, спрямовуючи його рух у бік гравця за допомогою компонента Nav Mesh Agent. Програмний код, що описує логіку виявлення та переходу в стан переслідування, наведено у Додатку А.

Таким чином, створена система забезпечує автоматичне виявлення гравця та ініціювання фази переслідування, що є фундаментальною складовою інтерактивної поведінки ворогів. (рис. 2.10)




### 2.2.6.2 Створення бойової логіки ворога

Для забезпечення можливості ворогів ефективно взаємодіяти з гравцем шляхом нанесення урону, було розроблено та інтегровано спеціалізований скрипт EnemyCombat. Цей компонент відповідає за визначення умов для атаки, її ініціацію, контроль часу перезарядки між атаками (кулдауну), а також за логіку фактичного нанесення шкоди гравцеві.

Скрипт EnemyCombat прикріплений до основного ігрового об'єкта ворога, що дозволяє йому отримати доступ до необхідних компонентів, таких як Animator для запуску анімацій атаки та SpriteRenderer для коректного позиціонування зони ураження. Ключові параметри бойової механіки ворога, такі як Attack Range, Attack Cooldown та Attack Damage визначені у його структурі. Програмна структура цього скрипта представлена у Додатку А. (рис. 2.11)

Центральним елементом побудови нанесення шкоди є публічний метод DealDamage(), який викликається за допомогою Animation Event у точно визначений кадр анімації атаки ворога. Це забезпечує візуальну синхронізацію моменту удару з фактичним нанесенням шкоди. Для визначення цілей використовується функція Physics2D.OverlapCircleAll(), яка формує

віртуальну кругову зону ураження з певним радіусом (Attack Zone Radius) та адаптивним зміщенням (Attack Zone Offset) відносно позиції ворога, враховуючи його напрямок (контролюється через SpriteRenderer.flipX). Програмний код логіки атаки та виявлення цілі наведено у Додатку А.

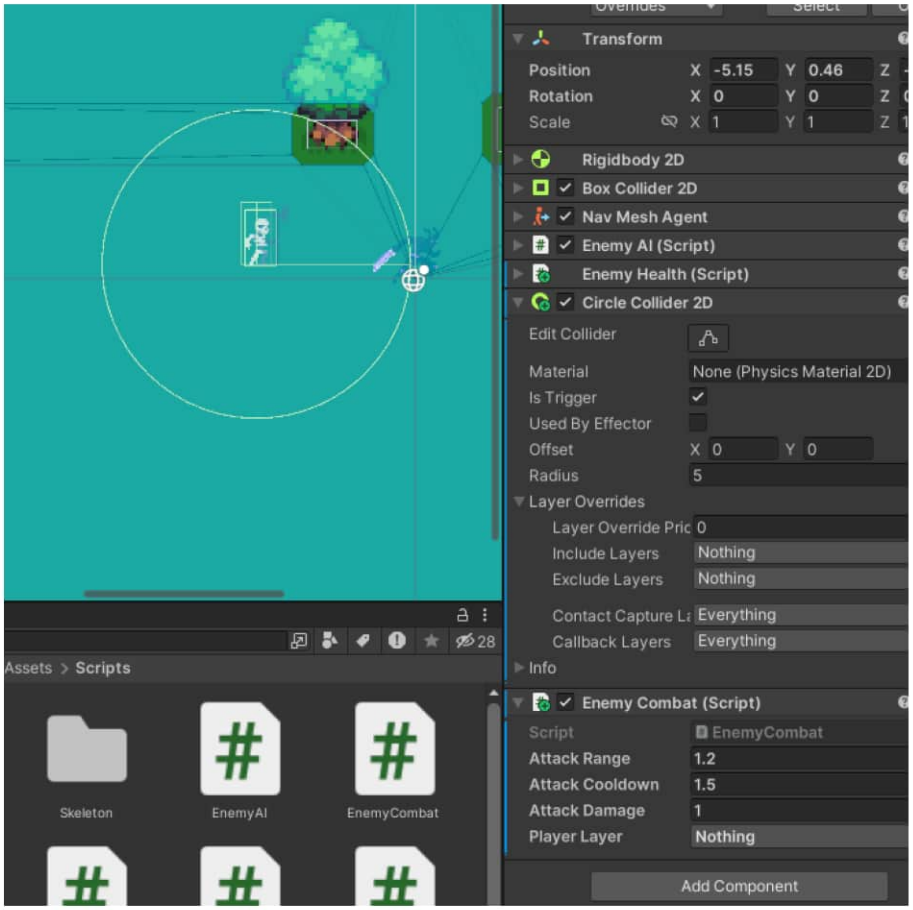



Рисунок 2.11 – Налаштування компонентів та параметрів скрипта EnemyCombat для об'єкта Skeleton

### 2.2.6.3 Синхронізація атаки ворога за допомогою Animation Event

Для забезпечення синхронізації візуального відображення атаки ворога з логікою нанесення урону, використовувався механізм Animation Event. Цей інструмент дозволив прив'язати виклик функції до конкретного кадру анімації. (рис. 2.12)

При налаштуванні анімації атаки ворога (стан Attack у контролері анімацій) був обраний конкретний кадр замаху, у якому було додано Animation Event. Для цієї події призначалася функція DealDamage(), реалізована у скрипті EnemyCombat. Це забезпечило активацію логіки нанесення урону саме в момент візуального удару, покращуючи реалізм взаємодії. Програмна реалізація методу DealDamage() детально описана у Додатку А.

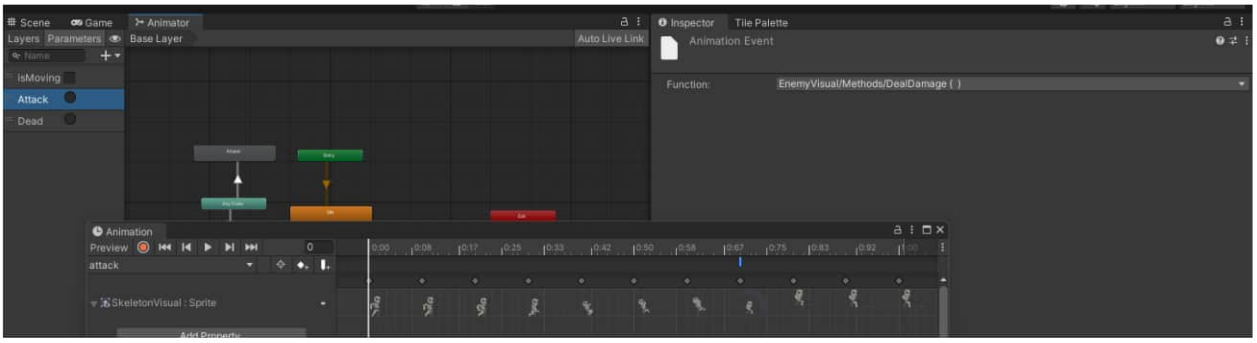



Рисунок 2.12 - Налаштування Animation Event для анімації атаки ворога

### 2.2.7 Логіка смерті монстра Skeleton

Завершальним кроком у розробці поведінки ворога Skeleton було впровадження повноцінної логіки його смерті. Цей механізм забезпечує коректну візуалізацію загибелі ворога, припинення його активних дій та контрольоване видалення зі сцени.

В основі для управління станом здоров'я ворога є скрипт EnemyHealth, прикріплений до основного ігрового об'єкта "Skeleton". Цей скрипт відстежує поточний рівень здоров'я ворога, обробляє отриманий урон і, при зменшенні здоров'я до нуля або нижче, ініціює послідовність дій, що відповідають його

загибелі. Програмна логіка цього скрипта та управління його станами за допомогою патерну State Management наведена у Додатку А.

Для візуального відображення смерті в Animator Controller монстра Skeleton було додано спеціальний тригер Dead. Цей тригер активується програмно зі скрипта EnemyHealth у момент, коли ворог отримує смертельний урон. Тригер Dead запускає перехід до відповідного стану анімації загибелі, яка також має назву Dead у контролері аніматора. Це забезпечує плавний та візуально узгоджений перехід до анімації смерті, що покращує загальну якість ігрового досвіду. (рис. 2.13)

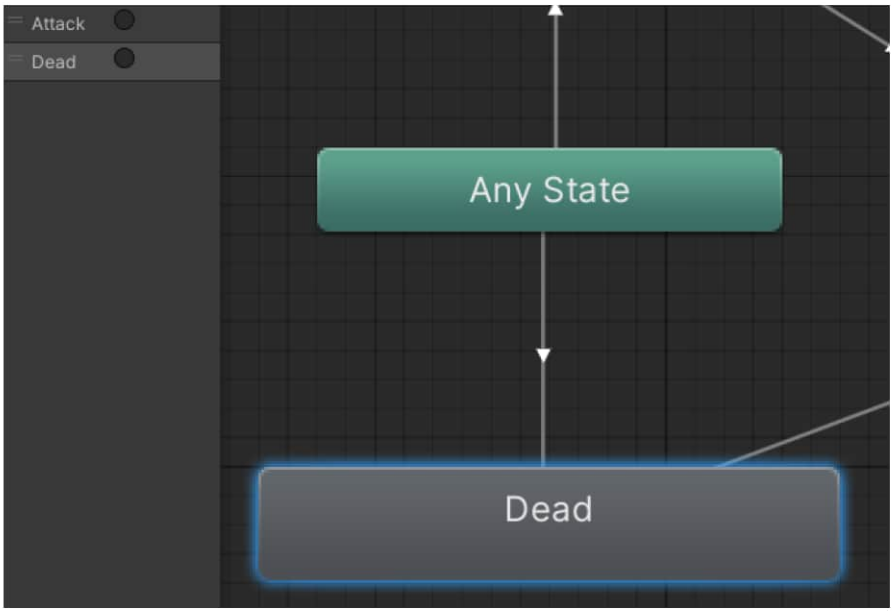



Рисунок 2.13 – Тригер Dead

Після запуску анімації смерті, скрипт EnemyHealth забезпечує повне завершення життєвого циклу ворога. Він деактивує компонент Collider2D на об'єкті ворога, щоб уникнути подальших фізичних взаємодій. Параметри компонента Rigidbody2D змінюються, що повністю зупиняє рух ворога та його фізичну взаємодію з ігровим світом. Скрипти EnemyAI та EnemyCombat вимикаються, оскільки мертвий ворог не повинен переслідувати чи атакувати. Також запускається корутина (DelayedDestroy()), яка забезпечує затримку перед повним видаленням ігрового об'єкта ворога зі сцени. Тривалість цієї

затримки враховує довжину анімаційного кліпу смерті, гарантуючи, що об'єкт зникне лише після повного відтворення анімації.

Консоль відображає послідовність подій: отримання урону героєм, зменшення здоров'я Skeleton до нуля, і, нарешті, повідомлення "Skeleton погибает". Це підтверджує коректне спрацьовування всієї логіки смерті та візуальної реакції ворога. (рис. 2.14)

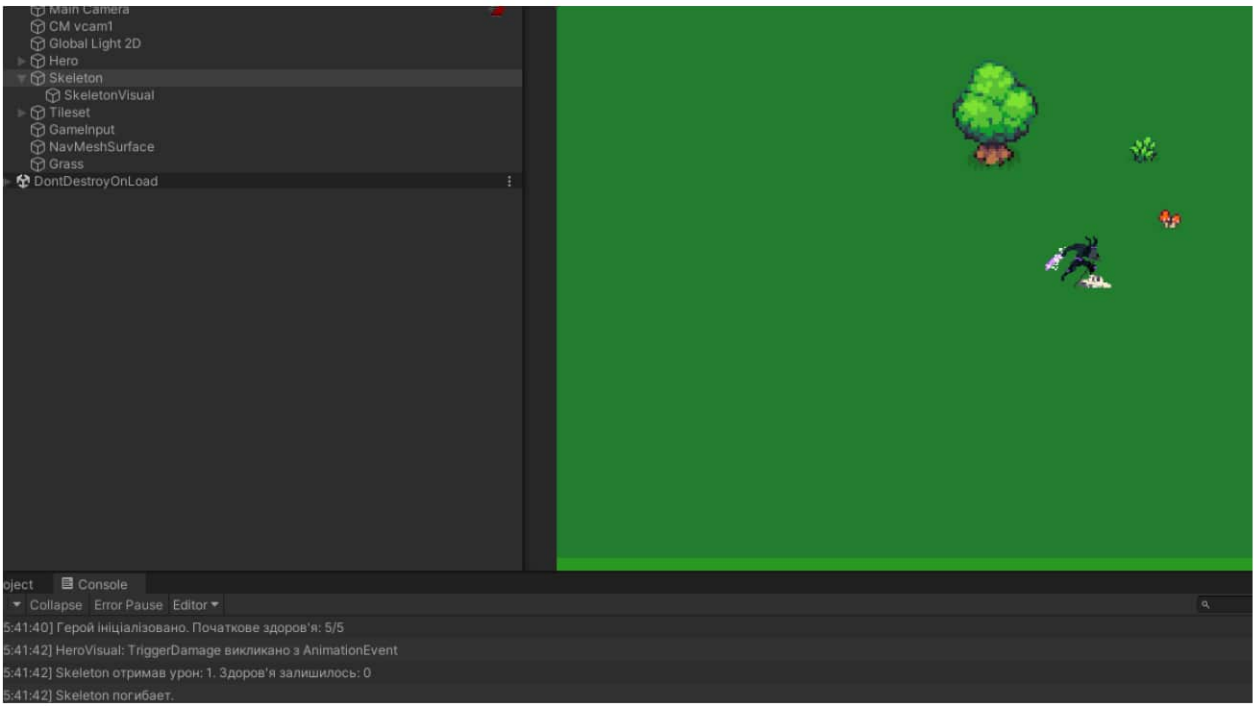



Рисунок 2.14 - Відображення отримання урону та загибелі монстра Skeleton у грі

## 2.3 Створення та функціонал головного героя

### 2.3.1 Створення об’єкта Hero та налаштування фізичної моделі

На початковому етапі розробки на ігрову сцену було додано порожній об'єкт "Hero", який став кореневим для персонажа. Для відокремлення логіки від візуалізації було створено дочірній об'єкт HeroVisual, до якого додали компоненти Sprite Renderer для відображення графіки та Animator для

керування анімаціями. Така структура дозволяє незалежно маніпулювати візуальною частиною, не впливаючи на фізичну модель. (рис. 2.15)

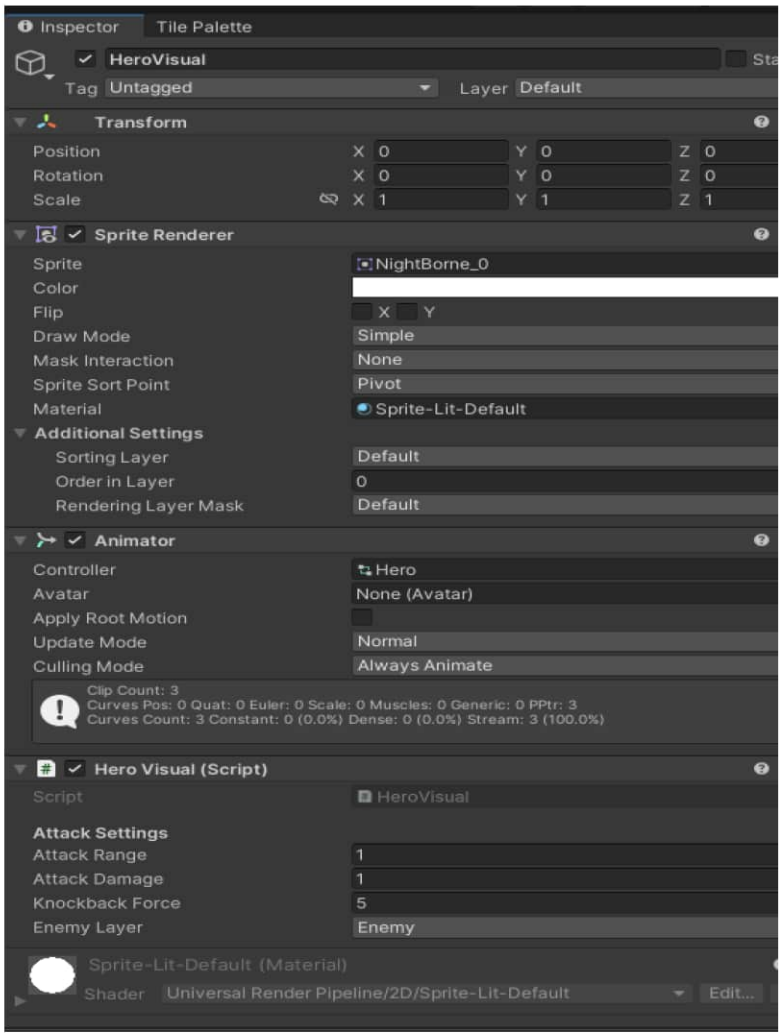



Рисунок 2.15 - Компоненти дочірнього об'єкта HeroVisual

Для взаємодії з ігровим світом основний об'єкт Hero було налаштовано. Йому було присвоєно тег та шар Player для ідентифікації системою. Далі було додано фізичні компоненти. Компонент Rigidbody 2D було налаштовано для гри з видом зверху: Body Type встановлено як Dynamic, Gravity Scale – 0, а також було заморожено обертання по осі Z (Freeze Rotation Z) для стабільності. (рис.2.16)

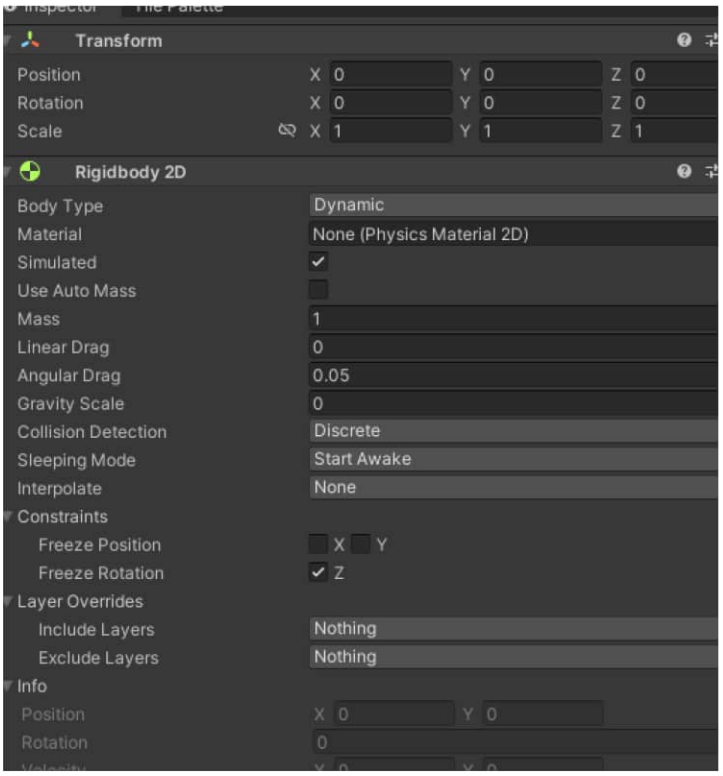



Рисунок 2.16 – Налаштування компонентів Hero

Далі для формування фізичних границь і реєстрації зіткнень було додано компонент Box Collider 2D. Розміри та положення цього підігнано під контури спрайту персонажа, що забезпечує точну та реалістичну взаємодію з іншими об'єктами на сцені. Всі скрипти, що відповідають за логіку та здоров'я, були прикріплені до основного об'єкта Hero. (рис. 2.17)

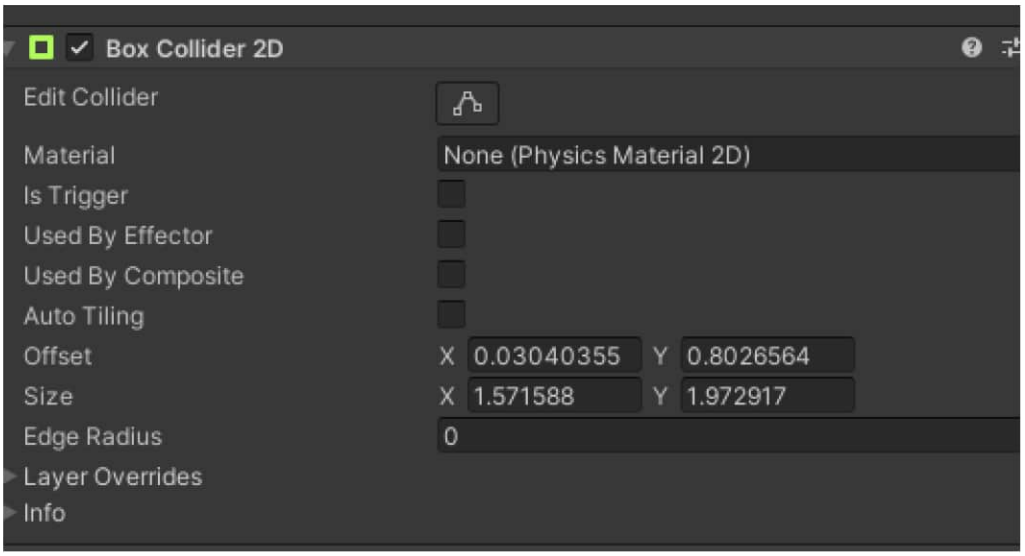



Рис.2.17 – Налаштування Box Collider 2D об'єкта Hero

### 2.3.2 Підготовка графічних ресурсів та анімацій

Для початку було інстальовано асет головного героя Hero [11], оскільки саме він є є центральною фігурою в грі Hero’s Adventure.

Наступним кроком у роботі з графікою стало нарізання спрайтів за допомогою вбудованого Sprite Editor. Інструмент було переключено в режим Multiple, після чого за допомогою автоматичної сітки та ручних коригувань було виділено кожен кадр анімації. Це дозволило отримати окремі зображення для анімацій спокою, бігу, атаки та смерті персонажа.

Для кожного з отриманих фрагментів було встановлено точку опори (Pivot). Відповідно до стилю анімації, Pivot було зміщено до центру нижнього краю спрайта. Такий вибір забезпечує коректне вирівнювання персонажа на ігровій поверхні під час пересування та запобігає візуальному «підстрибуванню» анімацій. (рис. 2.18)

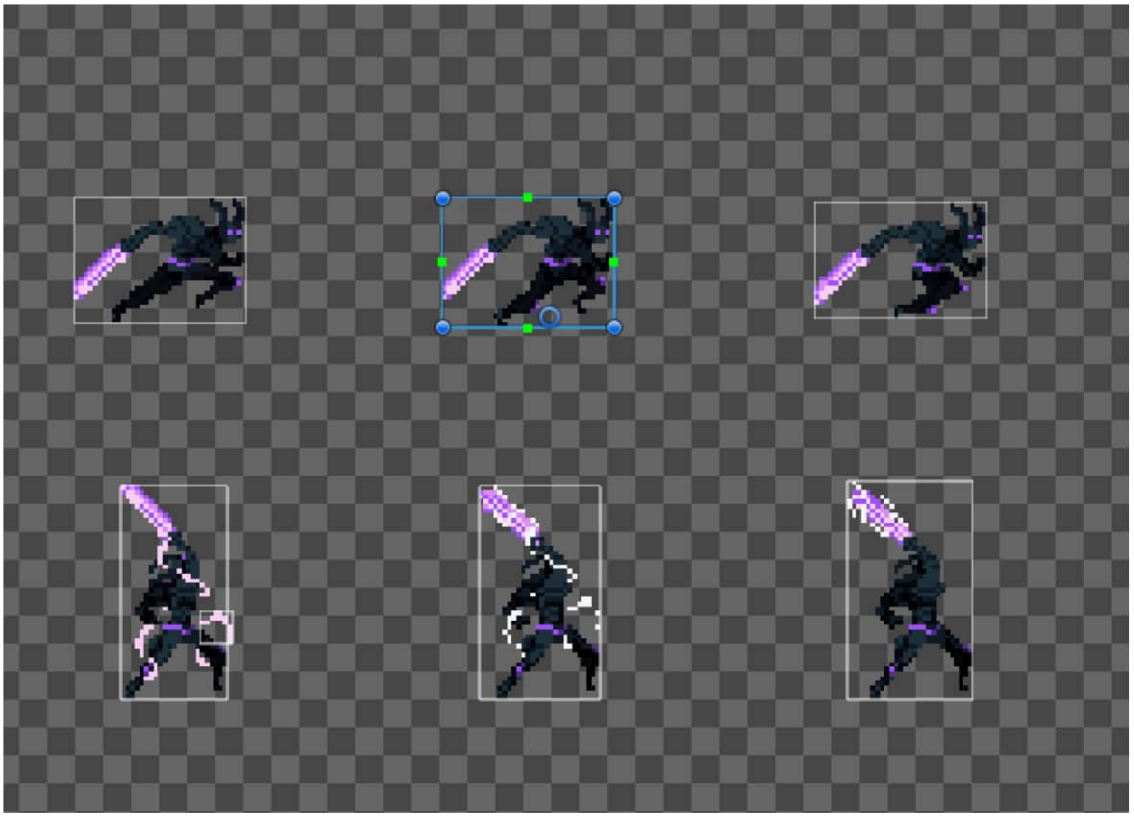


Рисунок 2.18 - Налаштування та нарізка спрайтів героя

### 2.3.3 Підготовка графічних ресурсів та створення анімацій

Після того, як було закладено фундамент для фізичної поведінки героя, наступним етапом стала робота з його візуальним представленням та анімаціями. Цей процес розпочався з імпорту та налаштування графічних асетів. Спрайт-аркуш персонажа було імпортовано в проєкт, після чого в інспекторі налаштувань для цього файлу було змінено параметр Texture Type на Sprite (2D and UI), а Sprite Mode встановлено в режим Multiple. Ці зміни є обов'язковими, оскільки вони вказують рушію Unity, що даний файл є не простою текстурою, а набором двовимірних зображень, які необхідно обробляти окремо. (рис.2.19)

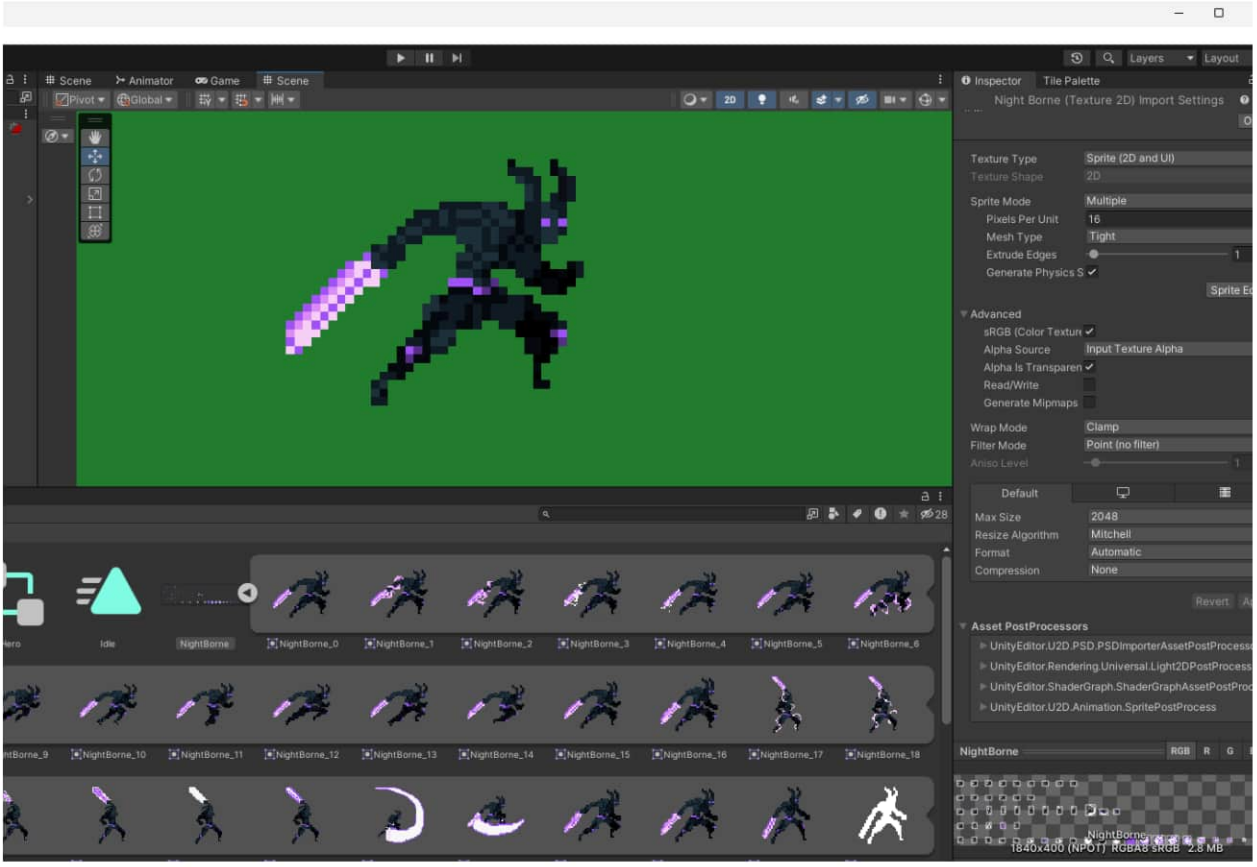



Рисунок 2.19 – Налаштування Спрайту

Наступним кроком стало нарізання спрайт-аркуша на окремі кадри за допомогою вбудованого інструменту Sprite Editor. У редакторі було використано функцію автоматичного нарізання за сіткою, яка виділила кожен

кадр анімації в окремий спрайт. Це дозволило отримати готові до використання зображення для всіх станів персонажа: спокою, бігу, атаки та смерті. Такий підхід значно прискорює процес підготовки ресурсів, дозволяючи зосередитись на створенні анімаційних послідовностей. (рис. 2.20)

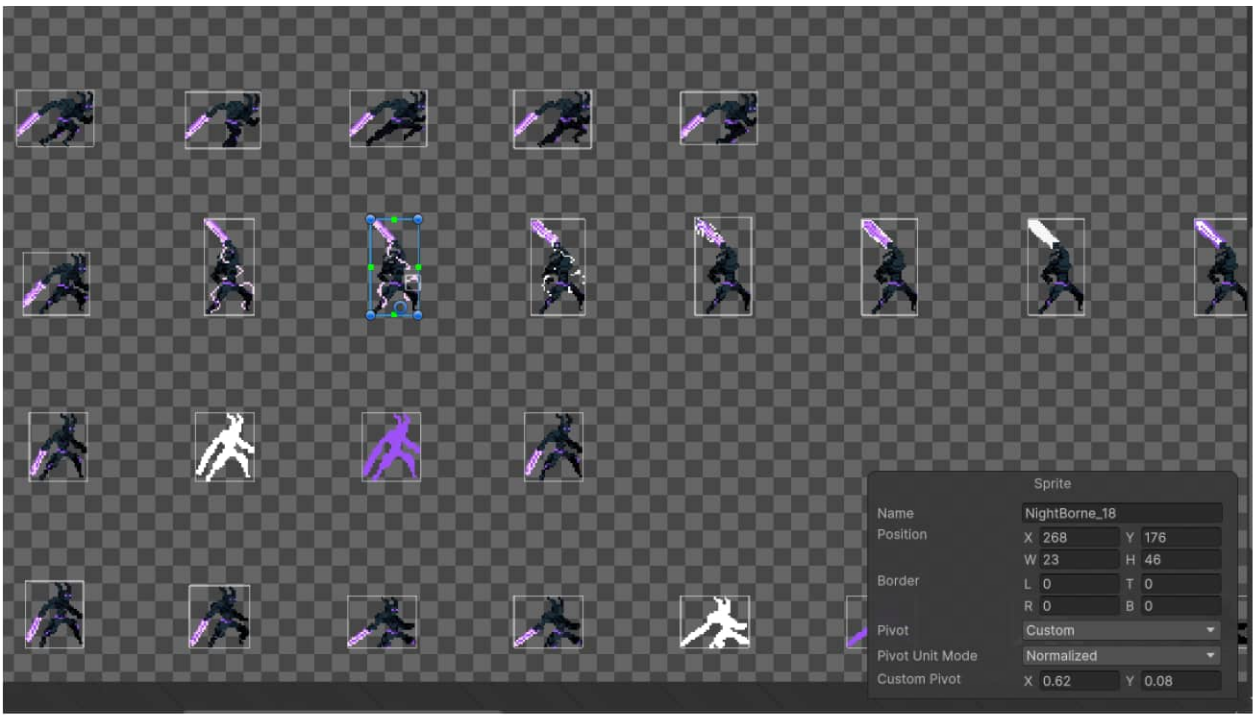



Рисунок 2.20 - Нарізання спрайту і налаштування положення Pivot

Важливим кроком у підготовці спрайтів було налаштування точки опори (Pivot) для кожного кадру. Pivot є точкою відліку для позиціонування, обертання та масштабування спрайту. Щоб забезпечити коректну поведінку персонажа на ігровій сцені, Pivot було зміщено до центру нижнього краю кожного зображення. Таке рішення гарнтує, що незалежно від зміни розміру спрайту під час анімації, "ноги" персонажа завжди залишаються на одному рівні відносно поверхні, що унеможливлює візуальний ефект "плавання" над землею.

Маючи набір підготовлених спрайтів, було розпочато процес створення анімаційних кліпів. У вікні Animation для об'єкта "HeroVisual" було створено чотири анімаційні кліпи: Idle, Run, Attack та Dead (рис. 2.21). Кожен кліп було наповнено відповідною послідовністю спрайтів, які були просто перетягнуті на часову шкалу анімації. Шляхом налаштування частоти кадрів для кожного кліпу було досягнуто плавності та природності рухів персонажа.

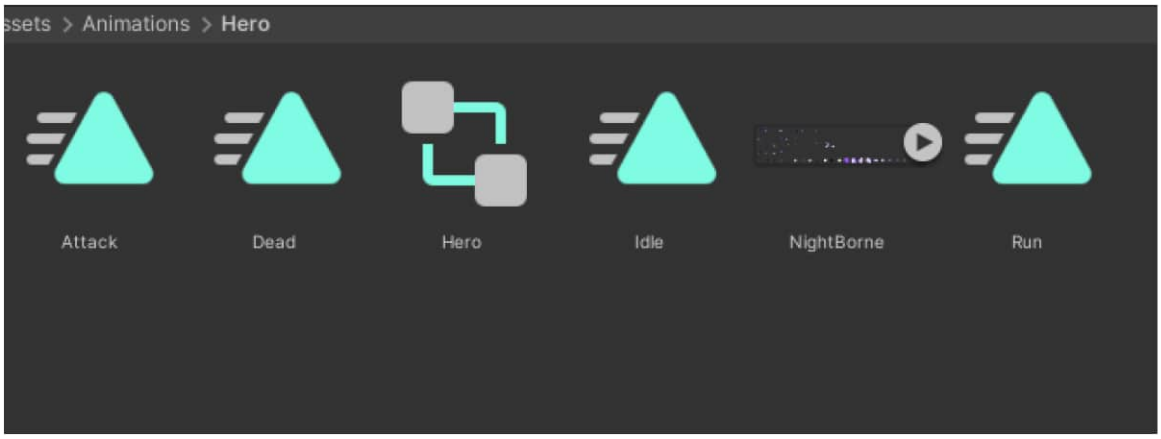



Рисунок 2.21 - Налаштовані анімації

Останім етапом налаштування анімацій було створення та конфігурація Animator Controller. Цей компонент діє як скінченний автомат (Finite State Machine), що керує переходами між анімаційними станами. У контролері було створено стани, що відповідають кожному анімаційному кліпу. Для керування переходами між ними було створено параметри: IsRunning (типу bool) для перемикання між станами спокою та бігу, а також Attack та Dead (типу Trigger) для запуску відповідних анімацій. Логіка переходів була налаштована таким чином, щоб забезпечити миттєву реакцію аніматора на зміни стану персонажа, що надходять з ігрових скриптів. (рис. 2.22)

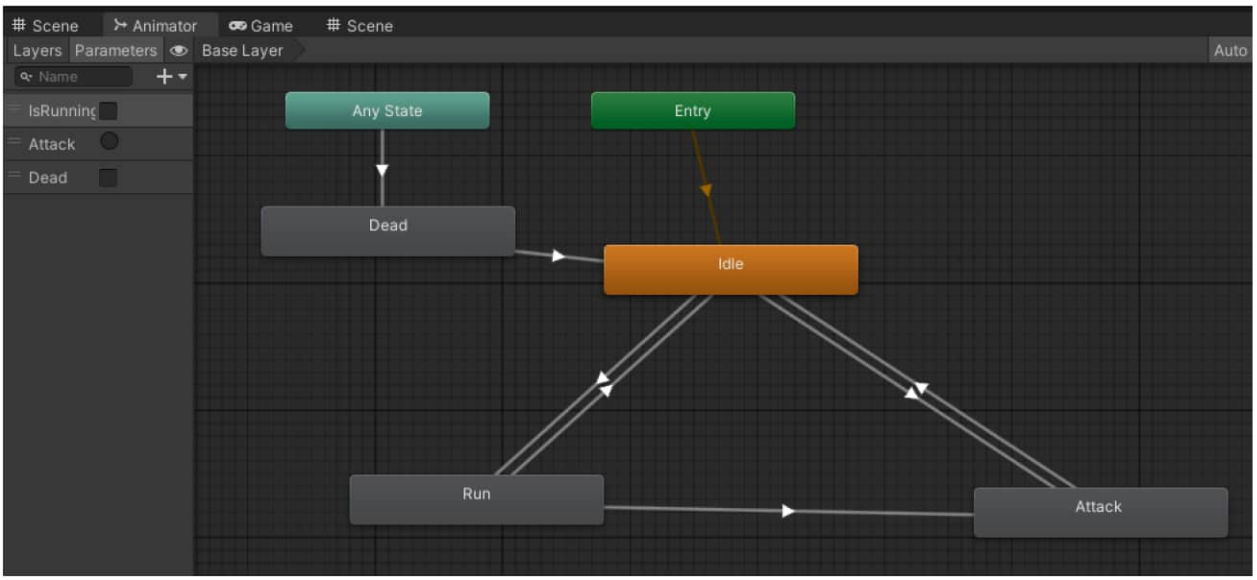



Рисунок 2.22 – Налаштована логіка анімацій героя

### 2.3.4 Реалізація системи керування

З метою створення гнучкої, сучасної та розширюваної системи керування було прийнято рішення відмовитися від застарілого Input Manager на користь нового пакета Input System від Unity. Цей вибір аргументовано ключовими перевагами, як-от повне виділення логіки гри від коду обробки вводу, нативна підтримка різноманітних пристроїв та можливість створювать контекстно-залежні набори дій. Процес інтеграції розпочався зі встановлення пакета через Package Manager, після чого у налаштуваннях проєкту параметр Active Input Handling було встановлено в режим Both щоб забезпечити максимальну сумісність.

Центральним елементом нової системи став ассет Input Actions, якому було надано ім'я HeroInputActions, і який став єдиним центром для визначення усіх можливих дій гравця. В його структурі було створено дві логічні карти дій (Action Maps): Hero та Combat. Перша карта об'єднала всі дії, що стосуються пересування персонажа, тоді як друга була призначена виключно для бойових дій. Таке розділення є ефективним архітектурним рішенням, що дозволяє динамічно вмикати чи вимикати цілі набори елементів керування залежно від ігрового контексту.

У межах карти Hero було налаштовано дію Move, що відповідає за рух персонажа. Її тип Action Type було встановлено як Value, а тип контролю Control Type - Vector2, оскільки рух є тривалим процесом, який найкраще описується двовимірним вектором напрямку. Щоб надати гравцеві вибір, було реалізовано дві альтернативні схеми керування через композитні прив'язки Composite Bindings: перша об'єднала клавіші WASD, а друга - клавіші-стрілки. Таким чином, незалежно від уподобань гравця, дія Move повертає коректний вектор руху. (рис. 2.23)

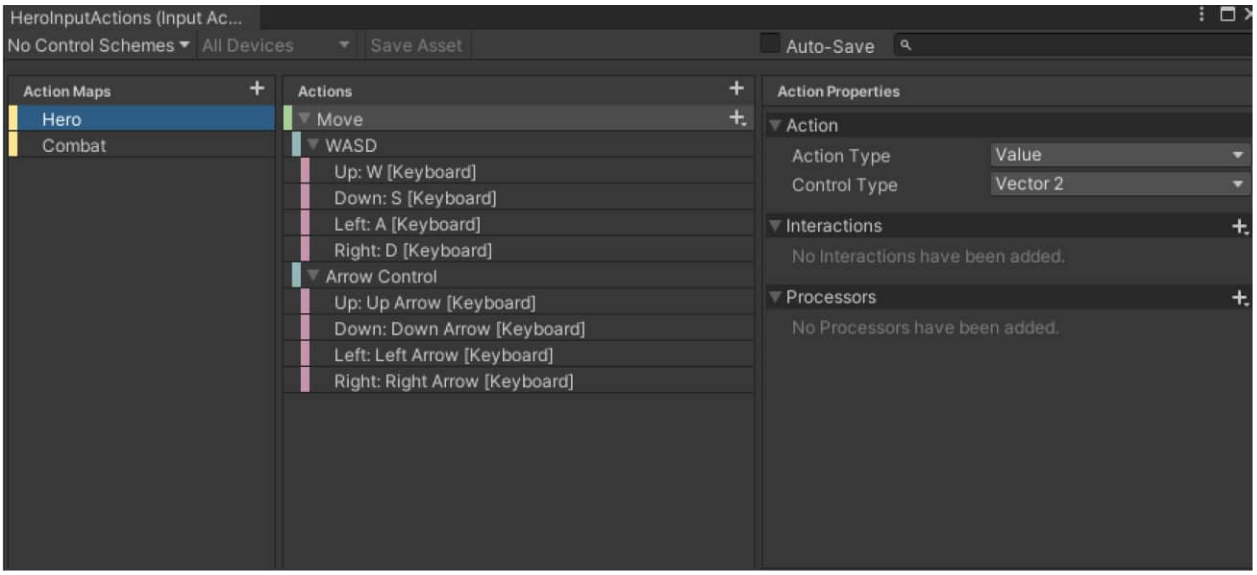



Рисунок 2.23 - Налаштування дії Move в Input Actions

Для реалізації бойової механіки у карті Combat було створено дію Attack. Оскільки атака є миттєвою, дискретною подією, її тип було встановлено як Button. Цю дію було прив'язано до натискання лівої кнопки миші, що є загальноприйнятим стандартом для ігор даного жанру та забезпечує інтуїтивно зрозуміле керування для гравця. (рис. 2.24)

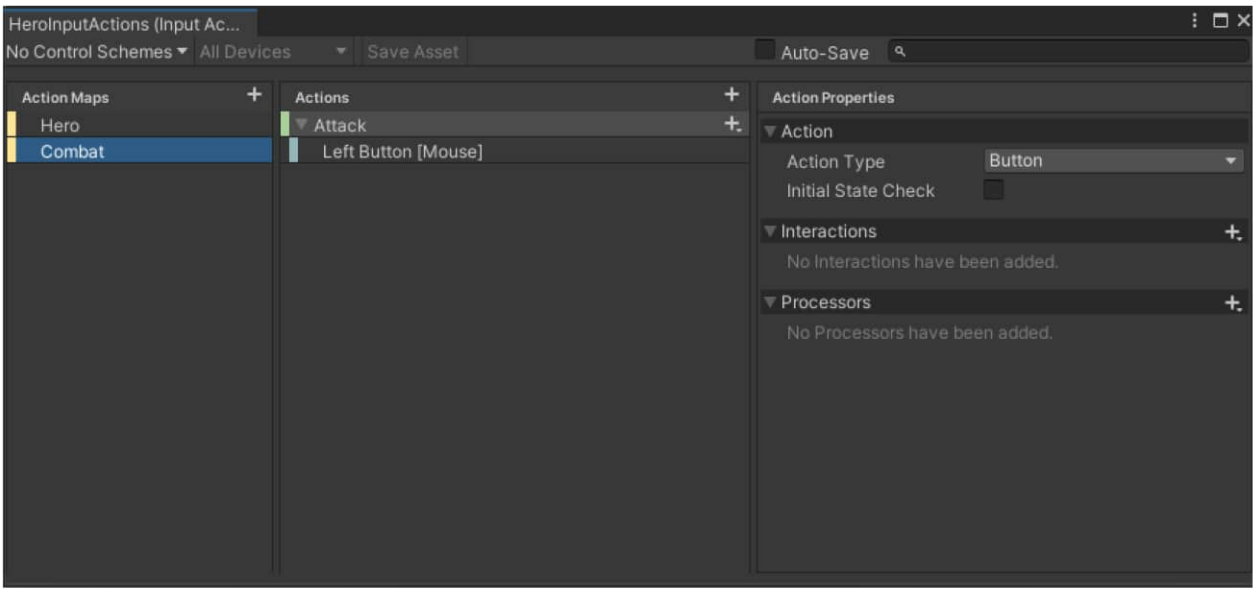



Рисунок 2.24 - Налаштування дії Attack в Input Actions

Для зв'язування асету HeroInputActions з ігровою логікою було розроблено скрипт GameInput, який реалізує патерн проєктування Singleton. Це забезпечує єдину глобальну точку доступу до системи вводу, що значно спрощує архітектуру та усуває потенційні конфлікти. У методі Awake цього скрипта відбувається ініціалізація та активація всіх визначених карт дій. Програмна реалізація цього скрипта наведена у Додатку А.

У скрипті GameInput було застосовано два підходи до обробки даних. Для миттєвої дії атаки використано подієво-орієнтований підхід: скрипт підписується на подію started дії Attack і, при її спрацьовуванні, ініціює власну подію OnHeroAttack. Це ефективний механізм, що забезпечує низьку зв'язність компонентів. Для безперервного процесу руху застосовано метод опитування через публічний метод GetMovementVector, який в будь-який момент повертає поточний вектор руху для використання у фізичних розрахунках. Повна реалізація цього скрипта наведена у Додатку А.

Окрім обробки кнопок, GameInput також відповідає за надання інформації про положення курсора миші. Метод GetMousePosition напряму зчитує екранні координати курсора. Ці дані, хоч і не є дією у звичному

розумінні, є критично важливими для ігрової механіки, оскільки використовуються скриптом HeroVisual для реалізації повороту спрайту персонажа в напрямку прицілювання, що робить керування більш чутливим та динамічним. Програмна реалізація цього скрипта наведена у Додатку А.

### 2.3.5 Реалізація життєвого циклу та бойової системи героя

Процес нанесення шкоди ворогові починається з дії гравця. Коли гравець натискає кнопку атаки, скрипт GameInput ініціює подію OnHeroAttack. На цю подію реагує скрипт Hero, який, у свою чергу, викликає метод PlayAttackAnimation() у дочірньому компоненті HeroVisual. Цей метод активує тригер Attack в Animator Controller, запускаючи анімацію удару (рис. 2.25). Ключовим моментом цієї механіки є використання Animation Event – спеціальної мітки на часовій шкалі анімації. У кадрі, що візуально відповідає моменту удару, ця подія викликає метод TriggerDamage() у скрипті HeroVisual.

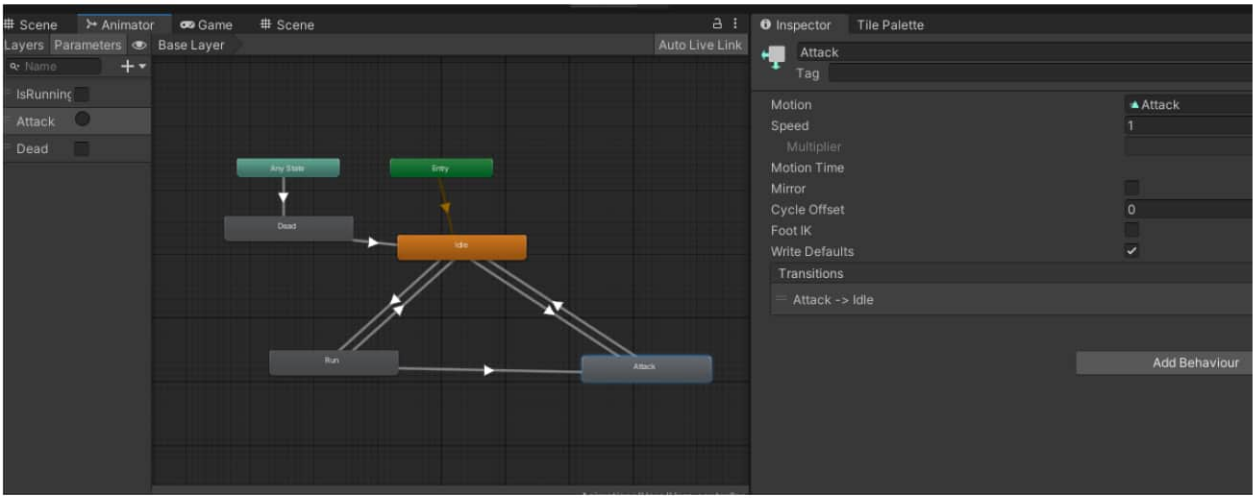



Рисунок 2.25 – Налаштування тригерів

Метод TriggerDamage() відповідає за фактичне нанесення шкоди. Він створює невидиму круглу зону (Physics2D.OverlapCircleAll) навколо героя з радіусом, заданим у полі Attack Range. Система перевіряє наявність у цій зоні об'єктів на шарі Enemy. Якщо ворога знайдено, скрипт отримує доступ до його компонента EnemyHealth і викликає метод TakeDamage, передаючи значення ушкодження з поля Attack Damage. Таким чином, нанесення шкоди чітко

синхронізоване з візуальною анімацією. Детальна програмна реалізація цього ланцюжка подій наведена у скриптах у Додатку А.

Отримання ушкоджень та життєвий цикл героя контролюються скриптом HeroHealth. Цей компонент містить інформацію про максимальне (maxHealth) та поточне здоров'я персонажа. Максимальне здоров'я було встановлено на 5 одиниць в інспекторі Unity. Коли ворог успішно атакує героя, його скрипт EnemyCombat викликає публічний метод TakeDamage(int damage) у HeroHealth героя. Цей метод зменшує поточне здоров'я і перевіряє, чи не досягло воно нульової позначки (рис. 2.26). Програмна реалізація скрипта HeroHealth наведена у Додатку А.

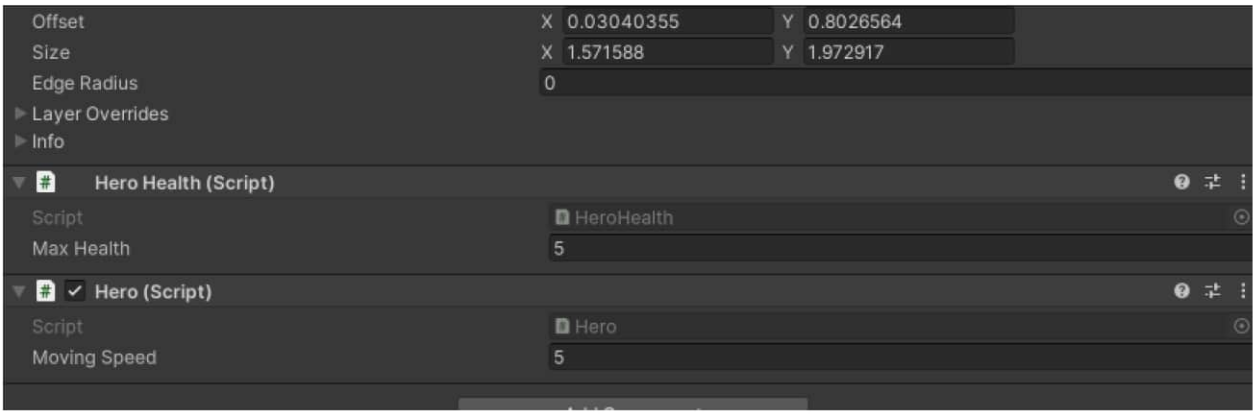



Рисунок 2.26 – Скрипти об’єкта Hero

Завершальним етапом життєвого циклу є логіка смерті, реалізована у приватному методі Die() скрипта HeroHealth. Коли здоров'я героя закінчується, цей метод запускається автоматично. Першочергово він активує тригер Dead в аніматорі для негайного відтворення анімації смерті. Далі, для запобігання побічним ефектам, встановлюється прапорець isDead і послідовно вимикаються всі активні компоненти: скрипт Hero (щоб зупинити рух), HeroVisual (щоб зупинити візуальні оновлення), а також Collider2D (щоб тіло не взаємодіяло з іншими об'єктами). Наостанок, Rigidbody2D переводиться у кінематичний режим, що повністю зупиняє його фізичну симуляцію. Повна реалізація цієї логіки наведена у Додатку А.

## 2.4 Налаштування камери

### 2.4.1 Cinemachine камера

Для створення динамічної камери, що плавно слідує за головним героєм, було вирішено використати пакет Unity - Cinemachine. Цей інструмент дозволяє реалізовувати складні операторські задачі без написання коду, надаючи набір готових модулів для керування камерою. Процес інтеграції розпочався зі встановлення пакета Cinemachine через Package Manager з репозиторію Unity Registry. (рис. 2.27)

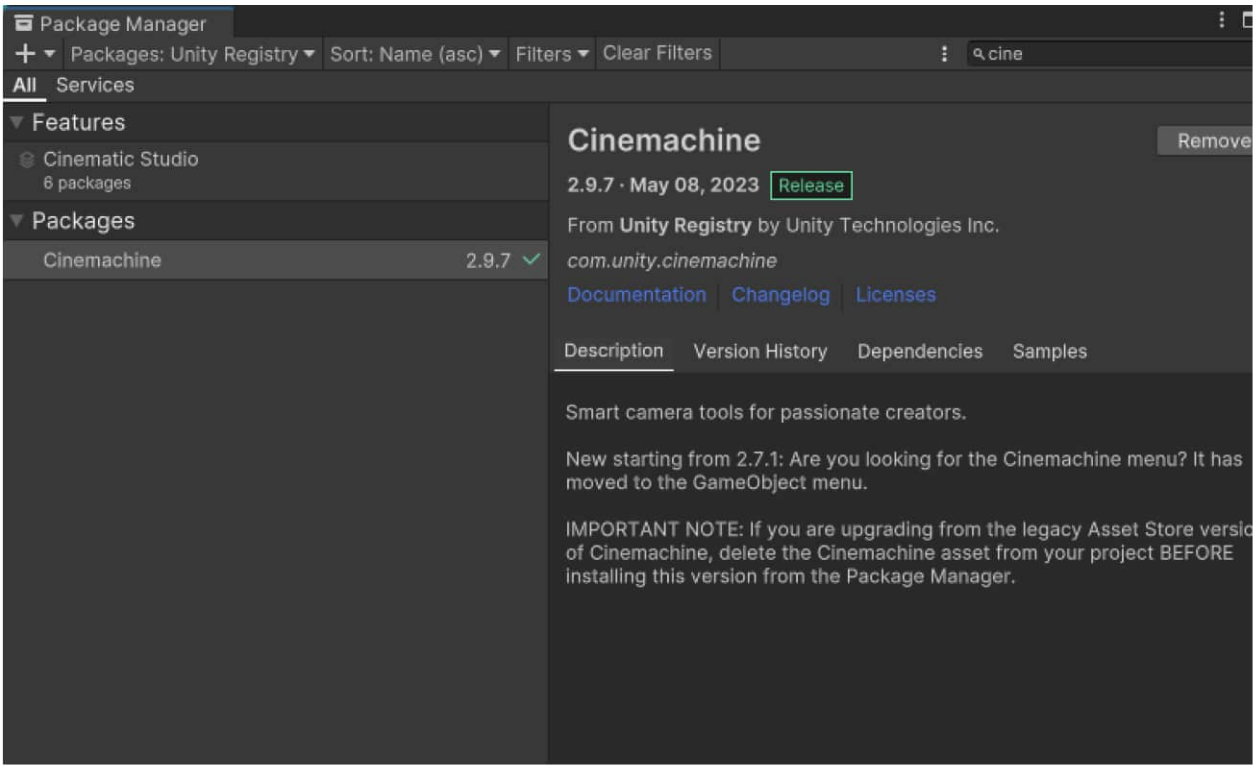



Рисунок 2.27 – Встановлення пакета Cinemachine через Package Manager

Ключовою особливістю Cinemachine є те, що він не замінює основну камеру сцени, а керує нею. Після додавання першого об'єкта Cinemachine, до Main Camera автоматично додається компонент CinemachineBrain. Цей компонент є мозком усієї системи: він відстежує всі активні віртуальні камери на сцені та плавно змінює позицію й параметри основної камери, щоб вони відповідали налаштуванням поточної активної віртуальної камери. (рис. 2.28)

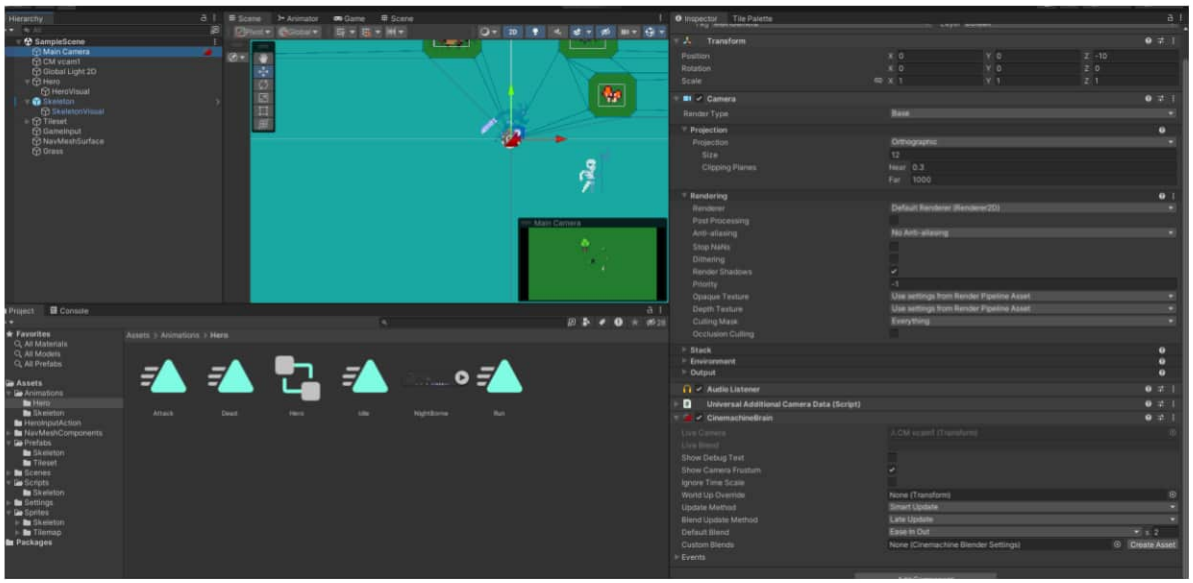


Рисунок 2.28 – Компонент CinemachineBrain на основній камері сцени

Наступним кроком стало створення самої віртуальної камери. Через меню GameObject > Cinemachine було додано об'єкт Virtual Camera, якому було надано ім'я "CM vcam1". Для реалізації слідування за героєм, його об'єкт Hero було перетягнуто у поле Follow компонента CinemachineVirtualCamera. Це головне налаштування, яке змушує віртуальну камеру постійно рухатись за персонажем. Оскільки гра є двовимірною з ортографічною проєкцією, у розділі Lens було встановлено параметр Ortho Size на 12, що визначило масштаб видимої області гри. (рис. 2.29)

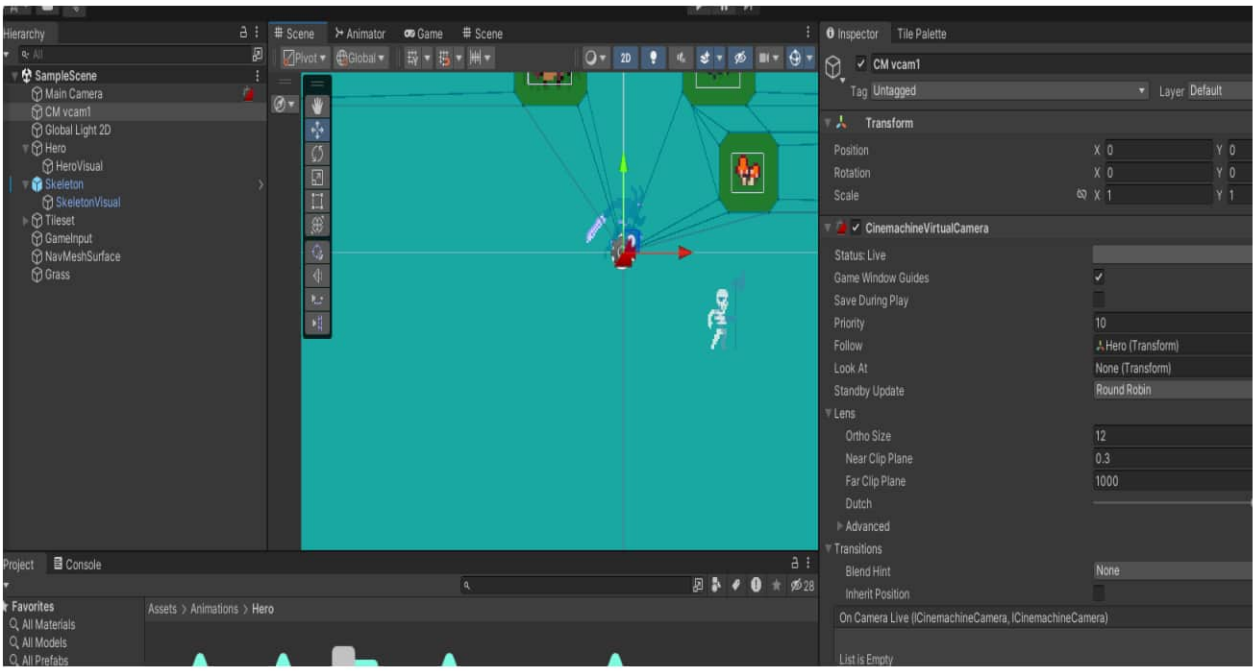



Рисунок 2.29 - Налаштування віртуальної камери для слідування за героєм

В результаті, налаштування основної камери сцени - Main Camera, також було приведено у відповідність до ортографічного режиму з аналогічним значенням Size. Таким чином, було реалізовано наступний ланцюжок взаємодії: гравець керує об'єктом Hero; віртуальна камера "CM vcam1" відстежує позицію Hero; CinemachineBrain на основній камері отримує дані від "CM vcam1" і оновлює реальну позицію камери на сцені. Такий підхід забезпечив плавну та динамічну камеру, що слідує за гравцем, і заклав фундамент для можливих подальших ускладнень, як-от перемикання між кількома камерами чи створення кінематографічних ефектів.

## 2.5 Побудова навігаційної сітки для пересування AI-агентів у 2D-середовищі

### 2.5.1 Початкове налаштування системи навігації AI у Unity

Для створення системи AI пересування ворожих персонажів, яка б дозволяла їм самостійно рухатися по заданих точках, уникати зіткнень з об'єктами та обходити перешкоди на шляху, було використано систему навігації. Основним засобом для реалізації цієї логіки стало підключення пакета AI Navigation, який додається до проєкту через Unity Package Manager. (рис. 2.30)

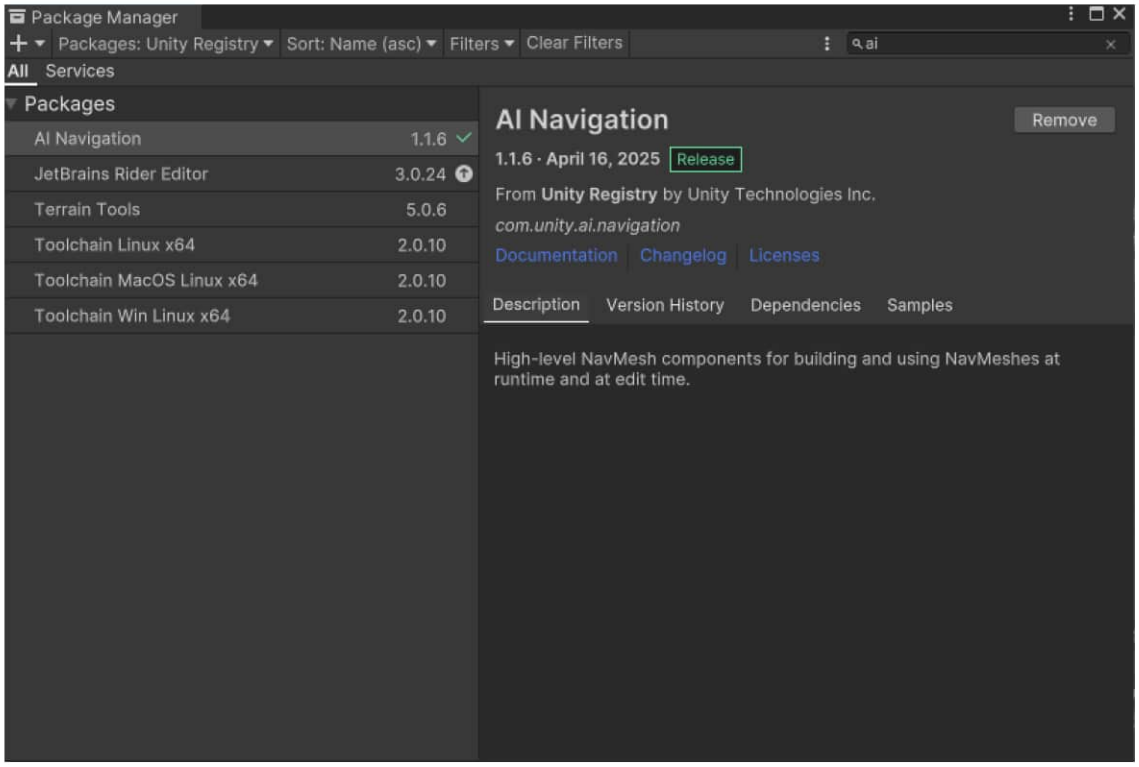



Рисунок 2.30 - Пакет AI Navigation

AI Navigation є інструментом, що дозволяє будувати навігаційні сітки (NavMesh) для об'єктів сцени. Після генерації такої сітки вороги отримують можливість знаходити оптимальні маршрути до цільових точок, автоматично оминаючи перешкоди. Однак слід зазначити, що цей інструмент із самого початку розроблявся переважно для 3D-оточення. За замовчуванням усі

обчислення базуються на даних глибини та висоти, що не відповідає умовам 2D-гри з виглядом згори.

Через це виникла низка обмежень. Наприклад, у двовимірній сцені компоненти NavMeshSurface та NavMeshAgent не працюють коректно, якщо використовуються в стандартному вигляді. Усі ці елементи орієнтовані на простір з осями X, Y, Z, тоді як у 2D середовищі основна площина зазвичай розміщується в площині X-Y, а вісь Z використовується для сортування об’єктів за відстанню до камери.

AI Navigation є інструментом, що дозволяє будувати навігаційні сітки (NavMesh) для об'єктів сцени. Після генерації такої сітки вороги отримують можливість знаходити оптимальні маршрути до цільових точок, автоматично оминаючи перешкоди. Однак слід зазначити, що цей інструмент із самого початку розроблявся переважно для 3D-оточення. За замовчуванням усі обчислення базуються на даних глибини та висоти, що не відповідає умовам 2D-гри з виглядом згори.

Через це виникла низка обмежень. Наприклад, у двовимірній сцені компоненти NavMeshSurface та NavMeshAgent не працюють коректно, якщо використовуються в стандартному вигляді. Усі ці елементи орієнтовані на простір з осями X, Y, Z, тоді як у 2D середовищі основна площина зазвичай розміщується в площині X-Y, а вісь Z використовується для сортування об’єктів за відстанню до камери.

### 2.5.2 Адаптація NavMesh для 2D та налаштування Tilemap

Після встановлення пакета AI Navigation через Unity Package Manager виникла потреба в розширенні його можливостей для роботи в 2D-середовищі. Оскільки вбудовані компоненти системи NavMesh переважно орієнтовані на тривимірний простір, було завантажено стороннє розширення з офіційного репозиторію GitHub [13]. Це розширення базується на стандартних елементах

NavMesh, але додатково забезпечує повноцінну підтримку 2D-навігації, включаючи необхідні компоненти та інструменти для побудови навігаційної сітки на площині XY (рис.2.31).

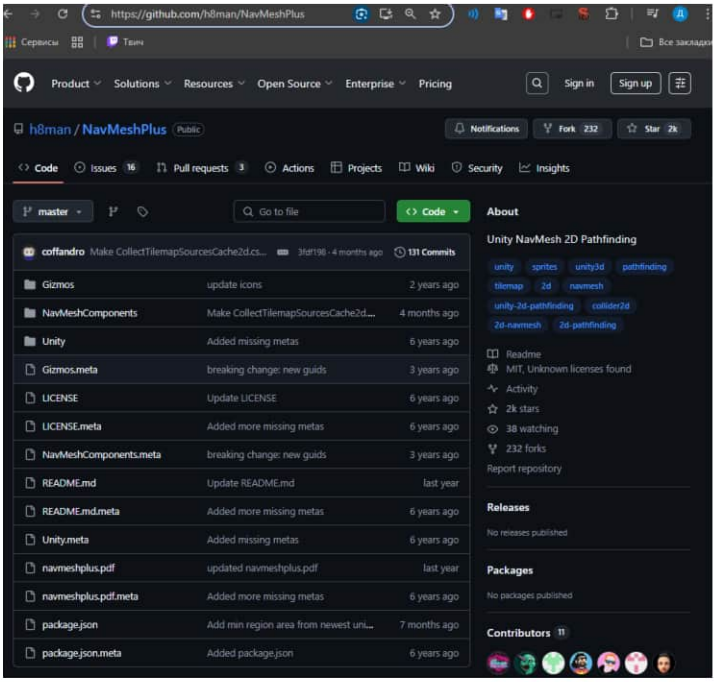



Рис.2.31 - Репозиторій NavMeshPlus із підтримкою 2D-навігації

Після впровадження розширення до проєкту стали доступні адаптовані компоненти, які дозволяють повноцінно працювати з навігаційними сітками у двовимірному ігровому просторі (рис.2.32).

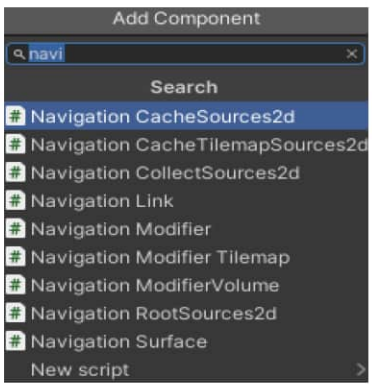



Рисунок 2.32 - Список адаптованих компонентів для роботи з 2D-навігацією

Далі до Tilemap-об’єкта GridPlatform(Grass), який виконує роль основної карти сцени, було додано компонент Navigation Modifier. Його основна

функція - визначення типу поверхні, яку система має враховувати під час побудови навігаційної сітки. Щоб позначити цю область як прохідну, в параметрах компонента було активовано опцію Override Area та обрано значення Walkable. Це означає, що всі елементи, розміщені на даній Tilemap, автоматично включаються до навігаційного простору й можуть використовуватись AI-агентами для прокладання маршруту.

### 2.5.3 Побудова та результати NavMesh

При побудові NavMesh на основі GridPlatform усі об’єкти (зокрема тайли з колайдерами) автоматично розпізнаються системою як елементи навігаційного простору, по якому може рухатися AI-агент (монстр). Це гарантує коректний облік перешкод та визначення дозволених маршрутів під час обчислення шляху.

Далі у сцені було створено окремий порожній об'єкт з назвою NavMeshSurface, який виконує функцію контейнера для всіх компонентів, пов’язаних з побудовою навігаційної сітки. До нього додано адаптований для 2D компонент Navigation Surface, який відповідає за обчислення, збереження та оновлення самої сітки в межах визначеної області. Паралельно було додано компонент Navigation CollectSources 2D - він виконує роль збирача джерел, з яких формується сітка (тайлмапи, об’єкти з колайдерами, модифікатори тощо) (рис. 2.33).

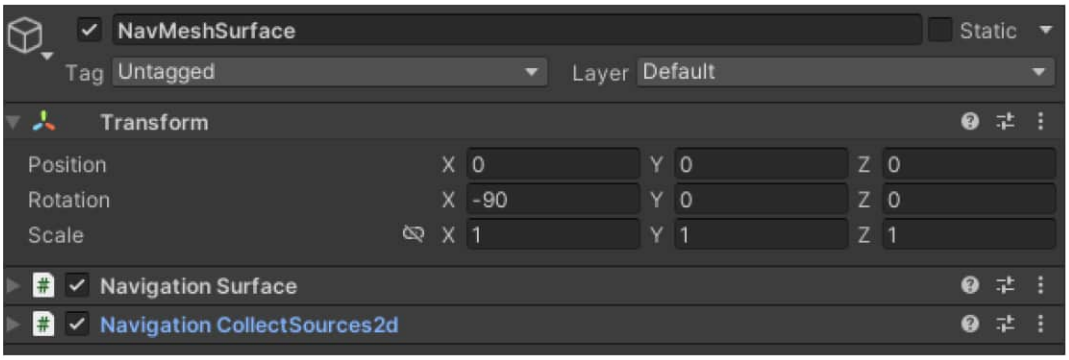



Рисунок 2.33 - Структура об’єкта NavMeshSurface з компонентами Navigation Surface та CollectSources 2D для побудови навігаційної сітки

Перед запуском побудови сітки було активовано опцію Rotate Surface to XY у компоненті CollectSources 2D - це необхідно для того, щоб повернути площину побудови з дефолтної XZ у площину XY, яка використовується у 2D-проєктах з виглядом згори. Завершальним етапом стало натискання кнопки Bake у компоненті Navigation Surface - це призвело до генерації навігаційної сітки на основі зібраних джерел (рис. 2.34).

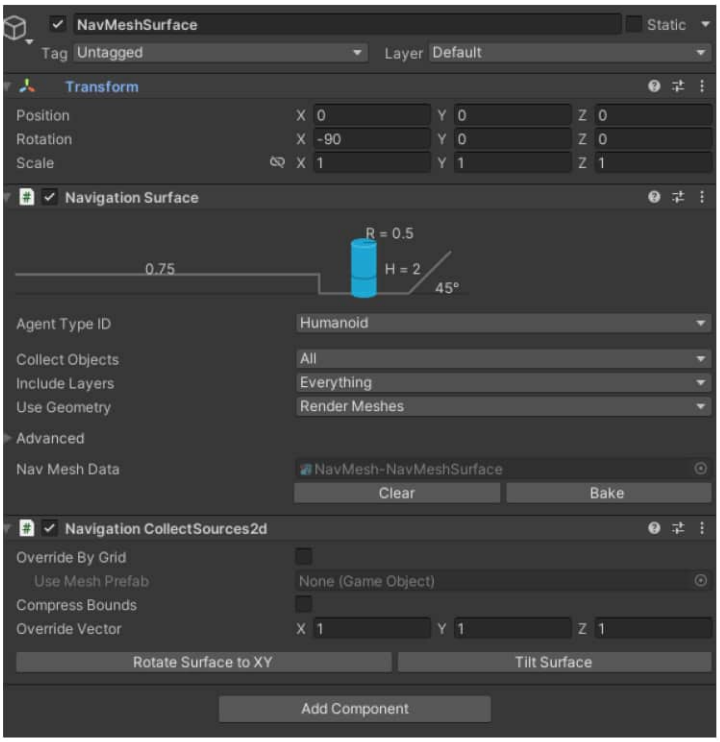



Рисунок 2.34 - Побудова 2D-NavMesh із врахуванням площини XY

У результаті було створено повноцінну навігаційну сітку, сумісну з 2D-оточенням. Система враховує всі модифіковані області, дозволяє AI-монстрам(агентам) обирати оптимальні шляхи та ефективно оминати перешкоди (рис. 2.35).

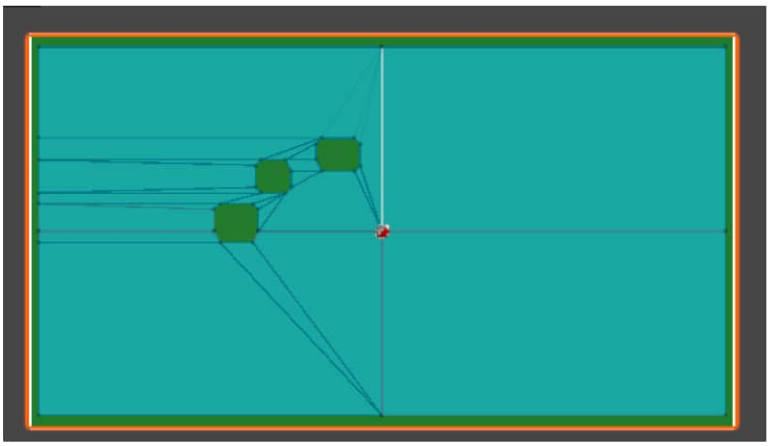


Рисунок 2.35 - Створена навігаційна сітка для AI-агентів

Далі для коректного виключення окремих зон із навігаційної сітки було застосовано компонент Navigation Modifier Volume. Його використання є доцільним у випадках, коли потрібно позначити лише частину об'єкта як непрохідну, наприклад, область довкола дерева або декоративного елемента, за яким передбачений прохід (рис. 2.35).

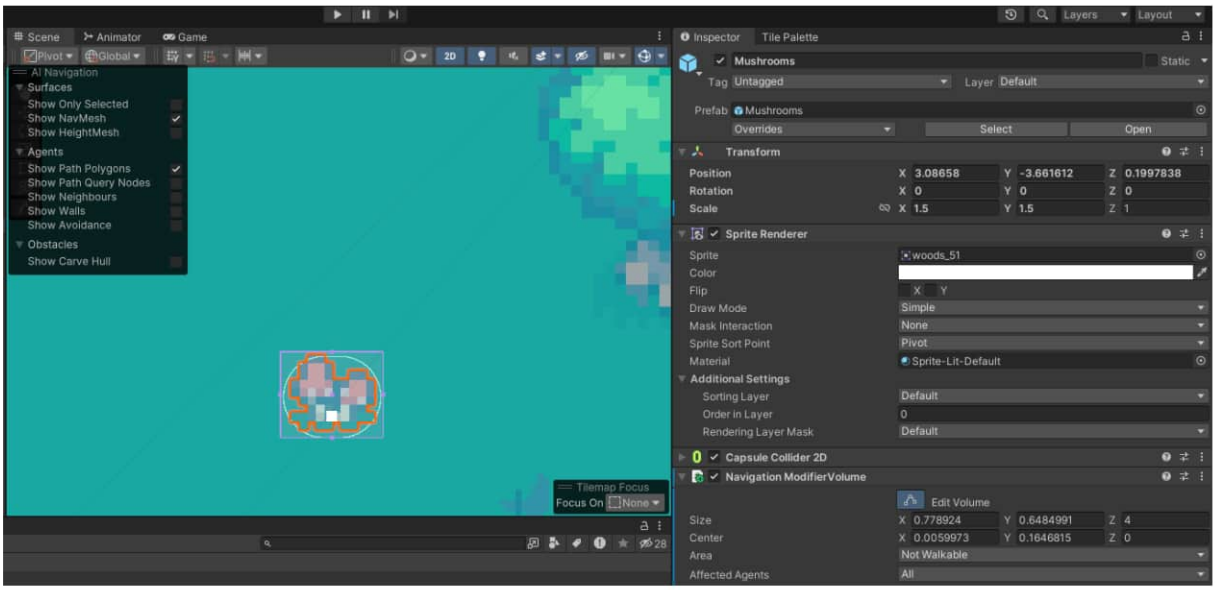



Рисунок 2.35 - Додання і налаштування NavigationModifierVolume до об’єкта Mushrooms

На наступному етапі було здійснено налаштування всіх об’єктів сцени, які мають розглядатися як перешкоди для AI-агентів. Для кожного з них відповідним чином застосовано компоненти модифікації навігаційного

простору, що дозволяє системі точно враховувати непрохідні зони під час побудови маршрутів.

Завершальним етапом стало повторне звернення до об’єкта NavMeshSurface, де в компоненті Navigation Surface було натиснуто кнопку Bake. У результаті цього процесу було оновлено навігаційну сітку з урахуванням усіх нових обмежень. Створена конфігурація забезпечує правильну логіку обходу: монстри не намагаються проходити крізь щільно розташовані об’єкти, а шукають оптимальний шлях в обхід перешкод. Такий підхід підвищує достовірність моделі поведінки AI у складному середовищі (рис. 2.36).

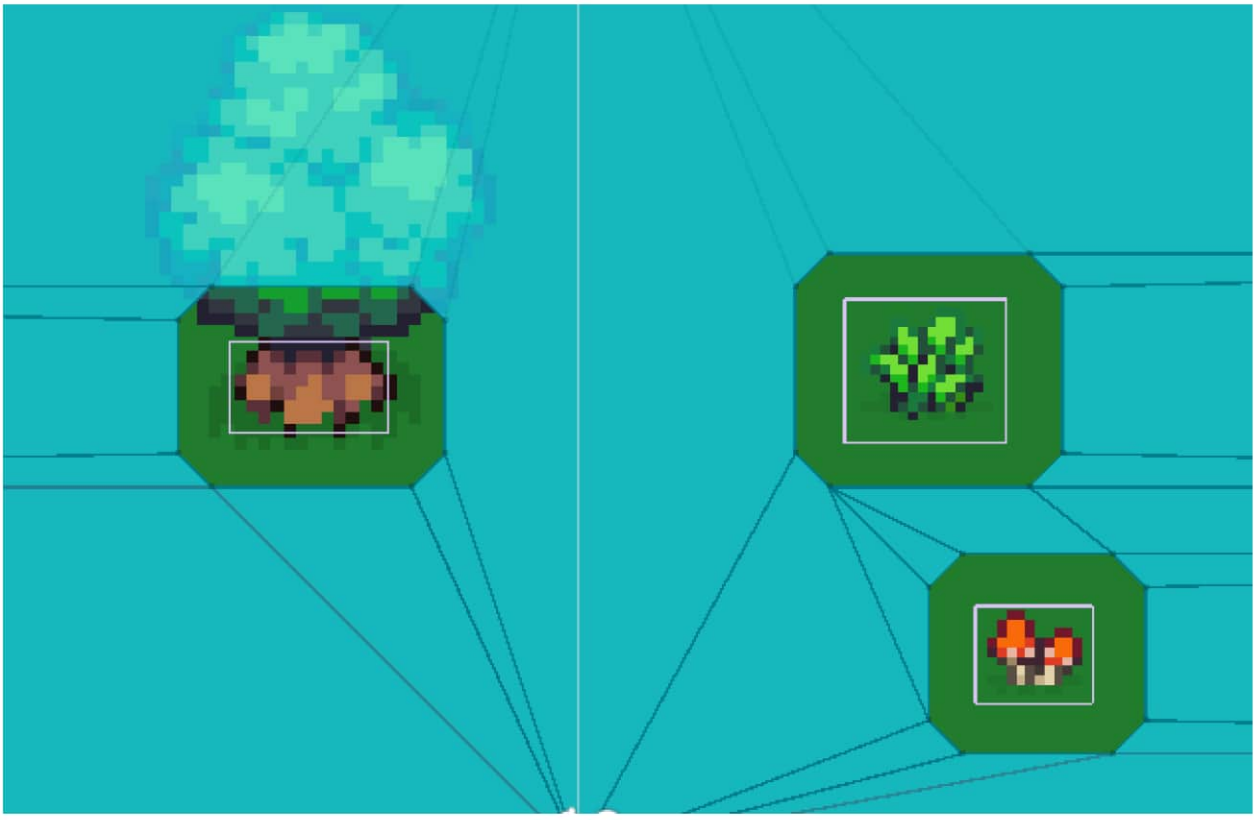


Рисунок 2.36 - Оновлена навігаційна сітка з налаштованими перешкодами для AI

## 2.6. Створення ігрового світу за допомогою Tilemap

### 2.6.1 Побудова ігрового середовища

Для побудови ігрового оточення та ландшафту було використано вбудовану систему Unity - Tilemap. Цей інструмент дозволяє ефективно створювати великі двовимірні рівні на основі сітки, що є оптимальним для 2D-ігор. На першому етапі було створено базовий шар Tilemap для землі, який за допомогою палітри був заповнений єдиним суцільним тайлом зеленого кольору. Такий підхід забезпечив створення однорідної та візуально цілісної поверхні, яка слугує основою для всього ігрового світу.

Для додавання деталей, таких як дерева, кущі та каміння, було застосовано гібридний підхід. Замість того, щоб бути частиною сітки тайлів, ці об'єкти були реалізовані як окремі ігрові об'єкти (префаби), розміщені поверх базового шару. Кожен такий об'єкт, наприклад "Shrub" (Кущ), має власний компонент Sprite Renderer для візуалізації та індивідуальний колайдер, як-от Capsule Collider 2D, для точної фізичної взаємодії. Це дозволило не тільки створити більш деталізовані форми перешкод, але й заклало можливість для додавання унікальної логіки до кожного декоративного елемента в майбутньому. Кінцевий результат такого поєднання, що створює візуально насичений та інтерактивний світ, можна побачити на сцені гри (рис. 2.37).

Рисунок 2.37 – Створення ігрової сцени

## 2.7 Процедурна генерація оточення

### 2.7.1 Реалізація алгоритму випадкового розміщення об'єктів

З метою підвищення різноманітності ігрового світу та автоматизації процесу наповнення рівня деталями, було впроваджено механізм процедурної генерації. Для цього було розроблено алгоритм, що базується на випадковому розміщенні об'єктів у заздалегідь визначеній області. Цей підхід дозволяє створювати унікальні конфігурації оточення при кожному запуску, зменшуючи обсяг ручної роботи та додаючи елемент непередбачуваності.

Логіка генерації інкапсульована у скрипті RandomGanarate.cs. Цей компонент має кілька публічних полів, які дозволяють гнучко налаштовувати процес в інспекторі Unity: numberObject визначає загальну кількість об'єктів для створення, а minRange та maxRange задають межі квадратної області, в якій відбуватиметься генерація. Основний алгоритм знаходиться у методі Generate(). При кожному його виклику з масиву префабів обирається випадковий елемент, створюється його екземпляр на сцені, після чого для

нього генерується унікальна позиція шляхом розрахунку випадкових координат X та Y у заданих межах. Програмна реалізація цього алгоритму наведена у Додатку А.

Результатом роботи цього скрипта є сцена, наповнена об'єктами, що розташовані хаотично, але в межах заданої території. Такий підхід створює ілюзію природного, нерукотворного ландшафту, на відміну від розміщення об'єктів по строгій сітці. Візуальним прикладом результату роботи такого алгоритму є розміщення точок у просторі, де кожна точка символізує позицію згенерованого об'єкта, демонструючи нерівномірний та органічний розподіл (рис.2.38).

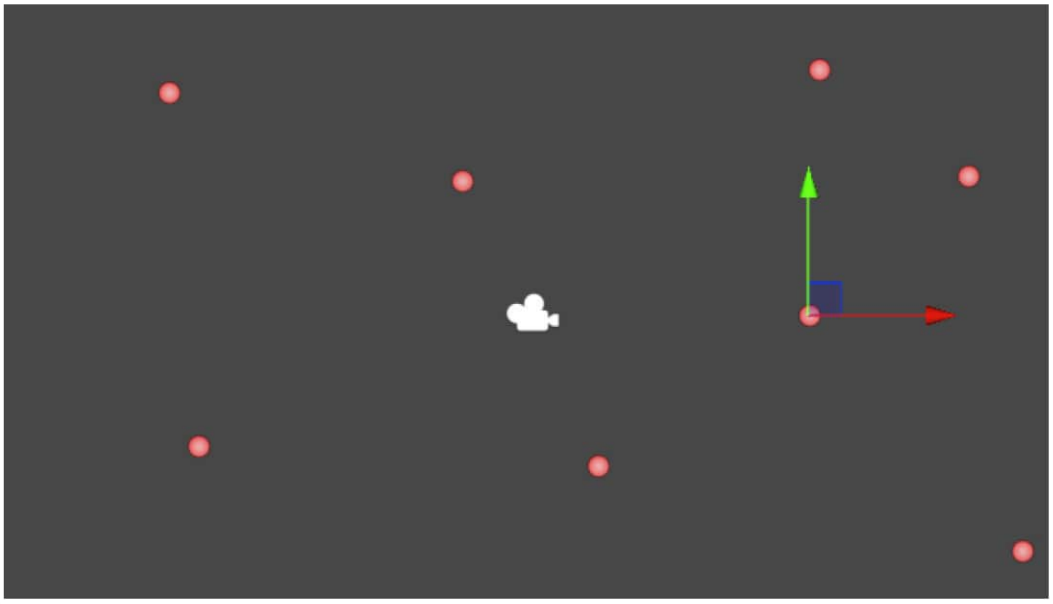


Рисунок 2.38 – Візуалізація результату випадкового розміщення об'єктів у просторі

## 2.8 Висновки по другому розділу

У другому розділі було спроєктовано та реалізовано ключові системи 2D-RPG ігрового застосунку з використанням інструментарію рушія Unity. У результаті виконання розділу:

1) Визначено загальну концепцію гри, її цільову аудиторію та ключові особливості, такі як процедурна генерація світу та динамічна система бою.
2) Створено та налаштовано головного героя, реалізовано механіку його пересування та адаптивного керування за допомогою пакета Input System, а також розроблено систему здоров'я та нанесення шкоди ворогам через Animation Events.
3) Розроблено динамічну систему поведінки ворожих AI-агентів, що включає скінченний автомат станів (FSM) для анімацій, логіку переслідування гравця, бойову механіку та повний життєвий цикл від появи до смерті.
4) Впроваджено систему навігації NavMesh для штучного інтелекту, адаптовано її для 2D-середовища за допомогою стороннього розширення та побудовано навігаційну сітку, що враховує перешкоди.
5) Реалізовано ігровий світ з використанням гібридного підходу: базовий ландшафт створено за допомогою системи Tilemap, а його наповнення деталями (декораціями) автоматизовано через алгоритм процедурної генерації.
6) Налаштовано динамічну камеру, що плавно слідує за гравцем, з використанням пакета Cinemachine, що покращує ігровий досвід та фокусує увагу на головному герої.
7) Сформовано архітектуру ігрових об'єктів, що передбачає розділення фізичної та логічної сутності від візуального представлення (на прикладі Hero/HeroVisual та Skeleton/SkeletonVisual).

# ВИСНОВКИ

В результаті виконання даної кваліфікаційної роботи було розроблено ігровий застосунок у жанрі 2D-RPG на рушії Unity. Ключову увагу було приділено реалізації двох основних аспектів, заявлених у темі: процедурної генерації ігрового світу для забезпечення високої реграбельності та динамічної системи поведінки ворогів на основі скінченних автоматів.

Розроблений прототип має завершену архітектуру, що включає систему керування головним героєм за допомогою пакета Input System, механіку дослідження світу, двосторонню бойову систему для гравця та ворогів, синхронізовану з анімаціями через Animation Events, а також інтелектуальну камеру на базі Cinemachine, що плавно слідує за персонажем.

У ході роботи було вирішено низку технічних завдань. Для навігації AI-агентів у 2D-середовищі стандартний інструмент AI Navigation було успішно адаптовано за допомогою стороннього розширення NavMeshPlus. Побудова рівня виконана за гібридним підходом, що поєднує ефективність системи Tilemap для створення ландшафту та гнучкість префабів для деталізації оточення. Автоматизація наповнення світу реалізована через алгоритм випадкового розміщення об'єктів.

Результатом роботи є функціональний прототип ігрового застосунку з цілісною структурою, налаштованими базовими механіками та продуманою архітектурою. Представлений проєкт демонструє успішне поєднання механік процедурної генерації та керованого станами штучного інтелекту в середовищі Unity і може слугувати міцною основою для подальшого розширення та створення повноцінного ігрового продукту.

# ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. Створення штучного інтелекту ворогів у іграх. Поради Senior. [Електронний ресурс] URL: https://gamedev.dou.ua/blogs/enemy-ai-in-games/ (дата звернення: 29.04.2025).
2. Вступ до процедурної генерації в Unity. [Електронний ресурс] URL: https://uk.sharpcoderblog.com/blog/an-introduction-to-procedural-generation-in-unity (дата звернення: 30.04.2025).
3. Процедурне генерування світу в Unity. [Електронний ресурс] URL: https://uk.sharpcoderblog.com/blog/world-generation-in-unity (дата звернення: 01.05.2025).
4. Hollow Knight. [Електронний ресурс] URL: https://store.steampowered.com/app/367520/Hollow_Knight/ (дата звернення: 02.05.2025).
5. Cave Story+. [Електронний ресурс] URL: https://store.nicalis.com/products/cave-story?variant=31806412914824 (дата звернення: 03.05.2025).
6. Disgaea: Hour of Darkness. [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Disgaea:_Hour_of_Darkness (дата звернення: 05.05.2025).
7. Hyper Light Drifter. [Електронний ресурс] URL: https://store.steampowered.com/app/257850/Hyper_Light_Drifter/ (дата звернення: 06.05.2025).
8. Dead Cells. [Електронний ресурс] URL: https://store.steampowered.com/app/588650/Dead_Cells/ (дата звернення: 07.05.2025).
9. Skeleton Sprite Pack Електронний ресурс] URL: https://jesse-m.itch.io/skeleton-pack (дата звернення: 10.05.2025)

10. Undertale. [Електронний ресурс] URL: https://store.steampowered.com/app/391540/Undertale/ (дата звернення: 08.05.2025).

11. Nightborne Warrior: 2D Character Sprites. [Електронний ресурс] URL: https://creativekind.itch.io/nightborne-warrior (дата звернення: 09.05.2025).
12. Natural Free Assets: Asset Pack for Tilemap. [Електронний ресурс] URL: https://sscary.itch.io/natural-free-assets (дата звернення: 10.05.2025).
13. NavMeshPlus: Unity NavMesh API for 2D. [Електронний ресурс] URL: https://github.com/h8man/NavMeshPlus (дата звернення: 11.05.2025).

# ДОДАТОК А. ВИХІДНИЙ КОД ОСНОВНИХ СКРИПТІВ

## А.1 Програмний код модуля Hero.cs

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

public class Hero : MonoBehaviour
{
    public static Hero Instance { get; private set; }

    [SerializeField] private float movingSpeed = 5f;

    private Rigidbody2D rb;
    private HeroVisual heroVisual;
    private HeroHealth heroHealth;

    private float minMovingSpeed = 0.1f;
    private bool isRunning = false;

    private void Awake() {
        if (Instance != null && Instance != this) {
            Destroy(gameObject);
            return;
```

```
    }
    Instance = this;

    rb = GetComponent<Rigidbody2D>();
    heroVisual = GetComponentInChildren<HeroVisual>();
    heroHealth = GetComponent<HeroHealth>();
}
private void Start() {
    if (GameInput.Instance != null) {
        GameInput.Instance.OnHeroAttack += GameInput_OnHeroAttack;
    }
}

private void OnDestroy() {
    if (GameInput.Instance != null) {
        GameInput.Instance.OnHeroAttack -= GameInput_OnHeroAttack;
    }
}

private void GameInput_OnHeroAttack(object sender, System.EventArgs e) {
    if (heroHealth != null && heroHealth.IsDead()) {
        return;
    }
    heroVisual.PlayAttackAnimation();
}
```

```
    private void FixedUpdate() {
        if (heroHealth != null && heroHealth.IsDead()) {
            rb.velocity = Vector2.zero;
            isRunning = false;
            return;
        }
        HandleMovement();
    }

    private void HandleMovement() {
        if (GameInput.Instance == null) return;

        Vector2 inputVector = GameInput.Instance.GetMovementVector();

        rb.MovePosition(rb.position + inputVector * (movingSpeed * Time.fixedDeltaTime));

        if (Mathf.Abs(inputVector.x) > minMovingSpeed || Mathf.Abs(inputVector.y) >
minMovingSpeed) {
            isRunning = true;
        } else {
            isRunning = false;
        }
    }

    public bool IsRunning() {
```

```
        return isRunning;
    }

    public Vector3 GetHeroScreenPosition() {
        if (Camera.main == null) return Vector3.zero;
        Vector3 heroScreenPosition =
Camera.main.WorldToScreenPoint(transform.position);
        return heroScreenPosition;
    }
}
```

**А.2 Програмний код модуля HeroHealth.cs**

```
using UnityEngine;

public class HeroHealth : MonoBehaviour
{
    [SerializeField] private int maxHealth = 5;
    private int currentHealth;
    private bool isDead = false;

    private Animator animator;

    private void Awake() {
        currentHealth = maxHealth;
        animator = GetComponentInChildren<Animator>();
    }
```

```
public void TakeDamage(int damage) {
    if (isDead) {
        return;
    }

    currentHealth -= damage;

    if (currentHealth <= 0) {
        Die();
    }
}

private void Die() {
    if (isDead) return;

    isDead = true;

    if (animator != null) {
        animator.SetTrigger("Dead");
    }

    Hero heroComponent = GetComponent<Hero>();
    if (heroComponent != null) {
        heroComponent.enabled = false;
```

```
        }
        HeroVisual heroVisualComponent = GetComponentInChildren<HeroVisual>();
        if (heroVisualComponent != null) {
            heroVisualComponent.enabled = false;
        }

        Collider2D coll = GetComponent<Collider2D>();
        if (coll != null) {
            coll.enabled = false;
        }

        Rigidbody2D rb = GetComponent<Rigidbody2D>();
        if (rb != null) {
            rb.velocity = Vector2.zero;
            rb.isKinematic = true;
        }
    }

    public bool IsDead() {
        return isDead;
    }
}
```

**А.3 Програмний код модуля HeroVisual.cs**

```
using System.Collections;
```

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

public class HeroVisual : MonoBehaviour
{
    private Animator animator;
    private SpriteRenderer spriteRenderer;

    private const string IS_RUNNING = "IsRunning";
    private const string ATTACK_TRIGGER = "Attack";

    public float attackRange = 1f;
    public int attackDamage = 1;
    public float knockbackForce = 5f;
    public LayerMask enemyLayer;

    private void Awake() {
        animator = GetComponent<Animator>();
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    private void Update() {
        if (Hero.Instance != null) {
            animator.SetBool(IS_RUNNING, Hero.Instance.IsRunning());
```

```
            AdjustHeroFacingDirection();
        }
    }

    private void AdjustHeroFacingDirection() {
        if (Camera.main == null || Mouse.current == null) {
            return;
        }
        Vector3 mouseWorldPosition =
Camera.main.ScreenToWorldPoint(Mouse.current.position.ReadValue());
        Vector3 heroPosition = transform.position;

        if (mouseWorldPosition.x < heroPosition.x) {
            spriteRenderer.flipX = true;
        } else {
            spriteRenderer.flipX = false;
        }
    }

    public void PlayAttackAnimation() {
        animator.SetTrigger(ATTACK_TRIGGER);
    }

    public void TriggerDamage() {
        Collider2D[] hitEnemies = Physics2D.OverlapCircleAll(transform.position,
attackRange, enemyLayer);
```

```
        foreach (Collider2D enemyCollider in hitEnemies) {
            EnemyHealth enemy = enemyCollider.GetComponent<EnemyHealth>();
            if (enemy == null) {
                enemy = enemyCollider.GetComponentInParent<EnemyHealth>();
            }

            if (enemy != null) {
                Vector2 direction = (enemy.transform.position - transform.position).normalized;
                enemy.TakeDamage(attackDamage, direction, knockbackForce);
            }
        }
    }

    private void OnDrawGizmosSelected() {
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
}
```

## A.4 Програмний код модуля EnemyAI.cs

```
using UnityEngine;
using UnityEngine.AI;

public class EnemyAI : MonoBehaviour
```

```
{
    [SerializeField] private float aggroRange = 5f;
    [SerializeField] private float attackRange = 1.5f;
    [SerializeField] private float roamInterval = 3f;
    [SerializeField] private float idleDuration = 2f;
    [SerializeField] private float minRoamRange = 4f;
    [SerializeField] private float maxRoamRange = 5f;

    private NavMeshAgent agent;
    private Transform target;
    private Vector3 basePosition;

    private enum State { Roaming, Idle, Chasing, Attacking, Dead }
    private State currentState;
    private float stateTimer;

    private Animator animator;
    private EnemyHealth enemyHealth;

    private void Awake() {
        agent = GetComponent<NavMeshAgent>();
        agent.updateRotation = false;
        agent.updateUpAxis = false;
        animator = GetComponentInChildren<Animator>();
        enemyHealth = GetComponent<EnemyHealth>();
```

```
    basePosition = transform.position;
    currentState = State.Roaming;
    stateTimer = roamInterval;
}

private void Update() {
    if (currentState == State.Dead || (enemyHealth != null && enemyHealth.IsDead())) {
        agent.isStopped = true;
        if (animator != null) animator.SetBool("IsMoving", false);
        return;
    }

    switch (currentState) {
        case State.Roaming:
            stateTimer -= Time.deltaTime;
            if (stateTimer <= 0f) {
                RoamRandomly();
                currentState = State.Idle;
                stateTimer = idleDuration;
                if (animator != null) animator.SetBool("IsMoving", true);
            }
            break;

        case State.Idle:
            stateTimer -= Time.deltaTime;
```

```
        if (animator != null) animator.SetBool("IsMoving", false);
        if (stateTimer <= 0f) {
            currentState = State.Roaming;
            stateTimer = roamInterval;
        }
        break;

    case State.Chasing:
        if (target == null) {
            currentState = State.Roaming;
            if (animator != null) animator.SetBool("IsMoving", false);
            agent.isStopped = false;
            return;
        }

        float distToTarget = Vector3.Distance(transform.position, target.position);

        if (distToTarget <= attackRange) {
            currentState = State.Attacking;
            agent.isStopped = true;
            if (animator != null) animator.SetBool("IsMoving", false);
        } else if (distToTarget <= aggroRange) {
            agent.isStopped = false;
            agent.SetDestination(target.position);
            FaceTarget();
```

```
            if (animator != null) animator.SetBool("IsMoving", true);
        } else {
            target = null;
            currentState = State.Roaming;
            agent.isStopped = false;
            if (animator != null) animator.SetBool("IsMoving", false);
        }
        break;

    case State.Attacking:
        if (target == null) {
            currentState = State.Roaming;
            if (animator != null) animator.SetBool("IsMoving", false);
            agent.isStopped = false;
            return;
        }

        distToTarget = Vector3.Distance(transform.position, target.position);

        if (distToTarget > attackRange) {
            currentState = State.Chasing;
            agent.isStopped = false;
            if (animator != null) animator.SetBool("IsMoving", true);
        } else {
            FaceTarget();
```

```
            }
            break;
        }
    }

    private void RoamRandomly() {
        Vector3 randomDirection = Random.insideUnitSphere *
Random.Range(minRoamRange, maxRoamRange);
        randomDirection += basePosition;
        randomDirection.y = transform.position.y;

        NavMeshHit hit;
        if (NavMesh.SamplePosition(randomDirection, out hit, maxRoamRange,
NavMesh.AllAreas)) {
            agent.SetDestination(hit.position);
            agent.isStopped = false;
        } else {
            agent.isStopped = true;
            if (animator != null) animator.SetBool("IsMoving", false);
            currentState = State.Idle;
            stateTimer = idleDuration;
        }
    }

    private void FaceTarget() {
        if (target == null) return;
```

```
        if (target.position.x < transform.position.x) {
            transform.localScale = new Vector3(-Mathf.Abs(transform.localScale.x),
transform.localScale.y, transform.localScale.z);
        } else {
            transform.localScale = new Vector3(Mathf.Abs(transform.localScale.x),
transform.localScale.y, transform.localScale.z);
        }
    }

    private void OnTriggerEnter2D(Collider2D other) {
        if (other.CompareTag("Player")) {
            target = other.transform;
            currentState = State.Chasing;
        }
    }

    private void OnTriggerExit2D(Collider2D other) {
        if (other.CompareTag("Player")) {
            if (other.transform == target) {
                target = null;
                currentState = State.Roaming;
                if (animator != null) animator.SetBool("IsMoving", false);
                agent.isStopped = false;
            }
        }
```

```
    }

    public void OnDeath() {
        currentState = State.Dead;
        if (agent != null) agent.isStopped = true;
        if (animator != null) animator.SetBool("IsMoving", false);
        enabled = false;
    }

    private void OnDrawGizmosSelected() {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, aggroRange);
        Gizmos.color = Color.red;
        Gizmos.DrawWireSphere(transform.position, attackRange);
    }
}
```

## А.5 Програмний код модуля EnemyHealth.cs

```
using System.Collections;
using UnityEngine;

public class EnemyHealth : MonoBehaviour
{
    [SerializeField] private int maxHealth = 3;
    private int currentHealth;
```

```
    private bool isDead = false;

    private Animator animator;
    private Rigidbody2D rb;
    private Collider2D coll;

    private EnemyCombat enemyCombat;
    private EnemyAI enemyAI;

    private void Awake() {
        currentHealth = maxHealth;
        animator = GetComponentInChildren<Animator>();
        rb = GetComponent<Rigidbody2D>();
        coll = GetComponent<Collider2D>();
        enemyCombat = GetComponent<EnemyCombat>();
        enemyAI = GetComponent<EnemyAI>();
    }

    public void TakeDamage(int damage, Vector2 knockbackDirection, float
knockbackForce) {
        if (isDead) return;

        currentHealth -= damage;

        if (rb != null) {
            if (!rb.isKinematic) {
```

```
            rb.AddForce(knockbackDirection * knockbackForce, ForceMode2D.Impulse);
        }
    }

    if (currentHealth <= 0) {
        Die();
    }
}

private void Die() {
    if (isDead) return;

    isDead = true;

    if (coll != null)
        coll.enabled = false;

    if (rb != null) {
        rb.velocity = Vector2.zero;
        rb.isKinematic = true;
    }

    if (animator != null) {
        animator.SetTrigger("Dead");
    }
```

```
    if (enemyCombat != null) {
        enemyCombat.Die();
    }
    if (enemyAI != null) {
        enemyAI.OnDeath();
    }

    StartCoroutine(DelayedDestroy());
}

private IEnumerator DelayedDestroy() {
    if (animator != null) {
        float clipLength = GetClipLength("Dead");
        if (clipLength > 0) {
            yield return new WaitForSeconds(clipLength + 0.1f);
        } else {
            yield return new WaitForSeconds(1f);
        }
    } else {
        yield return new WaitForSeconds(1f);
    }
    Destroy(gameObject);
}
```

```
    private float GetClipLength(string clipName) {
        if (animator == null || animator.runtimeAnimatorController == null) {
            return 0f;
        }

        foreach (AnimationClip clip in animator.runtimeAnimatorController.animationClips)
{
            if (clip.name == clipName) {
                return clip.length;
            }
        }
        return 0f;
    }

    public bool IsDead() {
        return isDead;
    }
}
```

## A.6 Програмний код модуля EnemyCombat.cs

```
using System.Collections;
using UnityEngine;

public class EnemyCombat : MonoBehaviour
{
    public float attackRange = 1.2f;
```

```
public float attackCooldown = 1.5f;
public int attackDamage = 1;
[SerializeField] private float attackZoneRadius = 0.5f;
[SerializeField] private Vector2 attackZoneOffset = new Vector2(0.5f, 0f);
public LayerMask playerLayer;

private Animator animator;
private Transform target;
private HeroHealth heroHealth;
private bool canAttack = true;
private EnemyAI enemyAI;
private EnemyHealth enemyHealthSelf;
private SpriteRenderer spriteRenderer;

private void Start() {
    animator = GetComponentInChildren<Animator>();
    enemyAI = GetComponent<EnemyAI>();
    enemyHealthSelf = GetComponent<EnemyHealth>();
    spriteRenderer = GetComponentInChildren<SpriteRenderer>();
}

private void Update() {
    if (enemyHealthSelf != null && enemyHealthSelf.IsDead()) {
        return;
    }
```

```
    if (target == null)
        FindTarget();

    if (target != null) {
        float dist = Vector2.Distance(transform.position, target.position);
        if (dist <= attackRange && canAttack) {
            StartCoroutine(Attack());
        }
    }
}

private void FindTarget() {
    GameObject player = GameObject.FindGameObjectWithTag("Player");
    if (player != null) {
        target = player.transform;
        heroHealth = player.GetComponent<HeroHealth>();
        if (heroHealth == null) {
            heroHealth = player.GetComponentInChildren<HeroHealth>();
            if (heroHealth == null) {
                heroHealth = player.GetComponentInParent<HeroHealth>();
            }
        }
    }
}
```

```
    private IEnumerator Attack() {
        canAttack = false;
        if (animator != null) animator.SetTrigger("Attack");
        yield return new WaitForSeconds(attackCooldown);
        canAttack = true;
    }

    public void DealDamage() {
        if (heroHealth == null) {
            return;
        }

        Vector2 currentAttackZoneOffset = attackZoneOffset;
        if (spriteRenderer != null && spriteRenderer.flipX) {
            currentAttackZoneOffset.x *= -1;
        }
        Vector2 attackOrigin = (Vector2)transform.position + currentAttackZoneOffset;
        Collider2D[] hitPlayers = Physics2D.OverlapCircleAll(attackOrigin,
attackZoneRadius, playerLayer);

        foreach (Collider2D playerCollider in hitPlayers) {
            HeroHealth targetHeroHealth = playerCollider.GetComponent<HeroHealth>();
            if (targetHeroHealth == null) {
                targetHeroHealth = playerCollider.GetComponentInParent<HeroHealth>();
            }
```

```
            if (targetHeroHealth != null && !targetHeroHealth.IsDead()) {
                targetHeroHealth.TakeDamage(attackDamage);
            }
        }
    }

    public void Die() {
        StopAllCoroutines();
        canAttack = false;
        enabled = false;
    }

    private void OnDrawGizmosSelected() {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, attackRange);
        Gizmos.color = Color.red;
        Vector2 currentAttackZoneOffset = attackZoneOffset;
        if (spriteRenderer != null && spriteRenderer.flipX) {
            currentAttackZoneOffset.x *= -1;
        }
        Vector2 gizmoAttackOrigin = (Vector2)transform.position +
currentAttackZoneOffset;
        Gizmos.DrawWireSphere(gizmoAttackOrigin, attackZoneRadius);
    }
}
```

## А.7 Програмний код модуля EnemyVisual.cs

```
using UnityEngine;


public class EnemyVisual : MonoBehaviour
{
    public void DealDamage() {
    }
}
```

## А.8 Програмний код модуля GameInput.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.InputSystem;


public class GameInput : MonoBehaviour
{

    public static GameInput Instance { get; private set; }


    private HeroInputActions heroInputActions;


    public event EventHandler OnHeroAttack;
```

```
    private void Awake() {
        Instance = this;

        heroInputactions = new HeroInputActions();
        heroInputActions.Enable();

        heroInputActions.Combat.Attack.started += HeroAttack_started;
    }

    private void HeroAttack_started(InputAction.CallbackContext obj) {
        OnHeroAttack?.Invoke(this, EventArgs.Empty);
    }

    public Vector2 GetMovementVector() {
        Vector2 inputVector = heroInputActions.Hero.Move.ReadValue<Vector2>();
        return inputVector;
    }

    public Vector3 GetMousePosition() {
        Vector3 mousePos = Mouse.current.position.ReadValue();
        return mousePos;
    }
}
```

## А.9 Програмний код модуля RandomGanarate.cs

```
using UnityEngine;


public class RandomGanarate : MonoBehaviour
{
   public GameObject[] grass;
   public int numberObject;
   private int generatedObject = 0;
   public float minRange, maxRange;


   void Update()
   {
      if (generatedObject < numberObject)
      {
         Generate();
         generatedObject++;
      }
   }


   public void Generate()
   {
      int rand = Random.Range(0, grass.Length);
      var cell = Instantiate(grass[rand], transform.position, Quaternion.identity);
      cell.transform.position = new Vector3(Random.Range(minRange, maxRange),
Random.Range(minRange, maxRange), transform.position.z);
   }
```

```
}
```

**А.10 Програмний код модуля GameHelper.cs**

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

namespace HeroAdventure.GameHelper
{

   public static class HeroHelper
   {
      public static Vector3 GetRandomDir() {
         return new Vector3(Random.Range(-1f, 1f), Random.Range(-1f, 1f)).normalized;
      }
   }

}
```