Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ГРА В ЖАНРІ «TOWER DEFENCE» НА БАЗІ UNITY З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ
GAME IN THE GENRE OF «TOWER DEFENSE» BASED ON UNITY WITH THE USE OF ARTIFICIAL INTELLIGENCE

спеціальність 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

освітня програма «Інженерія програмного забезпечення»
(назва освітньої програми)

Виконав: здобувач вищої освіти
Групи ІПЗ-41
Качковський Артем Андрійович
_____
(підпис)

Керівник:
к.т.н., доцент
Ящук Андрій Анатолійович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«01» 06 2023 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна
_____
(підпис)

Луцьк – 2023 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра інженерії програмного забезпечення
Ступінь вищої освіти: *бакалавр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *121 Інженерія програмного забезпечення*
Освітня програма: «*Інженерія програмного забезпечення*»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«02» 02 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Качковський Артем Андрійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: Гра в жанрі «Tower Defence» на базі Unity з використанням штучного інтелекту

Керівник роботи: к.т.н., доцент Ящук Андрій Анатолійович
затверджені наказом закладу вищої освіти від «*23*» *грудня 2022* р. № *961-01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи «*9*» *червня 2023 р.*
3. Вихідні дані до роботи: *технічне завдання, технічне та програмне забезпечення, вимоги до розробки програмного забезпечення, ергономічні вимоги до функціонування програмного засобу, Мова програмування C#*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Актуальність використання штучного інтелекту у ігральних програмах. Мета роботи та галузь застосування. Аналіз, визначення вимог до розроблюваної гри та проектування програмного забезпечення. Реалізація гри з використанням штучного інтелекту. Тестування та валідація гри. Висновки.
5. Перелік графічного матеріалу:
5 рисунків

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Ящук А. А.*		
Специфікація вимог до розробленої системи	*Ящук А. А.*		
Розробка об'єкта проектування	*Ящук А. А.*		
Нормоконтроль	*Ящук А. А.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	2,82%		
Академічна доброчесність	*Ящук А.А.*		

7. Дата видачі завдання *«2» лютого 2023 р.*

КАЛЕНДАРНИЙ ПЛАН

№ З/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Огляд літературних джерел по темі кваліфікаційної роботи бакалавра*	*05.02.2023 р.*	вик.
2	*Аналіз проблеми розробки та впровадження об'єкту проектування*	*27.02.2023 р.*	вик.
3	*Обґрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання*	*10.03.2023 р.*	вик
4	*Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних*	*17.04.2023 р.*	вик.
5	*Практична реалізація об'єкта проектування та розробка бази даних*	*18.05.2023 р.*	вик
6	*Тестування та налагодження об'єкта проектування*	*01.06.2023*	вик.
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*09.06.2023 р.*	вик.

Здобувач вищої освіти _____ *Качковський А. А.*
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ *Ящук А.А.*
(підпис) (прізвище, ініціали)

Міністерство освіти і науки України
Луцький національний технічний університет

Рецензія
на кваліфікаційну роботу

Здобувач вищої освіти: *Качковський Артем Андрійович*
Тема: *«Гра в жанрі «Tower Defence» на базі Unity з використанням штучного інтелекту».*

Коротка характеристика кваліфікаційної роботи та прийнятих рішень:
Кваліфікаційна робота бакалавра складається з трьох розділів, в яких проаналізовані проблематика та поставлені завдання за темою роботи, здійснено теоретичне дослідження та практичну реалізацію гри на базі Unity з елементами штучного інтнелекту

Самостійні розробки і пропозицій автора: *Автор самостійно розробив гру в жанрі «Tower Defence» на Unity з еоементами штучного інтелекту.*

Практичне значення роботи *полягає в теоретичному обґрунтуванні методів штучного інтелекту в комп'ютернх іграх і їх впровадженні в розробленій комп'ютерній грі.*

Недоліки: *Істотних недоліків в роботі не виявлено.*

Загальний висновок: *У кваліфікаційній роботі бакалавра наведені результати вирішення актуального завдання використання штучного інтелекту в іграх. Робота є результатом самостійного наукового дослідження. Істотних недоліків не виявлено. Кваліфікаційна робота здобувача освіти Качковського Артема Андрійовича рекомендується до захисту екзаменаційній комісії та заслуговує оцінки «відмінно».*

Рецензент: Савчин Павло Вікторович
(прізвище, ім'я, по-батькові, посада)

Рецензент кваліфікаційної роботи _____
(підпис)

«08» червня 2023 р.

АНОТАЦІЯ

Качковський А. А. Гра в жанрі «Tower Defence» з використанням штучного інтелекту. Рукопис.

Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності Інженерії програмного забезпечення. Луцький національний технічний університет. Луцьк, 2023.

Робота присвячена розробці комп'ютерної гри в жанрі «Tower Defence» з використанням штучного інтелекту. Метою роботи є створення геймплею, який використовує алгоритми штучного інтелекту для управління ворожими силами та стратегічним плануванням гравця.

У розділі аналізу та визначення вимог до розроблюваної гри розглядається актуальність використання штучного інтелекту у гральних програмах, а також проводиться аналіз існуючих гравців на ринку та визначаються функціональні та нефункціональні вимоги до гри.

У наступному розділі розглядається проектування програмного забезпечення гри, включаючи архітектуру гри та інтеграцію штучного інтелекту. Особлива увага приділяється розробці моделі поведінки ворожих сил та оборонних веж, а також імплементації алгоритмів штучного інтелекту.

Після реалізації гри проводиться тестування та валідація, в ході яких перевіряється коректність функціонування гри та вплив штучного інтелекту на геймплей та користувацький досвід. Отримані результати використовуються для корекції та вдосконалення гри.

Ключові слова: гра, штучний інтелект, програмне забезпечення, Unity, ML-agents.

ANNOTATION

Kachkovskyi A. A. Game in the «Tower Defence» genre using artificial intelligence. Manuscript.

Bachelor's qualification work of the educational program «Software Engineering» in the specialty of Software Engineering. Lutsk National Technical University. Lutsk, 2023.

The work is dedicated to the development of a computer game in the «Tower Defence» genre using artificial intelligence. The aim of the work is to create gameplay that utilizes artificial intelligence algorithms to control enemy forces and strategic planning for the player.

The analysis section discusses the relevance of using artificial intelligence in gaming applications, analyzes existing players in the market, and identifies the functional and non-functional requirements for the game.

The next section focuses on the design of game software, including game architecture and integration of artificial intelligence. Special attention is given to the development of the behavior model of enemy forces and defensive towers, as well as the implementation of artificial intelligence algorithms.

After the game is implemented, testing and validation are conducted to verify the correctness of the game's functioning and the impact of artificial intelligence on gameplay and user experience. The obtained results are used for refinement and enhancement of the game.

Keywords: game, artificial intelligence, software, Unity, ML-agents.

ЗМІСТ

ВСТУП

У сучасному світі інформаційні технології є невід’ємною частиною нашого повсякденного життя. Однією з популярних областей розробки програмного забезпечення є створення комп’ютерних ігор. Це захоплююче творче завдання, яке вимагає поєднання програмування, графічного дизайну, ігрового дизайну та інших компонентів.

Метою цієї кваліфікаційної роботи бакалавра є розробка гри «Tower Defence» з використанням штучного інтелекту, з використанням платформи Unity, Zenject та ML-Agents. Головною метою розробки гри є створення захоплюючого інтерактивного досвіду для користувачів та дослідження можливостей штучного інтелекту в гральних додатках.

Об’єктом дослідження є розробка гри "Tower Defence".

Предметом дослідження є використання штучного інтелекту для керування персонажами в цій грі.

Завдання на роботу:

- розробка основної структури гри та інтерфейсу користувача: створення архітектури гри та управління рівнями;
- інтеграція Universal Render Pipeline (URP) для візуального оформлення гри;
- використання ML-Agents для навчання моделей штучного інтелекту;
- використання Zenject для організація коду та керування залежностями;
- реалізація системи заклинань для захисту порталу: розробка механіки заклинань та їх різноманітності;
- впровадження системи збереження прогресу гри;
- тестування та налагодження гри для стабільної роботи.

Оцінка сучасного стану проблеми

На сьогоднішній день існує велика кількість ігор на ринку, але розробка гри з використанням штучного інтелекту є актуальним та викликаючим завданням. Споживачі все більше цікавляться іграми, які мають інтелектуальні

алгоритми та можуть адаптуватися до їх дій. Штучний інтелект в гральних додатках відкриває нові можливості для створення складних ігрових сценаріїв, інноваційного геймплею та взаємодії з гравцем.

Світові тенденції вирішення поставлених задач

На сучасному гейм-ринку існує кілька ігрових движків, які надають потужні інструменти для розробки комп'ютерних ігор. Один з найпопулярніших ігрових движків – Unity, який забезпечує широкі можливості для створення ігрових проектів різного рівня складності.

У цій роботі для реалізації гри «Tower Defence» використовується фреймворк Zenject, який забезпечує реалізацію інверсії керування та управління залежностями в програмі.

Для досягнення інтелектуального поведінки в грі, використовується платформа ML-Agents, розроблена компанією Unity Technologies. ML-Agents дозволяє навчати агентів гри за допомогою методів машинного навчання, що дозволяє створювати складні ігрові сценарії з інтелектуальним поведінкою.

Взаємозв'язок з іншими роботами

Результати цієї роботи базуються на попередніх дослідженнях та розробках в галузі розробки комп'ютерних ігор, штучного інтелекту та використання платформи Unity. Попередні дослідження та практичні розробки в цій галузі стали підґрунтям для розвитку цього проекту та визначення його унікальних особливостей.

РОЗДІЛ 1
АНАЛІЗ РОЗРОБКИ КОМП'ЮТЕРНИХ ІГОР З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

У сучасному ігровому середовищі можна знайти кілька ігор в жанрі «Tower Defence», але небагато з них використовують штучний інтелект для керування ворогами. Більшість існуючих ігор в цьому жанрі використовують визначені шаблони поведінки для персонажів, що обмежує варіативність та непередбачуваність геймплею. Такий підхід може призвести до стереотипного та прогнозованого розвитку подій у грі, що зменшує інтерес користувачів до ігор цього жанру.

Однак, за останні роки з'явилися певні прориви в галузі використання штучного інтелекту в іграх. Завдяки розвитку технологій машинного навчання, стало можливим створення навчених моделей, які можуть самостійно набувати досвіду та вдосконалювати свою поведінку на основі взаємодії з грою. Це дає можливість розширити можливості інтелектуального керування ворогами у грі.

Створення нової системи, яка використовуватиме навчені моделі штучного інтелекту є доцільним та має кілька переваг. Перш за все, це дозволить створити більш реалістичний та непередбачуваний геймплей, оскільки вороги будуть виявляти нові стратегії та адаптуватися до дій гравця. Це забезпечить більшу глибину та варіативність гри, що сприятиме підвищенню інтересу користувачів та продовженню тривалості геймплею.

Додатковою перевагою створення системи з використанням штучного інтелекту є можливість автоматичної оптимізації поведінки ворогів. Моделі штучного інтелекту можуть адаптуватися до різних умов та розуміти, які дії є найбільш ефективними в конкретних ситуаціях. Це дозволить досягти балансу у геймплеї та підвищити складність гри, роблячи її викликом для гравців різного рівня.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

В даному пункті сформульовано конкретні цілі кваліфікаційної роботи, які повинні бути досягнуті під час розробки гри в жанрі «Tower Defence» на базі Unity з використанням штучного інтелекту. Завдання включають:

– розробку основної структури гри та інтерфейсу користувача: створення базової архітектури гри управління рівнями та інші важливі компоненти; розробка інтуїтивного та зручного інтерфейсу користувача, який забезпечить зручну навігацію та взаємодію гравця з грою;

– інтеграцію Universal Render Pipeline (URP) для створення візуального оформлення гри: використання URP для створення візуально привабливого та оптимізованого візуального стилю гри; налаштування освітлення, матеріалів та ефектів;

– використання ML-Agents для навчання моделей штучного інтелекту для керування персонажами в грі: інтеграція ML-Agents, фреймворку штучного інтелекту в Unity; налаштування середовища та параметрів навчання, щоб досягти оптимальної ефективності; навчання моделей;

– використання Zenject для розробки та керування скриптами гри: використання Zenject, фреймворку інверсії керування та впровадження залежностей, для організації коду та керування скриптами гри; забезпечення модульності та розширюваності системи;

– реалізацію системи заклинань, які гравець може використовувати для захисту порталу: розробка механіки заклинань, яка дозволить гравцю активно взаємодіяти з грою та впливати на хід битви; реалізація заклинань з унікальними властивостями та ефектами для забезпечення різноманітності геймплею;

– впровадження системи збереження прогресу гри. розробка механізму збереження та завантаження прогресу гравця;

– тестування та налагодження гри для забезпечення стабільної роботи.

Висновки до розділу 1

Аналізуючи сучасний стан проблеми, можна зробити висновок, що використання штучного інтелекту в іграх жанру «Tower Defence» має великий потенціал для покращення геймплею. Багато існуючих ігор в цьому жанрі обмежені варіативністю та непередбачуваністю через використання статичних шаблонів поведінки для ворогів. Застосування навчених моделей штучного інтелекту дозволить створити більш реалістичний та захоплюючий геймплей, де вороги будуть адаптуватися до дій гравця і виявляти нові стратегії.

Були сформульовані конкретні завдання, які включають розробку основної структури гри та інтерфейсу користувача, використання Universal Render Pipeline для візуального оформлення, інтеграцію ML-Agents для навчання моделей штучного інтелекту, використання Zenject для керування скриптами гри, реалізацію системи заклинань для гравця, впровадження системи збереження прогресу гри та тестування та налагодження геймплею.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення

Функціональні вимоги

Головне меню. Відображення головного меню зі списком доступних опцій. Можливість почати нову гру. Можливість завершити гру і вийти до головного меню. Можливість налаштувати параметри гри (складність, налаштування звуку тощо). Можливість переглянути інструкцію гри.

Гра. Створення гри з випадковим розміщенням ворогів та об’єктів. Відображення головного ігрового інтерфейсу з необхідною інформацією (рахунок, кількість доступних заклинань тощо). Поява нових хвиль ворогів з певним інтервалом. Рух ворогів залежно від їхнього штучного інтелекту. Використання заклинань гравцем для атаки ворогів або підтримки порталу. Завершення гри при досягненні певної умови (наприклад, втрата всього здоров’я порталу або виконання певного завдання).

Налаштування. Можливість змінити налаштування гри (складність, налаштування звуку тощо). Можливість вибрати рівень складності гри (легкий, середній, важкий). Можливість налаштувати параметри графіки (роздільна здатність, якість зображення тощо). Можливість змінити налаштування звуку (гучність звуку, включення/вимкнення звуків тощо).

Інструкція. Відображення інструкції гри, яка пояснює правила та механіку гри. Пояснення основних елементів графічного інтерфейсу та їх функціональності.

Нефункціональні вимоги

Продуктивність. Гра повинна працювати плавно без помітних затримок або лагів. Завантаження рівнів та переходи між ними мають бути швидкими.

Сумісність. Гра повинна бути сумісною з різними платформами, зокрема Windows, macOS, Linux. Вигляд та функціональність гри мають бути адаптованими до різних розмірів екрану.

Візуальний дизайн. Гра повинна мати привабливий та естетичний візуальний дизайн. Використання Universal Render Pipeline (URP) для забезпечення високої якості рендерингу та графічних ефектів.

Безпека. Гра повинна забезпечувати безпеку даних гравців, зокрема шляхом захисту конфіденційної інформації.

Штучний інтелект. Вороги гри повинні мати ефективний штучний інтелект, який дозволяє їм приймати розумні рішення та адаптуватися до ситуацій у грі. Використання ML-Agents для навчання моделей.

Мова програмування та інструменти. Використання мови програмування C# для реалізації логіки гри та взаємодії об’єктів. Використання Unity для розробки гри та його вбудованих можливостей. Використання Zenject для керування залежностями та ін’єкції залежностей в скриптах гри. Використання Rider як інтегрованого середовища написання коду.

Тестування. Проведення тестування гри для перевірки функціональності та виявлення можливих помилок та дефектів. Забезпечення якості програмного забезпечення шляхом систематичного тестування та виправлення виявлених проблем.

Вибір структурної моделі програмної системи

Обґрунтування вибору певного типу моделі для виконання завдання з моделювання програмної системи та її елементів є важливим кроком у процесі розробки програмного забезпечення. В даному випадку було обрано структурну модель програмної системи.

Структурна модель дозволяє відображати складові системи та їх взаємозв’язки, що є важливим для аналізу та розуміння структури системи. Застосування структурної моделі дозволяє виявити проблеми проектування, зробити оптимізації та забезпечити кращу організацію програмного коду. У даному випадку, використання структурної моделі допоможе аналізувати та

представляти компоненти, модулі, класи, інтерфейси та залежності між ними, що допоможе забезпечити легкість розробки та підтримки програмної системи.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Детальніше розглянемо обрані засоби, методи та алгоритми для розробки програмного забезпечення ігрової системи.

Unity

Unity є одним з найпопулярніших інтегрованих середовищ розробки ігор у світі. Він володіє потужними засобами, які дозволяють розробникам створювати ігри з великою кількістю деталей та ефектів. Unity підтримує різні платформи, такі як Windows, macOS, Android, iOS, PlayStation і Xbox, що дозволяє розгортати гру на різних пристроях. Розробники можуть використовувати Unity для створення як простих, так і складних ігор завдяки його зручному інтерфейсу та широкому спектру можливостей [1].

Unity надає розробникам багато інструментів і ресурсів, які спрощують розробку ігор. Воно має вбудований редактор сцен, де розробники можуть створювати та редагувати візуальні компоненти гри, такі як об’єкти, персонажі, ландшафти та інше. Unity також підтримує роботу з 3D-моделями та анімаціями, дозволяючи розробникам створювати реалістичні світи та рухомі об’єкти. За допомогою Unity, розробники можуть також додавати звукові ефекти, музику та інші аудіовізуальні компоненти до своїх ігор.

Unity підтримує різні мови програмування, але одним з головних компонентів є мова програмування C#. Ця мова є основною мовою програмування, яку підтримує Unity, і надає розробникам багато можливостей для розробки ігрової логіки. C# є потужною і ефективною мовою, яка підтримує об’єктно-орієнтоване програмування (ООП). Це дозволяє розробникам створювати класи, об’єкти та використовувати успадкування та поліморфізм, що сприяє структурованості і організованості коду. Використання

C# дозволяє розробникам створювати чистий і організований код, що полегшує підтримку та розширення програмної системи.

Unity надає інтеграцію з багатьма іншими інструментами та ресурсами, що полегшують розробку ігор. Наприклад, воно має вбудовану підтримку фізичного моделювання, що дозволяє розробникам створювати реалістичну фізику для об’єктів у грі. Unity також підтримує інтеграцію з різними двигунами штучного інтелекту, що дозволяє створювати персонажів зі складною поведінкою та реалістичною штучною інтелектуальною системою.

Однією з найбільших переваг Unity є його активна спільнота розробників. Unity має велику базу знань, форуми, онлайн-курси та документацію, що дозволяє розробникам швидко отримувати допомогу та підтримку в процесі розробки. Спільнота Unity також активно ділиться знаннями, розробляє власні ресурси та створює розширення, що полегшують розробку ігор.

URP

Universal Render Pipeline (URP) є графічним рушієм, який входить до складу Unity. URP надає розробникам можливості рендерингу графіки з високою якістю, включаючи освітлення, тіні, ефекти та анімацію. За допомогою URP розробники можуть створювати привабливу візуальну графіку, яка покращує враження гравців від гри. URP також забезпечує оптимізацію рендерингу, що дозволяє досягти високої продуктивності навіть на менш потужних пристроях [2].

URP має багато особливостей, які роблять його потужним інструментом для розробників ігор. Воно підтримує різні види освітлення, такі як точкове освітлення, напрямне освітлення, плямисте освітлення та інше. Розробники можуть контролювати колір, інтенсивність та напрям освітлення для створення потрібного настрою у грі. URP також надає можливості рендерингу тіней, включаючи м’які тіні, тіні від прозорих об’єктів та динамічні тіні. Це дозволяє створювати реалістичні та деталізовані сцени у грі.

Окрім основних графічних ефектів, URP також підтримує спеціальні ефекти, такі як блискітки, розмиття, кольорові фільтри та інші. Ці ефекти дозволяють розробникам створювати унікальні візуальні стилі та ефекти, що надають грі особливу атмосферу. URP також має підтримку шейдерів, що дозволяє розробникам контролювати вигляд та поведінку матеріалів у грі. Так, наприклад, можна створити шейдер власноруч з унікальними властивостями (рис. 2.1).

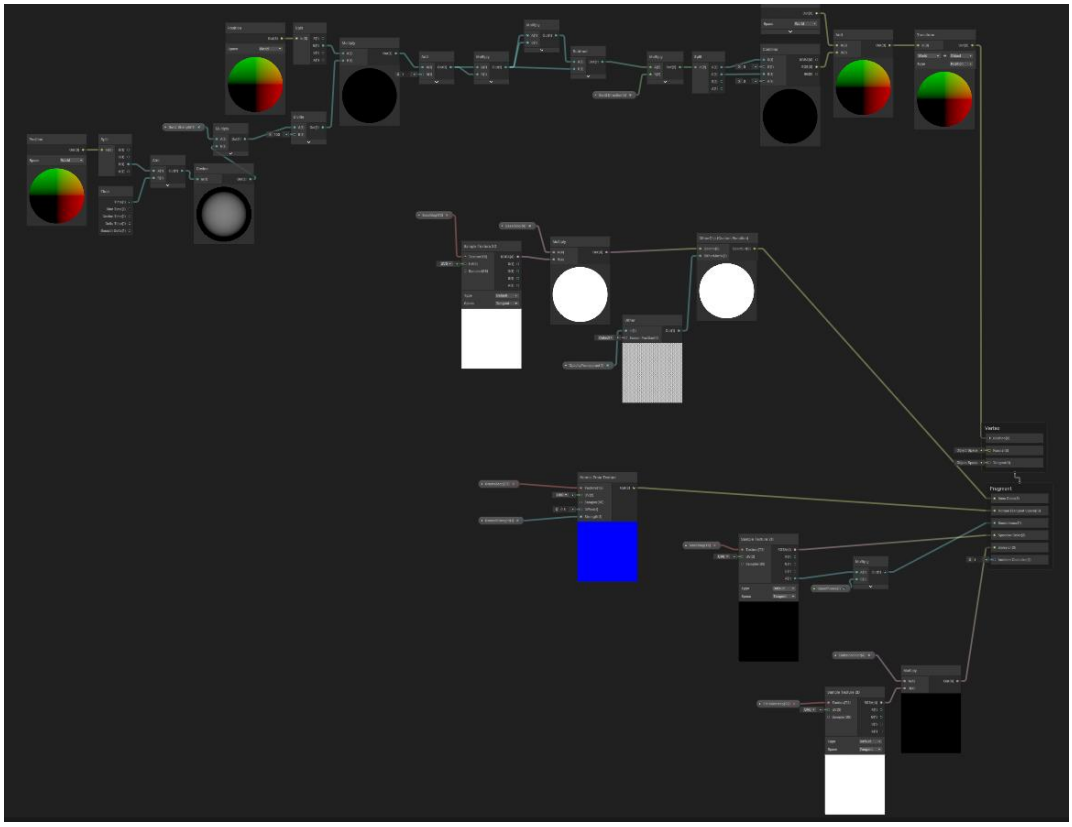

Рисунок 2.1 – Приклад шейдеру URP з гри

URP має переваги над іншими графічними рушіями завдяки своїм оптимізаційним можливостям. Воно працює ефективно навіть на менш потужних пристроях, дозволяючи розробникам створювати графічно насичені ігри, які працюють плавно та мають високу продуктивність. URP також має підтримку мобільних платформ, що дозволяє розробникам створювати ігри для смартфонів та планшетів з високою якістю графіки.

ML-Agents

ML-Agents є потужною бібліотекою для навчання, розробленою Unity, яка дозволяє розробникам створювати штучний інтелект для ворогів та визначати їх поведінку на основі навчання. Ця бібліотека відкриває безліч можливостей для розширення геймплею та створення захоплюючих інтерактивних можливостей для гравців.

ML-Agents базується на ідеї навчання з підсиленням, яка полягає в тому, щоб навчити агента приймати рішення у віртуальному середовищі, отримуючи зворотний зв’язок у вигляді нагород та покарань. Цей процес навчання дає агенту змогу виробити навички та стратегії, що дозволяють йому досягати поставлених цілей. Завдяки ML-Agents розробники можуть створювати ворогів зі складною поведінкою, які здатні адаптуватися до дій гравця, роблячи ігровий процес більш цікавим [3].

Основними компонентами ML-Agents є агенти та середовище. Агенти - це об’єкти у грі, які взаємодіють з середовищем та вчаться на основі своїх дій. Середовище надає агентам віртуальне просторове середовище, в якому вони діють та взаємодіють з об’єктами та іншими агентами. Це може бути симуляція реального світу або фантастична ігрова локація, як, наприклад на рисунку 2.2.

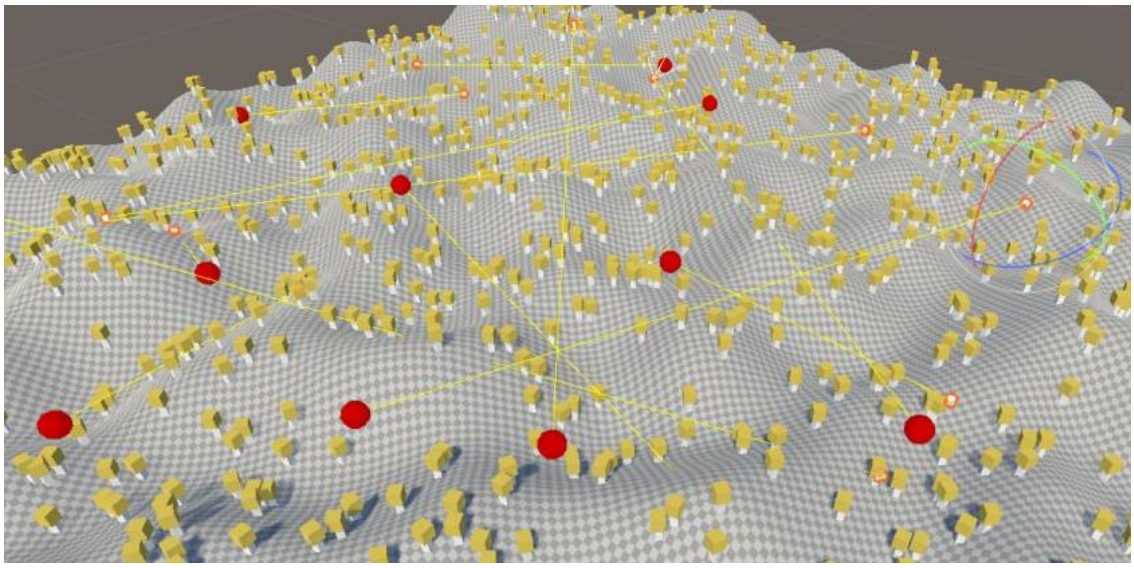

Рисунок 2.2 – Локація для навчання агента «BirdAgent» з гри

ML-Agents надає широкий набір інструментів та алгоритмів для тренування агентів. Розробники можуть використовувати готові алгоритми, такі як Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC) та Deep Q-Networks (DQN), або створити свої власні алгоритми, які відповідають конкретним потребам їхньої гри. Це дозволяє досягти гнучкості та налаштування навчання з підсиленням для досягнення бажаної поведінки агентів [4].

Крім того, ML-Agents дозволяє розробникам створювати складні системи штучного інтелекту, використовуючи не тільки алгоритми навчання з підсиленням, але й генетичні алгоритми, еволюційні стратегії та інші підходи. Це дає можливість експериментувати з різними методами навчання та знаходити оптимальні рішення для досягнення бажаної поведінки агентів у грі.

Застосування ML-Agents у розробці ігор відкриває широкі перспективи для створення інноваційних та захоплюючих геймплеїв. Розробники можуть експериментувати з різними типами агентів, їхніми взаємодіями та стратегіями, створюючи цікаві та непередбачувані ігрові сценарії. Комбінування ML-Agents з іншими інструментами Unity відкриває безліч можливостей для творчості та інновацій у галузі розробки ігор.

Zenject

Zenject є потужним інструментом для управління залежностями і внедрення залежностей у програмуванні. Він надає розробникам зручність і гнучкість у створенні і керуванні об’єктами та їх залежностями у складних програмних проектах.

Однією з ключових особливостей Zenject є його здатність автоматично вирішувати проблему залежностей між об’єктами. Розробникам не потрібно вручну вказувати, які об’єкти залежать від інших, Zenject автоматично визначає ці залежності на основі конфігурації, що встановлюється розробником. Це дозволяє легко створювати і змінювати структуру проекту, не занурюючись в деталі управління залежностями.

Ще однією важливою функцією Zenject є його здатність розрізняти різні типи створення залежностей. Він підтримує ін’єкцію залежностей через конструктори, властивості та методи, що надає розробникам гнучкість у виборі найбільш зручного підходу для їхніх потреб. Zenject також підтримує різні способи життєвого циклу об’єктів, дозволяючи розробникам контролювати моменти створення, знищення та перезавантаження об’єктів.

Висновки до розділу 2

У даному розділі було проведено аналіз, визначення вимог до розроблюваної системи та проектування програмного забезпечення для гри в жанрі «Tower Defence». Було обґрунтовано вибір певного типу моделі для моделювання програмної системи та її елементів. Були складені специфікація вимог до ПЗ. Використання Unity, C#, URP та ML-Agents надає необхідні інструменти для моделювання графіки, реалізації інтелектуальної поведінки та створення захопливої геймплею

РОЗДІЛ 3
РОЗРОБКА ГРИ В ЖАНРІ «TOWER DEFENCE» НА БАЗІ UNITY З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

3.1 Практична реалізація об’єкта проектування

Фрагмент коду, який наведений нижче, відображає використання Zenject для налаштування залежностей в проекті на Unity (лістинг 3.1).

Лістинг 3.1 – Код класу «GameInstallers»

```
[CreateAssetMenu(fileName = «GameInstallers», menuName =
«Installers/GameInstallers»)]
public class GameInstallers : ScriptableObjectInstaller<GameInstallers>
{
    [SerializeField] private GameUIView _view;
    public override void InstallBindings()
    {

Container.Bind<IProjectCoroutineManager>().To<ProjectCoroutineManager>().AsSingl
e();

Container.Bind<GameUIView>().FromComponentInNewPrefab(_view).UnderTransformGroup
(«UI»).AsSingle().NonLazy();
        Container.Bind<GameController>().AsSingle().NonLazy();

Container.Bind<InstantiatorView>().FromNewComponentOnNewGameObject().AsSingle();
    }
}
```

кінець лістингу 3.1

Цей код представляє клас GameInstallers, який успадковує від ScriptableObjectInstaller<GameInstallers>. ScriptableObjectInstaller є базовим класом, наданим Zenject, і він використовується для налаштування залежностей.

Атрибут [CreateAssetMenu(fileName = «GameInstallers», menuName = «Installers/GameInstallers»)] дозволяє створити новий об’єкт GameInstallers через меню Unity.

У методі InstallBindings виконується налаштування залежностей. Ось опис тих, що використовуються в цьому коді:

Container.Bind<IProjectCoroutineManager>().To<ProjectCoroutineManager>().AsSingle(): Цей рядок встановлює зв’язок між інтерфейсом

IProjectCoroutineManager та його реалізацією ProjectCoroutineManager. За допомогою To<ProjectCoroutineManager>(), Zenject знає, яку реалізацію використовувати для інтерфейсу. AsSingle() вказує, що реалізація буде відповідати лише одному екземпляру протягом життєвого циклу залежності.

Container.Bind<GameUIView>().FromComponentInNewPrefab(_view).UnderTransformGroup(«UI»).AsSingle().NonLazy(): Цей рядок встановлює зв’язок для GameUIView. За допомогою FromComponentInNewPrefab(_view), Zenject створює новий префаб, використовуючи компонент _view. UnderTransformGroup(«UI») вказує, що префаб буде доданий під групу з іменем «UI». AsSingle() вказує, що буде створений лише один екземпляр GameUIView. NonLazy() вказує, що ця залежність повинна бути встановлена під час ініціалізації контейнера.

Container.Bind<GameController>().AsSingle().NonLazy(): Цей рядок встановлює залежність для GameController. AsSingle() вказує, що буде створений лише один екземпляр GameController. NonLazy() вказує, що ця залежність повинна бути встановлена під час ініціалізації контейнера.

Container.Bind<InstantiatorView>().FromNewComponentOnNewGameObject().AsSingle(): Цей рядок встановлює залежність для InstantiatorView. FromNewComponentOnNewGameObject() вказує, що буде створений новий об’єкт з компонентом InstantiatorView. AsSingle() вказує, що буде створений лише один екземпляр InstantiatorView.

Цей код демонструє, як використовувати Zenject для налаштування залежностей в Unity проекті. Клас GameInstallers відповідає за конфігурацію контейнера залежностей та встановлення потрібних залежностей для різних компонентів гри. Завдяки Zenject, залежності можуть бути легко внедрені в інші класи шляхом конструктора або властивостей.

Наступний клас EnemySpawnerController, який відповідає за створення та управління ворогами в грі (лістинг 3.2).

Лістинг 3.2 – Код класу «EnemySpawnerController»

```
public class EnemySpawnerController
{
    private readonly EnemySpawnerDatabase _database;
    private readonly EnemyPrefabDatabase _prefabDatabase;
    private readonly LocationDatabase _locations;
    private readonly GameUIView _ui;
    private readonly IProjectCoroutineManager _coroutineManager;
    private readonly InstantiatorView _instantiator;
    private event Action OnWin;
    private event Action OnLose;
/…
    public EnemySpawnerController(EnemySpawnerDatabase database,
EnemyPrefabDatabase prefabDatabase, LocationDatabase locations, GameUIView ui,
IProjectCoroutineManager coroutineManager, InstantiatorView instantiator)
    {
        _database = database;
        _prefabDatabase = prefabDatabase;
        _locations = locations;
        _coroutineManager = coroutineManager;
        _instantiator = instantiator;
        _ui = ui;
        _enemies = new List<EnemyView>();
    }

    public void StartRound(int round, ELocation target)
    {
        _lose = false;
        _round = round;
        _target = _locations.GetData(target).Position;
        _roundData = _database.GetData(round);
        DestroyAllEnemies();
        _enemies = new List<EnemyView>();
        _ui.SetInfo($»<b>Start round {round}</b>«);
        _coroutineManager.StartCoroutine(SpawnWave());
    }
    IEnumerator SpawnWave()
    {
        yield return new WaitForSeconds(5);
        _wave = 1;
        foreach (var data in _roundData.Datas)
        {
            _ui.SetInfo($»<i>Wave {_wave}</i>«);
            _ui.SetWave(_wave);
            _wave++;
            for (int i = 0; i < data.Count; i++)
            {
                if (Random.Range(0,1f) <= data.Chance)
                {
                    var location =
_locations.GetData(data.Locations[Random.Range(0, data.Locations.Count)]);
                    var enemy
=_instantiator.InstantiateEnemyView(_prefabDatabase.GetPrefab(data.Type),
location.Position, location.Rotation);
                    enemy.Target = _target;
                    enemy.OnDead += OnDead;
                    enemy.OnFinish += OnFinish;
                    _enemies.Add(enemy);
                }
            }
            yield return new WaitForSeconds(data.TimeToNextWave);
            if(_lose) break; }
```

Продовження лістингу 3.2

```
        while (!_lose && _enemies.Count!=0)
        {
            yield return new WaitForEndOfFrame();
        }
        if (!_lose)
        {
            _ui.SetInfo($»<b>PASS</b>«);
            OnWin?.Invoke();
        }
    }

    private void OnDead(EnemyView view)
    {
        view.OnDead -= OnDead;
        view.OnFinish -= OnFinish;
        _enemies.Remove(view);
    }

    private void OnFinish(EnemyView view)
    {
        _lose = true;
        _ui.SetInfo($»<b>LOSE</b>«);
        DestroyAllEnemies();
        _coroutineManager.StartAfterTimeout(5f, ()=> StartRound(_round,
ELocation.TARGET_1));
        OnLose?.Invoke();
    }

    void DestroyAllEnemies()
    {
        //…
    }
}
```

кінець лістингу 3.2

Код використовує принципи і патерни Zenject для керування залежностями в Unity-проекті.

В конструкторі EnemySpawnerController використовуються параметри, що вказують на необхідні залежності. Ці залежності передаються через конструктор, де вони зберігаються в приватних полях класу [5].

Клас EnemySpawnerController використовує EnemySpawnerDatabase, EnemyPrefabDatabase, LocationDatabase, GameUIView, IProjectCoroutineManager, InstantiatorView як залежності для виконання своїх функцій.

Клас EnemySpawnerController використовує події (OnWin, OnLose) для повідомлення про події, такі як перемога або поразка в грі.

Клас EnemySpawnerController має метод StartRound, який починає новий раунд гри. В цьому методі використовуються залежності, які були внедрені через конструктор.

Клас EnemySpawnerController використовує IProjectCoroutineManager для запуску корутин (SpawnWave) та обробки спавну ворогів в різних хвилях.

Клас EnemySpawnerController використовує різні методи для управління грою, такі як OnDead та OnFinish, які викликаються при смерті або завершенні ворогів.

Загалом, цей код демонструє використання принципів і патернів Zenject для керування залежностями та внедрення компонентів в гру. Клас EnemySpawnerController відповідає за управління створення ворогів та обробку подій у грі, використовуючи різні залежності та методи, що дозволяють керувати грою залежно від різних умов та подій.

Також для полегшення створення нових рівнів та створення універсального коду, було створено датабазу для інформації про ворогів, налаштування якої є легким завданням для розробника (рис. 3.1).

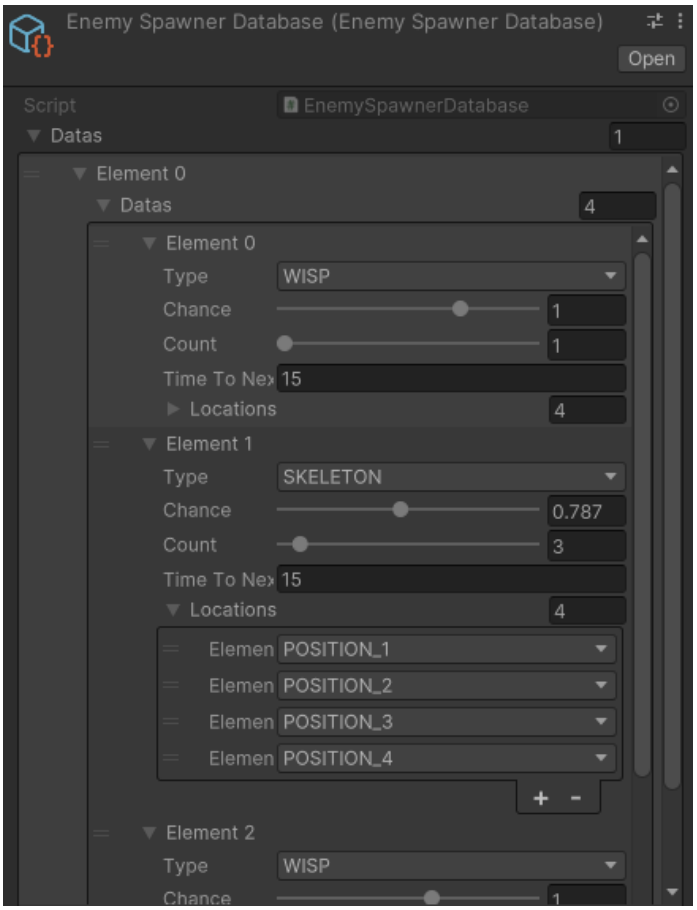

Рисунок 3.1 – Представлення бази «EnemySpawnerDatabase» в інтерфейсі

Клас EnemyView є компонентом Unity і відповідає за візуальне відображення ворогів у грі. Основними особливостями цього класу є наявність подій OnFinish та OnDead, які викликаються при завершенні або смерті ворога відповідно (лістинг 3.3).

Лістинг 3.3 – Код класу «EnemyView»

```
public class EnemyView : MonoBehaviour
{
    protected Vector3 _target;
    public event Action<EnemyView> OnFinish;
    public event Action<EnemyView> OnDead;
    public Vector3 Target
    {
        set{
        _target = value;
        SetTarget();
         }
    }

    protected virtual void SetTarget()
    {
        ;
    }
    // Update is called once per frame
```

Продовження лістингу 3.3

```
 void Update()
    {
        if (Vector3.Distance(transform.position, _target) < 2f)
        {
            Debug.Log(«++»);
            OnFinish?.Invoke(this);
        }
    }

    public void Death()
    {
        Debug.Log(«-»);
        OnDead?.Invoke(this);
        Destroy(gameObject);
    }
}
```

кінець лістингу 3.3

Основні елементи коду класу EnemyView:

- змінна _target зберігає координати цілі, до якої повинен рухатись ворог;
- властивість Target використовується для встановлення значення _target. При встановленні нового значення _target, викликається метод SetTarget(), який може бути перевизначений у похідних класах;
- у методі Update() перевіряється відстань між поточною позицією ворога (transform.position) та ціллю (_target). Якщо відстань менше 2 одиниць, то ворог вважається завершеним (OnFinish) і відповідна подія викликається;
- метод Death() викликається при смерті ворога. Він спочатку викликає подію OnDead, щоб повідомити про смерть ворога. Потім графічний об’єкт ворога (gameObject) знищується за допомогою методу Destroy().

Цей клас визначає основні функціональність для ворогів у грі. Він забезпечує рух ворогів до визначеної цілі, спрацьовує події при завершенні або смерті ворога, а також виконує обробку цих подій у відповідних методах. Клас EnemyView може бути успадкований та розширений для визначення конкретного поведінки ворогів у грі [6].

Також існують класи, які будуть наслідуватись від EnemyView, наприклад, WispView (лістинг 3.4).

Лістинг 3.4 – Код класу «EnemyView»

```
public class WispView : EnemyView
{
   [SerializeField] private BirdAgent _agent;
   protected override void SetTarget()
   {
      _agent.Target = base._target;
   }
}
```

кінець лістингу 3.4

Клас BirdAgent (Додаток А) є підкласом Agent з бібліотеки ML-Agents і використовується для навчання агента, що керує птахом у віртуальній середовищі. Цей клас визначає поведінку птаха, збирає спостереження та приймає рішення за допомогою методів, які викликаються фреймами або при початку епізоду. Для налаштування середовища навчання використовувався спеціальний конфігураційний файл (Додаток Б).

Основні елементи коду класу BirdAgent:

– поля private float _birdSpeed, private float _turnSpeed, private float _maxHeight і private float _rayDistance використовуються для налаштування параметрів поведінки птаха;
– змінні public Vector3 Target, public Transform TargetTransform та public Rigidbody rb використовуються для визначення цілі, трансформації цілі та Rigidbody компонента птаха відповідно;
– метод OnEpisodeBegin() виконує підготовку до нового епізоду навчання. Зберігаються початкові значення, обнуляються швидкість та кутова швидкість Rigidbody, а також обнуляється нагорода (_reward);
– метод CollectObservations(VectorSensor sensor) збирає спостереження про стан птаха і середовище. Значення дистанції до цілі та інші параметри використовуються для створення спостережень, які передаються у векторний сенсор (sensor). Ці спостереження використовуються в навчанні для прийняття рішень агентом;
– метод OnActionReceived(ActionBuffers actions) викликається фреймами і виконує дії, отримані від навчання. Значення дій, що контролюють рух і

обертання птаха, обмежуються та використовуються для зміни швидкості руху та обертання птаха. Також перевіряється зіткнення з підлогою та визначається нагорода на основі відстані до цілі;

– метод Heuristic(in ActionBuffers actionsOut) використовує клавіші на клавіатурі для керування птахом у режимі експерта. Він перетворює натискання клавіш на значення дій, які потім використовуються для руху та обертання птаха.
– метод OnDrawGizmos() використовується для візуалізації деяких об’єктів та ліній на сцені в редакторі Unity. Він малює лінію та кулю, які представляють ціль, а також малює лінію, яка показує напрямок погляду птаха.

Цей клас реалізує основну логіку агента птаха, включаючи збір спостережень, прийняття рішень на основі цих спостережень та виконання відповідних дій у середовищі. Він також надає можливість керувати птахом за допомогою клавіш у режимі експерта та візуалізує деякі об’єкти на сцені для наочності (рис. 3.2).

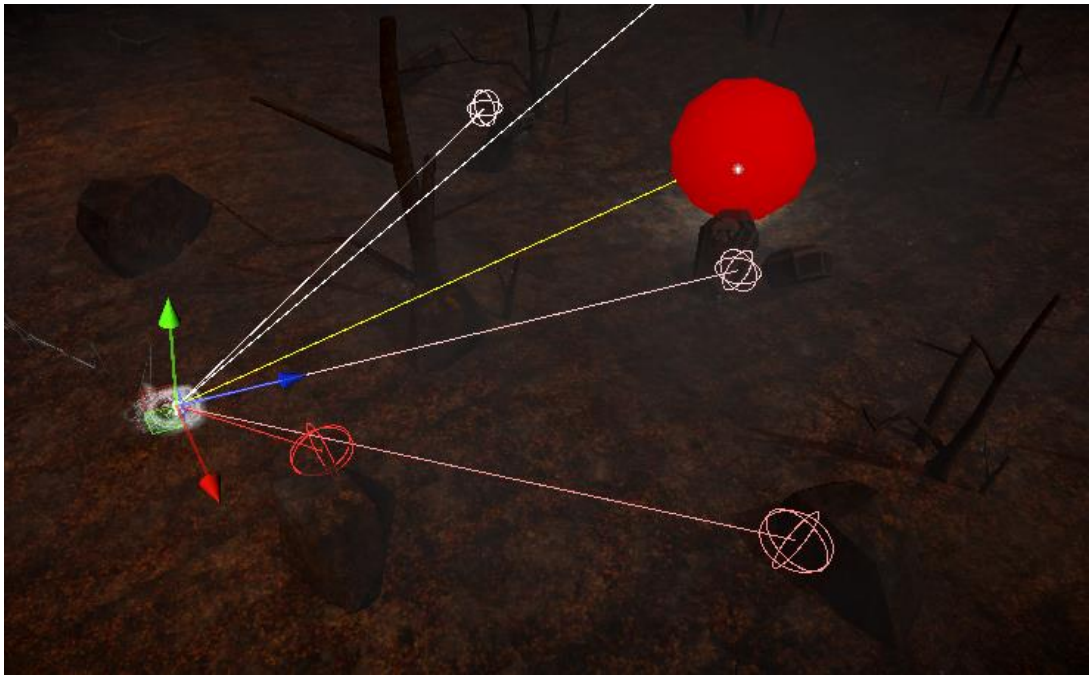

Рисунок 3.2 – Збір спостережень агентом через промені

Також для конфігурації машинного навчання використовувався файл поданий у додатку Б.

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

У цьому розділі буде представлено опис тестування та налагодження інформаційно-комп’ютерної системи «Tower Defence» з використанням штучного інтелекту. Було використано методи, які дозволили перевірити функціональність системи та забезпечити її стабільну роботу.

Вибір методів тестування

Для тестування гри «Tower Defence» з використанням штучного інтелекту було обрано наступні методи:

- модульне тестування. Для перевірки окремих модулів та функцій гри було використано модульне тестування. Кожен модуль був протестований на правильність роботи та відповідність очікуваному результату. Наприклад, були проведені модульні тести для перевірки функцій штучного інтелекту ворожих сил;
- інтеграційне тестування. Для перевірки взаємодії різних компонентів гри було проведене інтеграційне тестування. Було перевірено, як штучний інтелект ворожих сил взаємодіє зі системою розстановки веж та іншими складовими гри;
- функціональне тестування. Для перевірки функціональності гри було проведене функціональне тестування. Було перевірено, чи працюють всі основні функції гри і чи відповідають вони очікуваному результату. Наприклад, було перевірено, чи правильно реагують ворожі сили на розстановку веж гравцем;
- тестування продуктивності. Для перевірки продуктивності гри було проведене тестування продуктивності. Було оцінено, як швидко гра працює та яка її відповідь на різні дії гравця. Наприклад, було проведено тести на швидкість реакції штучного інтелекту ворожих сил на дії гравця.

Мінімальні системні вимоги та рекомендовані параметри

Для роботи інформаційної системи «Tower Defence» з використанням штучного інтелекту рекомендується наступне:

Мінімальні системні вимоги:

- операційна система: Windows 7/8/10 або macOS;
- процесор: Intel Core i3 або еквівалентний;
- оперативна пам’ять: 4 ГБ;
- графічна карта: NVIDIA GeForce GTX 660 або еквівалентний;
- вільне місце на жорсткому диску: 300 МБ.

Рекомендовані параметри:

- операційна система: Windows 10 або macOS;
- процесор: Intel Core i5 або еквівалентний;
- оперативна пам’ять: 8 ГБ;
- графічна карта: NVIDIA GeForce GTX 1060 або еквівалентний;
- вільне місце на жорсткому диску: 2 ГБ.

Результати тестування та налагодження

Під час тестування і налагодження системи «Tower Defence» з використанням штучного інтелекту було виявлено деякі недоліки, які було виправлено для забезпечення кращої функціональності та стабільності гри.

Одним з виявлених недоліків було повільне виконання штучного інтелекту ворожих сил, що призводило до затримок у грі. Для усунення цієї проблеми було проведено оптимізацію алгоритмів створення ворогів та вдосконалення процесу прийняття рішень ворогами. Це дало позитивний результат, і штучний інтелект став працювати швидше та більш ефективно.

Крім того, під час тестування було виявлено проблему з балансом гри, коли окремі рівні були занадто складними або надто простими. Було проведено аналіз рівнів та налаштування параметрів, щоб забезпечити більшу рівновагу та геймплейну варіативність. Після внесених змін рівні отримали баланс для забезпечення нормальної гри користувачам.

Також було виявлено деякі графічні проблеми, пов’язані з відображенням ефектів та анімацією гри. Завдяки виправленню цих проблем, гра стала давати більш детальну та реалістичну візуальну естетику, що покращило загальний враження від гри.

Під час тестування було приділено значну увагу багтрекінгу та виявленню помилок. Було створено спеціальну систему звітності про помилки, яка дозволяє гравцям звітувати про будь-які проблеми, з якими вони зіткнулися під час гри.

Крім того, було проведено тестування гри на різних пристроях. В результаті було здійснено додаткову оптимізацію та адаптацію гри, щоб вона працювала на різних пристроях з різною процесорною потужністю та роздільною здатністю. Для цього використовувались показники з різних частин гри: швидкість рендерингу, виконання скриптів та інші (рис. 3.3).

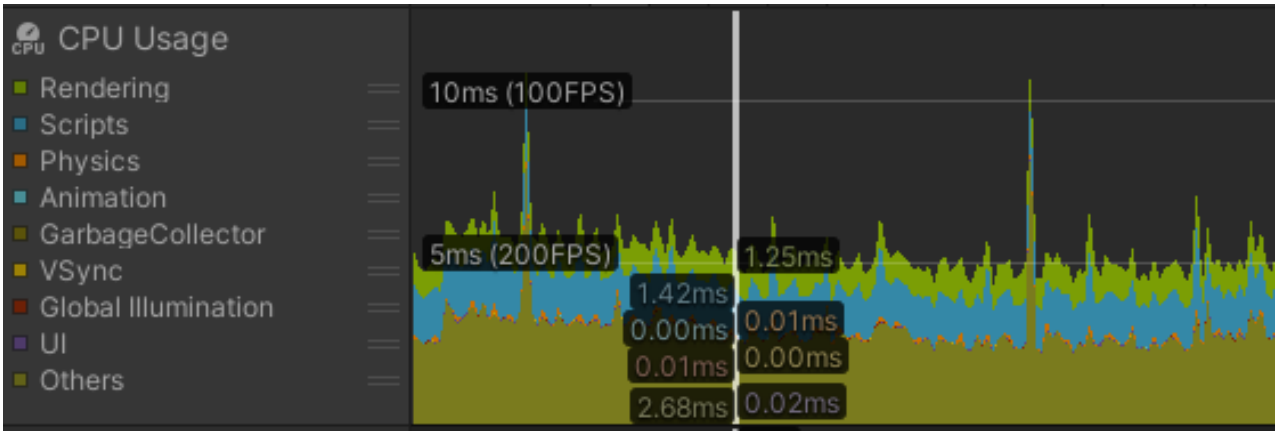

Рисунок 3.3 – Аналітика показників впливу на к-ть кадрів за секунду

Під час тестування було звернено увагу на реагування гри на дії гравців, ефективність алгоритмів штучного інтелекту та загальну стабільність роботи програми. Були виміряні параметри продуктивності, такі як кадрова частота, час завантаження та використання ресурсів. За результатами тестування були внесені необхідні зміни та оптимізації для покращення продуктивності та ефективності гри.

Загалом, завдяки інтенсивному тестуванню та налагодженню гри «Tower Defence» з використанням штучного інтелекту, була досягнута висока якість, стабільність та оптимізація програмного забезпечення. Розробники врахували отримані результати тестування та внесли необхідні зміни, щоб гравці могли насолоджуватися захоплюючим геймплеєм, розумним штучним інтелектом та вражаючою графікою. Гра «Tower Defence» стала однією з найкращих у своєму жанрі завдяки ретельному тестуванню та налагодженню, що дозволило досягти високої якості та задоволення для гравців.

3.3 Розробка інсталяційного пакета

У процесі підготовки інсталяційного пакета гри «Tower Defence» з використанням штучного інтелекту, структура каталогів і файлів важлива для належного розташування компонентів гри на комп’ютері користувача. При цьому, для збереження конфігурації програми та збережень гри використовується механізм PlayerPrefs, що дозволяє зберігати дані налагоджень інформаційно-комп’ютерної системи.

Ось основні складові структури каталогів та файлів, які будуть включені до інсталяційного пакета: головний каталог (основний каталог, в якому розташовані всі файли гри), виконуваний файл гри (основний виконуваний файл гри, який користувачі будуть запускати для початку гри. Наприклад, «TowerDefence.exe»), каталог з ресурсами (всі необхідні ресурси гри, такі як зображення, звуки, моделі персонажів, текстури та інші. Цей каталог може містити підкаталоги для кожного типу ресурсів. Наприклад, «Resources»), конфігураційні файли (всі файли конфігурації гри, які містять налаштування, такі як рівень складності, параметри графіки, налаштування штучного інтелекту та інші, зберігаються у форматі PlayerPrefs).

Деякі конфігурації зберігаються в системному реєстрі під ключами, які відповідають налаштуванням гри. Наприклад:

– ключ «DifficultyLevel» містить значення рівня складності гри;

- ключ «GraphicsSettings» містить налаштування графіки;
- ключ «AIOptions» містить налаштування штучного інтелекту;
- ключ «PlayerProgress» містить інформацію про досягнутий рівень гри користувачем;
- ключ «Achievements» містить дані про досягнення, отримані користувачем.

При встановленні та запуску гри, ці ключі та їх значення також будуть додані до системного реєстру користувача.

В процесі інсталяції будуть створюватися ярлики для швидкого доступу до гри. Це можуть бути ярлики на робочому столі, у меню «Пуск» або в інших розташуваннях, вибраних користувачем під час інсталяції. Ярлики будуть створені з використанням виконуваного файлу гри («TowerDefence.exe») та будуть мати встановлені параметри шляху до виконуваного файлу та іконки гри.

Ці компоненти структури каталогів та файлів, разом зі збереженням конфігурації та даних гри за допомогою PlayerPrefs, дозволять легко встановити гру «Tower Defence» на комп’ютерах з операційною системою Windows та забезпечать належне функціонування програми.

3.4 Економічне обґрунтування розробки

У розділі економічного обґрунтування розробки гри «Tower Defence» на платформі Steam, розглядається планування рентабельності та повернення інвестицій. Наступні розрахунки базуються на наступних даних:

Розглядається розбиття витрат на різні компоненти проекту. Давайте розглянемо деталізацію витрат.

Витрати на розробку коду.

Для розробки коду гри «Tower Defence» виділяється 200 доларів. Ця сума охоплює оплату розробників, які працювали над програмною архітектурою, імплементацією функціональності та оптимізацією коду.

Витрати на навчання моделей штучного інтелекту.

Для розробки та навчання моделей штучного інтелекту в грі «Tower Defence» виділяється 100 доларів. Ці витрати охоплюють навчання моделей із використанням відповідних наборів даних та обчислювальних ресурсів.

Витрати на створення спрайтів та графічних ресурсів.

Для створення спрайтів, текстур та інших графічних ресурсів у грі «Tower Defence» виділяється 150 доларів. Ця сума охоплює оплату художників, які створювали графічні елементи гри.

Витрати на збір гри до купи.

Для збору гри до купи і створення інсталяційного пакету виділяється 50 доларів. Ця сума охоплює оплату розробників, які займались збиранням гри та підготовкою її до розповсюдження.

Отже, загальна вартість розробки складає: витрати на розробку коду + Витрати на навчання моделей + Витрати на створення спрайтів + Витрати на збір гри до купи = 200 доларів + 100 доларів + 150 доларів + 50 доларів = 500 доларів.

Витрати на розробку гри «Tower Defence» становлять 500 доларів. Вартість реклами оцінюється у 1000 доларів. Очікувана кількість проданих копій: Очікується продати 3000 копій гри. Максимальна ціна продажу гри в різних регіонах становить 3 долари.

Тепер проведемо розрахунки.

Обсяг доходу. Обсяг доходу можна визначити, помноживши очікувану кількість проданих копій на ціну продажу. Таким чином, обсяг доходу від продажу гри «Tower Defence» становитиме: 3000 копій * 3 долари/копія = 9000 доларів.

Загальні витрати. Загальні витрати включають витрати на розробку гри та вартість реклами. У нашому випадку, загальні витрати становлять: Витрати = Витрати на розробку + Вартість реклами = 500 доларів + 1000 доларів = 1500 доларів.

Прибуток. Прибуток можна визначити, віднімаючи загальні витрати від обсягу доходу: Прибуток = Обсяг доходу - Загальні витрати = 9000 доларів - 1500 доларів = 7500 доларів.

Рентабельність. Рентабельність можна визначити, відносячи прибуток до загальних витрат та помножуючи на 100%: Рентабельність = (Прибуток / Загальні витрати) * 100% = (7500 доларів / 1500 доларів) * 100% = 500%.

Термін повернення інвестицій. Термін повернення інвестицій можна визначити, поділивши загальні витрати на прибуток: Термін повернення інвестицій = Загальні витрати / Прибуток = 1500 доларів / 7500 доларів = 0.2 (або 20%) [7].

Отже, з урахуванням витрат на розробку в розмірі 500 доларів та вартості реклами у розмірі 1000 доларів, очікується досягнення прибутку у розмірі 7500 доларів. Це забезпечує рентабельність у розмірі 500%. Термін повернення інвестицій складатиме близько 20%. Такі розрахунки свідчать про ефективність інвестицій у проект розробки гри «Tower Defence» на платформі Steam.

Висновки до розділу 3

У цьому розділі було розглянуто різні аспекти розробки гри «Tower Defence». Охоплені такі пункти, як практична реалізація об’єкта проектування, тестування та налагодження інформаційно-комп’ютерної системи, розробка інсталяційного пакета та економічне обґрунтування проекту.

Було описано конкретну реалізацію гри «Tower Defence» з використанням штучного інтелекту. Були розглянуті алгоритми, архітектура та особливості реалізації гри. В результаті було створено ігровий продукт, який готовий для наступних етапів тестування та налагодження.

Було описано методи тестування та налагодження гри «Tower Defence». Були використані різні підходи, включаючи модульне тестування, інтеграційне

тестування та системне тестування. Було виявлено та усунуто недоліки, що дозволило покращити якість гри перед випуском.

Було описана структура каталогів та файлів програмного продукту, а також налаштування для збереження конфігурації програми та створення ярликів. Крім того, розглянуто систему реєстрації та встановлення гри на комп’ютер користувача. Це забезпечить зручний та простий процес встановлення гри для кінцевого користувача.

Було розглянуто важливий аспект розробки гри – економічну ефективність. Було обрано платне розповсюдження гри через платформу Steam, що дозволить досягти широкого кола аудиторії гравців. Ціна гри була визначена на рівні, що не перевищує 5 доларів, з урахуванням регіональних варіацій. Був розрахований рентабельний рівень продажів та прибуток від реалізації гри.

У підсумку, розділ 3 надає повний огляд розробки гри «Tower Defence», починаючи від практичної реалізації об’єкта проектування до економічного обґрунтування розробки. Цей розділ дає розуміння процесу створення гри, його важливих складових та переваги планування та використання ефективних стратегій розробки. Завдяки виконанню всіх етапів розробки, гра «Tower Defence» готова до наступного етапу – випуску на ринок та отримання успіху серед гравців.

ВИСНОВКИ

У даній кваліфікаційній роботі було успішно виконано всі поставлені завдання, пов’язані з розробкою гри в жанрі «Tower Defence» на базі Unity з використанням штучного інтелекту. Кожне завдання відповідало конкретній меті роботи і вимагало впровадження відповідних технологій та рішень.

Було створено базову архітектуру гри, включаючи систему управління рівнями, головне меню, ігрове поле та інші важливі компоненти.

Розроблений інтуїтивний та зручний інтерфейс користувача, який забезпечує зручну навігацію та взаємодію гравця з грою. Досягнуто шляхом впровадження зрозумілих іконок, кнопок, враховуючи найкраще їхнє положення.

Інтегровано Universal Render Pipeline (URP) для створення візуального оформлення гри, який дозволив створити візуально привабливий та оптимізований візуальний стиль гри. Було налаштовано освітлення, матеріали та ефекти, щоб створити реалістичну графіку.

Використано ML-Agents для навчання моделей штучного інтелекту для керування персонажами в грі. Було налаштовано середовище та параметри навчання, щоб досягти оптимальної ефективності навчання моделей. Проведено тренування моделей штучного інтелекту, використовуючи підхід підсилення (reinforcement learning), для досягнення бажаних поведінкових стратегій ворожих сил.

Використано Zenject, фреймворк інверсії керування та впровадження залежностей, для організації коду та керування скриптами гри. Забезпечено модульність та розширюваність системи шляхом правильного розподілу відповідностей між різними компонентами гри та використанням ін’єкції залежностей.

Розроблена механіка заклинань, яка дозволяє гравцю активно взаємодіяти з грою та впливати на хід битви. Реалізовані заклинання з унікальними

властивостями та ефектами, що забезпечують різноманітність геймплею та стратегічні можливості гравця.

Впроваджено систему збереження прогресу гри. Забезпечено можливість зберігання даних про досягнення, рівні прогресу та інші важливі параметри гри для зручного продовження гри з попереднього моменту, а також налаштування, які виставив користувач.

Для забезпечення коректної роботи гри було проведено широкий спектр тестів для перевірки функціональності, стабільності та ефективності гри. Виявлені помилки та проблеми були виправлені, що дало змогу забезпечити стабільну роботу гри та задоволення користувачів.

В результаті виконання всіх завдань було досягнуто мети кваліфікаційної роботи – розроблено гру в жанрі «Tower Defence» з використанням штучного інтелекту, яка пропонує гравцям цікавий геймплей та високий рівень взаємодії. Отримані результати підтверджують успішне виконання проекту і вказують на потенціал подальшого розвитку та впровадження штучного інтелекту в ігрову індустрію.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity Documentation. Офіційна документація Unity. URL: https://docs.unity3d.com/ (дата звернення: 08.06.2023).
2. Unity Learn. Навчальні ресурси Unity. дата звернення: https://learn.unity.com/ (дата звернення: 08.06.2023).
3. VoidNine. plug-inml-agent для unity. Habr. URL: https://habr.com/ru/articles/416297/ (дата звернення: 10.06.2023).
4. Zenject GitHub Repository. Репозиторій Zenject на GitHub. URL: https://github.com/modesttree/Zenject (дата звернення: 08.06.2023).
5. ML-Agents GitHub Repository. Репозиторій ML-Agents на GitHub. URL: https://github.com/Unity-Technologies/ml-agents (дата звернення: 08.06.2023).
6. Unity Asset Store. Офіційний магазин активів Unity. URL: https://assetstore.unity.com/ (дата звернення: 08.06.2023).
7. Jackson, S. Mastering Unity 2D Game Development. Birmingham, UK: Packt Publishing. 2018. 456 p.

ДОДАТКИ

Додаток А

«BirdAgent»

```
public class BirdAgent : Agent
{

    [SerializeField]private float _birdSpeed = 5.0f; // Speed of the bird
    [SerializeField]private float _turnSpeed = 300;
    [SerializeField]private float _maxHeight = 15f;
    [SerializeField]private float _rayDistance = 10f;


    public Vector3 Target;
    public Transform TargetTransform;
    public Rigidbody rb; // Rigidbody component of the bird
    private Vector3 _startingPosition; // Starting position of the bird
    private float _previousDistance;
    private float _startDistance;
    private float _reward;


    public override void OnEpisodeBegin()
    {
        _previousDistance = Vector3.Distance(transform.position, Target);
        _startDistance = _previousDistance;
        rb.velocity = Vector3.zero;
        rb.angularVelocity = Vector3.zero;
        _reward = 0;
    }

    public override void CollectObservations(VectorSensor sensor)
    {
        _previousDistance = Vector3.Distance(transform.position, Target);
        _startDistance = _previousDistance;
        var targetDirection = Target - transform.position;
        sensor.AddObservation(Vector3.Dot(transform.forward,targetDirection));
//sensor
        sensor.AddObservation(Vector3.Dot(transform.right,targetDirection));
//sensor
        sensor.AddObservation(Vector3.Dot(transform.up,targetDirection));
//sensor
        float dist = Vector3.Distance(transform.position, Target);
        float normalizedDist = Mathf.Clamp((_startDistance - dist) /
_startDistance, -1f, 1f);
        sensor.AddObservation(normalizedDist); //1
        sensor.AddObservation(Vector3.Dot(transform.forward,Vector3.up));
//sensor
        RaycastHit hit;
        if (Physics.Raycast(transform.position, Vector3.down, out hit,
_maxHeight))
        {
            sensor.AddObservation((_maxHeight -
Vector3.Distance(hit.point,transform.position))/_maxHeight); //sensor
        }
        else
        {
            sensor.AddObservation(0f); //sensor
        }
    }

    public override void OnActionReceived(ActionBuffers actions)
    {
        float moveHorizontal = actions.ContinuousActions[0];
```

```
        var rotateY = Mathf.Clamp(actions.ContinuousActions[1], -1f, 1f);
        var rotateX = Mathf.Clamp(actions.ContinuousActions[2], -1f, 1f);
        var rotateZ = Mathf.Clamp(actions.ContinuousActions[3], -1f, 1f);
        var rotateDir = Vector3.zero;
        rotateDir = -transform.up * rotateY + transform.right * rotateX;
        rb.AddForce(transform.forward * (Mathf.Abs(moveHorizontal) * _birdSpeed
* Time.fixedDeltaTime), ForceMode.VelocityChange);
        transform.Rotate(Vector3.up, Time.fixedDeltaTime * _turnSpeed *
rotateX);
        transform.Rotate(Vector3.right, Time.fixedDeltaTime * _turnSpeed *
rotateY);
        transform.Rotate(Vector3.forward, Time.fixedDeltaTime * _turnSpeed *
rotateZ);

        RaycastHit hit;
        if (!Physics.Raycast(transform.position, Vector3.down, out hit,
_maxHeight))
        {
            //Debug.Log(«-» + reward);
            //SetReward(0f);
            //EndEpisode();
        }
        else
        {
            var distance = Vector3.Distance(transform.position, Target);
            if (distance < _previousDistance)
            {
                //AddReward(2*(_previousDistance - distance) / _startDistance);
                //reward += (_previousDistance - distance) / _startDistance;
                _previousDistance = distance;
                if (distance < 1f)
                {
                    //Debug.Log(«+» + reward);
                    //SetReward(5f);
                    //EndEpisode();
                }
            }
        }
    }
    public override void Heuristic(in ActionBuffers actionsOut)
    {
        var continuousActionsOut = actionsOut.ContinuousActions;
        continuousActionsOut[0] = Input.GetKey(KeyCode.UpArrow)? 1f :
(Input.GetKey(KeyCode.DownArrow)? 0.5f : 0);
        continuousActionsOut[1] = Input.GetKey(KeyCode.W) ? 1.0f :
(Input.GetKey(KeyCode.S) ? -1.0f : 0f) ;
        continuousActionsOut[2] = Input.GetKey(KeyCode.D) ? 1.0f :
(Input.GetKey(KeyCode.A) ? -1.0f : 0f) ;
        continuousActionsOut[3] = Input.GetKey(KeyCode.Q) ? 1.0f :
(Input.GetKey(KeyCode.E) ? -1.0f : 0f) ;
    }
    void OnDrawGizmos()
    {
        // Draw a yellow sphere at the transform's position
        Gizmos.color = new Color(255, 194, 0, 30);
        Gizmos.DrawLine(transform.position, Target);
        Gizmos.color = Color.blue;
        Gizmos.DrawLine(transform.position,
transform.position+transform.forward*_rayDistance);
        Gizmos.color = new Color(255, 0, 0, 50);
        Gizmos.DrawSphere(Target, 2);
    }
}
```

Додаток Б

«Config для BirdAgent»

```
behaviors:
  Basic:
    trainer_type: sac
    hyperparameters:
      learning_rate: 0.0003
      learning_rate_schedule: constant
      batch_size: 128
      buffer_size: 2000000
      buffer_init_steps: 1000
      tau: 0.01
      steps_per_update: 10.0
      save_replay_buffer: false
      init_entcoef: 0.01
      reward_signal_steps_per_update: 10.0
    network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.995
        strength: 2.0
      gail:
        gamma: 0.99
        strength: 0.1
        learning_rate: 0.0003
        use_actions: true
        use_vail: false
        demo_path: Assets\Demonstrations\Bird_0.demo
    keep_checkpoints: 5
    max_steps: 50000000
    time_horizon: 128
    summary_freq: 30000
```

Додаток В

«Менеджер корутинів»

```
public class AbstractCoroutineManager:ICoroutineManager
{
    protected CoroutineManagerView _view;

    protected AbstractCoroutineManager()
    {
        var gameObject = new GameObject(«CoroutineManager»);
        _view = gameObject.AddComponent<CoroutineManagerView>();
    }
    public Coroutine StartCoroutine(IEnumerator routine) {
        return _view.StartCoroutine(routine);
    }
    public Coroutine StartCoroutine(string routine) {
        return _view.StartCoroutine(routine);
    }
    public void StopAllCoroutines() {
        _view?.StopAllCoroutines();
    }
    public void StopCoroutine(IEnumerator routine) {
        Debug.Log(«--- StopCoroutine --->);
        _view?.StopCoroutine(routine);
    }
    public void StopCoroutine(Coroutine routine) {
        _view?.StopCoroutine(routine);
    }
    public void StopCoroutine(string routine) {
        _view?.StopCoroutine(routine);
    }
    public void StartAfterTimeout(float timeout, Action callback)
    {
        _view.StartCoroutine(PauseInternal(timeout, callback));
    }
    public void StartAfterRealTimeout(float timeout, Action callback)
    {
        _view.StartCoroutine(PauseRealInternal(timeout, callback));
    }
    public Coroutine StartAfterTimeoutCoroutine(float timeout, Action callback)
    {
       return _view.StartCoroutine(PauseInternal(timeout, callback));
    }
    private IEnumerator PauseInternal(float timeout, Action callback)
    {
        yield return new WaitForSeconds(timeout);
        callback?.Invoke();
    }
    private IEnumerator PauseRealInternal(float timeout, Action callback)
    {
        yield return new WaitForSecondsRealtime(timeout);
        callback?.Invoke();
    }
    public void StartOnNextFrame(Action callback)
    {
        _view.StartCoroutine(PauseInternal(0.0f, callback));
    }
}
```

Додаток Г

«GameUIView для керування інтерфейсом»

```
public class GameUIView : MonoBehaviour
{
    [SerializeField] private TMP_Text _infoText;
    [SerializeField] private TMP_Text _waveText;
    [SerializeField] private GameObject _spellDraw;
    [SerializeField] private SpellDraw _spellDrawScrips;
    [SerializeField] private Image _spell;
    [SerializeField] private Image _spellInDraw;
    [SerializeField] private Sprite _spellSprite;
    [SerializeField] private GameObject _helpText;
    private Vector2 _infoTextPos;
    public event Action OnCastDraw;
    public bool EnabledSpellDraw
    {
        get => _spellDraw.activeSelf;
        set
        {
            _spellDraw.SetActive(value);
            _helpText.SetActive(false);
        }
    }
    public void Start()
    {
        _infoText.DOFade(0, 0f);
        _infoTextPos = _infoText.rectTransform.anchoredPosition;
    }

    public void SetInfo(string text)
    {
        _infoText.DOKill();
        _infoText.text = text;

        _infoText.DOFade(1, 0.2f);
        _infoText.DOFade(0, 4f);
    }

    public void SetWave(int i)
    {
        _waveText.text = i.ToString();
    }

    public void CastDraw()
    {
        _spellInDraw.sprite = _spellSprite;
        _spell.gameObject.SetActive(true);
        _spell.sprite = _spellSprite;
        _spellDrawScrips.StartDraw();
        OnCastDraw?.Invoke();
    }
}
```