НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Комп’ютерна гра у жанрі шутер з використанням машинного навчання

Виконав : студент __4__ курсу, групи ІП-93
(шифр групи)

_____ Грибенко Єгор Володимирович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ст. викладач, доктор філософії Таран В.І._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н. Волокита А. М._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2023 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Грибенко Єгора Володимировича

1. Тема проєкту *Комп’ютерна гра у жанрі шутер з використанням машинного навчання*
керівник проєкту Таран Владислав Ігорович, ст. викладач, доктор філософії,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету № 2102-с від «31» травня 2023р.

2. Термін здачі студентом закінченого проєкту *5 травня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд та алаліз існуючих відеоігор жанру шутер та ігор з використанням технологій ШІ.

Розділ 2. Аналіз засобів розробки.

Розділ 3. Реалізація програмного продукту.

Розділ 4. Огляд та тестування розробленого продукту

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, Діаграма класів (функціональна схема), алгоритм дій програмного забезпечення (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «26» грудня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*5.12.2022-10.12.2022*	
2.	*Вивчення та аналіз завдання*	*10.12.2022-10.03.2023*	
3.	*Розробка архітектури та загальної структури системи*	*10.03.2023-20.03.2023*	
4.	*Розробка структур окремих підсистем*	*20.03.2023-01.04.2023*	
5.	*Програмна реалізація системи*	*01.04.2023-10.04.2023*	
6.	*Оформлення пояснювальної записки*	*10.04.2023-15.05.2023*	
7.	*Захист програмного продукту*	*20.04.2023*	
8.	*Передзахист*	*08.06.2023*	
9.	*Захист*	*20.06.2023*	

Студент-дипломник _____ Єгор ГРИБЕНКО
(підпис)

Керівник проєкту _____ Владислав ТАРАН
(підпис)

АНОТАЦІЯ

Робота присвячена розробці відеогри у жанрі тактичний шутер, із застосуванням технологій машинного навчання з метою поліпшення поведінки NPC у грі.

Для реалізації гри було використано рушій Unity, а також мову програмування C#.

З метою поліпшення поведінки ШІ-гравців у процесі розробки застосовано методи машинного навчання, а саме, засоби Unity Machine Learning Agents. Інтелектуальні агенти використовуватимуть технологію навчання з підкріпленням, поведінкове клонування та імітаційне навчання на основі дій гравця, для симуляції реалістичної поведінки.

Для створення візуального дизайну гри та ігрового контенту використано навички малювання та моделювання за кресленнями, і, відповідно, засоби Adobe Photoshop та Blender. Для пришвидшення часу розробки також будуть використані окремі матеріали з Unity Asset store, що знаходяться у вільному доступі.

Ключові слова: Комп’ютерна гра, Unity, C#, шутер, FPS, ML Agents, навчання з підкріпленням.

ANNOTATION

This paper is devoted to the development of a tactical shooter video game using machine learning technologies to improve the behavior of NPCs in the game.

The Unity engine and C# programming language were used to implement the game.

To improve the behavior of AI players, machine learning methods were applied in the development process, namely Unity Machine Learning Agents.

Intelligent agents will use reinforcement learning technology, behavioral cloning, and simulation learning based on player actions to imitate realistic behavior.

Drawing and modeling skills, as well as Adobe Photoshop and Blender tools, will be used to create the visual design of the game and game content. Some freely available materials from the Unity Asset store will also be used to speed up development time.

Key words: Computer game, Unity, C#, shooter, FPS, ML Agents, reinforcement learning.

Справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Комп'ютерна гра у жанрі	3		
			шутер з використанням			
			машинного навчання			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Комп'ютерна гра у жанрі	84		
			шутер з використанням			
			машинного навчання			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Комп'ютерна гра у жанрі	1		
			шутер з використанням			
			машинного навчання			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Комп'ютерна гра у жанрі	1		
			шутер з використанням			
			машинного навчання			
			Діаграма класів			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Комп'ютерна гра у жанрі	1		
			шутер з використанням			
			машинного навчання			
			Алгоритм дій програмного забезпечення			
			(принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Комп'ютерна гра у жанрі	43		
			шутер з використанням			
			машинного навчання			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА		
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***			
Розроб		Грибенко Є.В.			*Комп'ютерна гра у жанрі шутер з використанням машинного навчання* Опис альбому	Літ.	Аркуш 1 / Аркушів 1
Перев		Таран В.І.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93	

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Комп'ютерна гра у жанрі шутер з використанням машинного навчання*»

Київ – 2023

ЗМІСТ

НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2

ПІДСТАВИ ДЛЯ РОЗРОБКИ 2

МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2

ДЖЕРЕЛА РОЗРОБКИ 2

ТЕХНІЧНІ ВИМОГИ 3

Вимоги до розробленого продукту 3

Вимоги до програмного забезпечення 3

Вимоги до апаратної частини 3

ЕТАПИ РОЗРОБКИ 3

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Грибенко Є.В.			**Комп'ютерна гра у жанрі шутер з використанням машинного навчання** Технічне завдання	Літ.	Аркуш 1 / Аркушів 3
Перевірив		Таран В.І.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93	
Н. Контр.		Волокита А. М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку ігрового застосунку, а також на подальшу підтримку та вдосконалення розробленого продукту.

Областю застосування є аудиторія вітчизняних та зарубіжних гравців, що цікавляться військовою тематикою в іграх, зокрема екшн-іграми та шутерами. Така гра може слугувати для проведення дозвілля користувачами.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка ігрового застосунку та застосування методів машинного навчання для реалізації поведінки персонажів.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- В грі повинні бути реалізовані ігрові механіки руху, стрільби, прицілювання та нанесення шкоди супротивникам.
- Поведінка супротивників повинна бути реалізована за допомогою методів машинного навчання - навчання з підкріпленням та імітативного навчання.
- Для навчання інтелектуальних агентів повинні бути реалізовані додаткові середовища тренування
- Повинен бути створений ігровий контент за допомогою малювання текстур, моделювання та анімації.
- Повинні бути реалізовані головне меню, меню налаштувань звуку та керування, а також можливість користувача призупинити гру.

5.2. Вимоги до програмного забезпечення

- ОС Windows або Mac

5.3. Вимоги до апаратної частини

- ЦП не менше за Intel® Core (TM) i5-8300H.
- ROM не менше ніж 16 ГБ.
- RAM не менше ніж 6 ГБ.
- GPU Nvidia GeForce GTX 1050 або кращий.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	05.12.2022-10.12.2022
Вивчення та аналіз завдання	10.12.2022-10.03.2023
Розробка архітектури та загальної структури системи	10.03.2023-20.03.2023
Розробка структур окремих частин системи	20.03.2023-01.04.2023
Програмна реалізація системи	01.04.2023-10.04.2023
Виправлення помилок	10.04.2023-14.05.2023
Оформлення пояснювальної записки	15.05.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Комп’ютерна гра у жанрі шутер з використанням машинного навчання»

Київ – 2023

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 4

ВСТУП 5

РОЗДІЛ 1. ОГЛЯД ТА АЛАЛІЗ ІСНУЮЧИХ ВІДЕОІГОР ЖАНРУ ШУТЕР ТА ІГОР З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ШІ 8

1.1 Визначення понять та огляд предметної області 8

1.2 Огляд відеоігор у жанрі шутер 14

1.2.1 Counter Strike: Global Offensive (CS:GO) 14

1.2.2 Player Unknown Battlegrounds (PUBG) 17

1.2.3 SQUAD 19

1.3 Огляд ігрового ШІ у відеоіграх 20

1.3.1 Finite-state machine у грі Half-Life 20

1.3.2 Дерево поведінки у Halo 2: Anniversary 22

1.3.3 Drivatar у Forza Motorsport 6 25

ВИСНОВОК ДО РОЗДІЛУ 1 28

РОЗДІЛ 2. АНАЛІЗ ЗАСОБІВ РОЗРОБКИ 29

2.1 Ігрові рушї 29

2.1.1 Ігровий рушій Unity 30

2.1.3 Unreal Engine 31

2.1.4 Godot Engine 32

2.1.5 Вибір рушія та його додаткових технологій 33

2.1.6 Вибір технологій машинного навчання 34

2.2 Вибір джерел контенту гри 36

2.3 Вибір засобів створення контенту 39

ВИСНОВОК ДО РОЗДІЛУ 2 43

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ 44

3.1 Створення ігрового контенту 44
3.1.1 Малювання проекцій 3Д-моделей персонажів 45
3.1.2 Моделювання та анімація персонажів 47
3.1.2 Моделювання ігрової техніки 51
3.1.3 Створення ігрового оточення в Unity 53
3.2 Програмна реалізація ігрових механік 55
3.2.1 Керування персонажем 55
3.2 Реалізація ігрового ШІ 61
3.3 Робота над оптимізацією гри 70
ВИСНОВОК ДО РОЗДІЛУ 3 72
РОЗДІЛ 4. ОГЛЯД ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОДУКТУ 73
4.1 Огляд розробленої гри 73
4.2 Тестування інтерфейсу гри 78
ВИСНОВОК ДО РОЗДІЛУ 4 82
ВИСНОВКИ 84
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 86

ПЕРЕЛІК СКОРОЧЕНЬ

ШІ	Штучний інтелект
ШНМ	Штучна нейронна мережа
NPC	Non-player character
MMO	Massively multiplayer online
LOD	Level of detail
ПЗ	Програмне забезпечення
FPS	First person shooter
TPS	Third person shooter
PvP	Player versus player
FSM	Finite-state machine
API	Application programming interface
GPT	Generative pre-trained transformer
LSTM	Long short-term memory
ML	Machine learning
PPO	Proximal policy optimization
SAC	Soft actor-critic
RL	Reinforcement learning
GAIL	Generative adversarial imitation learning

ВСТУП

Мільйони людей по всьому світу грають у відеоігри прямо зараз. Очевидно, що це має неабияке фінансове значення. Зростаюча популярність ігор перетворюється на вражаючі суми грошей, тому ігрова індустрія посідає важливе значення на глобальному ринку.

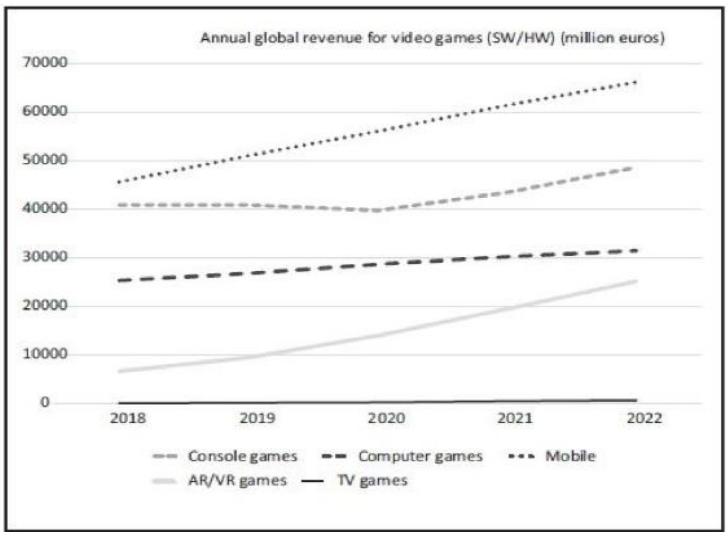


Рисунок - динаміка глобальної виручки за ігрове програмне забезпечення за роками [1]

У світі вже існує велика кількість комп'ютерних та мобільних відеоігор і тепер споживача буває важко здивувати новими іграми, бо так чи інакше вони починають приїдатися, здаватися користувачу нудними, одноманітними, в них вбачаються клішовані, запозичені ігрові механіки, які не обіцяють гравцю нових емоцій. Прибуток проекту прямо залежить від його споживання користувачами, його реклами і популярності, яка змушує багатьох витрачати гроші на покупку продукту. Але найбільш цінним є сповна задовольнити потреби користувача. Якщо гра виявиться хорошою і здобуде прихильність

серед спільноти, то спільнота сама допоможе поширити гру та її культуру для більшої аудиторії - люди будуть радити її друзям та знайомим, говорити про неї в соціальних мережах.

Що ж виокремлює ті самі хороші ігри, які захоплюють увагу користувача на багато десятків годин?

Існують, як мінімум, 4 ключових елементи успішної гри [2]:

- Геймплей
- Історія
- Дизайн та графіка
- Виклик гравцю

Якщо коротко, то:

Вдалий геймплей включає в себе цікаві ігрові механіки та веселий ігровий процес. Різноманітність механік в грі позитивно впливає на реіграбельність (англ. replayability) гри, захоплюючи більше уваги користувача. Більший час проведений в грі, зокрема допомагає користувачу поринути у створений світ гри, а значить дізнатись більше про її культуру та історію, що поведе за собою більше її поширення користувачем своїм друзям, які, в результаті, можливо, зацікавляться і теж її придбають. Багатокористувацька гра, хоч і суперечить історії, але здатна зробити користувача «постійним клієнтом», бо дозволяє гравцю нескінченно отримувати нові емоції, граючи та спілкуючись із новими, незнайомими людьми. Але, для цього, і сам геймплей гри, передусім, повинен бути ненудним.

Дизайн та графіка є ледь не найважливішим елементом, який може створити позитивне враження користувача. Яскраві, численні ефекти, вибухи, спалахи здатні додати екшена в гру, ефекти трясіння, анімації довкілля здатні створити відчуття присутності користувача у світі, а кольорова гамма, стиль декорацій, вигляд персонажів, звуки - утворюють атмосферу гри.

Виклик гравцю - те, що для багатьох гравців є одним з найважливіших елементів гри, без якого в грі, ніби, відсутній сенс, тобто те, заради чого ми граємо. Виклик - це певна проблема, що постає перед гравцем, і за її рішення, він отримує певну винагороду - чи то ігрові предмети, чи відчуття того, що ти чомусь навчився, що щось в собі переборов. Друге часто виявляється навіть більш захоплюючим.

З одного боку, комусь може прийти до вподоби, після довгого «грінда», відкриття рідкісних ігрових предметів або здібностей, які надають йому перевагу в грі, комусь - після пережитої історії та складнощів, перехід на новий рівень приносить задоволення.

З іншого боку, вигляд помираючого боса в грі Dark Souls, після багатьох десятків спроб його вбивства, або повідомлення про проходження складного рівня Geometry Dash після декількох тисяч спроб буває не просто приголомшливо радісним для хардкорного гравця, а й тим, що дійсно запам’ятовується надовго і тим, чим гравець може навіть хизуватись перед іншими.

Мультиплеєр також здатний надати користувачу гідний виклик та безмежний простір для самовдосконалення в процесі змагання з іншими. Недоліком є те, що він може дуже швидко відштовхнути користувача, якщо він не встигне перебороти поріг входження і зіштовхнеться із гравцями, що будуть значно сильніші ніж він. Таким чином постає проблема балансу і матчмейкінгу. Якщо ж гравець стає достатньо сильним та поборює поріг входження, то часті перемоги над іншими створюють йому приємний ігровий процес.

Але, звичайно, для кожного виклику - своя аудиторія. Головне - щоб витрачений час користувача був вартий цієї нагороди.

Отже, моєю задачою, як розробника ігор, постає створення хорошої гри (відповідно до наданого визначення), в міру моєї фантазії, як автора, та можливостей.

РОЗДІЛ 1. ОГЛЯД ТА АЛАЛІЗ ІСНУЮЧИХ ВІДЕОІГОР ЖАНРУ ШУТЕР ТА ІГОР З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ШІ

1.1 Визначення понять та огляд предметної області

Перш за все, слід розібратись, що являють собою поняття “комп’ютерна гра” та “відеогра”.

Поширеною думкою є те, що ці поняття взаємозамінні, але, насправді, існують як мінімум 2 відмінності.

Комп’ютерна гра - це програма, яка спонукає гравця взаємодіяти з віртуальним світом з метою розваги [3]. При цьому, гравець може використовувати різні способи вводу-виводу, наприклад, монітор, комп’ютерну мишу, клавіатуру, геймпад, набір віртуальної реальності, тощо.

Відеогра, в свою чергу, обов’язково передбачає використання візуального засобу виводу, але, при цьому, не обов’язково повинна використовувати комп’ютер (або мікропроцесор) як свою платформу [1].

Таким чином, гра може бути комп’ютерною, але не відеогрою, бо не надає візуальний вивід, і навпаки, може бути відеогрою, і при цьому її ігровий процес не буде забезпечуватися роботою мікропроцесора або комп’ютера.

Такими прикладами можуть бути:

Stop Thief (1979) - комп’ютерна гра, але не відеогра, бо ігровий пристрій має лише звуковий вивід.

Tennis for Two - перша в світі гра, реалізована на осцилографі та аналоговому комп’ютері. Таку гру можна назвати відеогрою, але, комп’ютер не

є класичним у своєму уявленні і не має мікропроцесора і, відповідно, гра не є цифровою, тому є підстави її вважати не комп'ютерною грою.

Але, незважаючи на знайдені різності в даних поняттях, в контексті теми роботи буду використовувати дані поняття як взаємозамінні, бо сучасні ігри, які підлягають дослідженню, знаходяться на перетині даних понять.

Також, з метою скорочення мови, в роботі буде вжито слово «гра» саме під розумінням комп'ютерної гри.

Сучасні ігри мають велику кількість жанрів та піджанрів, причому ігри можуть належати до кількох одразу. Як приклади, можна навести наступні жанри комп'ютерних ігор [3,4,5]:

- Симулятори - це ігри, які відтворюють певну ситуацію з реального або вигаданого світу і надають гравцю контроль над нею. Це може бути симулятор військових дій, пілотування апарата, симулятор водіння машини або спорту, тощо.
- Стратегії - передбачають гравцем планування та управління деякими внутрішньоігровими ресурсами - військовими одиницями, грошима, товарами або робочими ресурсами, тощо.
- Екшн-ігри - ігри, в яких гравець керує певним персонажем - бігає, стрибає, повзає, тощо, з метою знайдення шляху до наступного рівня, двері, порталу або інших цілей, які сприяють поступовому проходженню гри. Також до екшн-ігор можуть відноситися ігри, в яких гравець стріляє по певним супротивникам або об'єктам.
- Рольові ігри - дозволяють гравцю увійти у роль персонажа або групи, і взаємодіяти з ігровим навколишнім світом. В даному жанрі часто застосовується механіка розвитку персонажів по мірі надходження досвіду, перемог у боях, виконання завдань тощо.

Існують також інші жанри та піджанри ігор, але ці чотири цікаві тим, що гра досліджуваного піджанру "шутер", може належати як до одного з цих

жанрів, так і до декількох з них, або усіх одразу, якщо міститиме їх характерні ознаки.

Слід зазначити, що шутер у звичайному своєму уявленні, відноситься саме до жанру “екшн” ігор, бо основною ознакою шутеру є саме стрільба. Проте, границі розпливаються тоді, коли розробник додає до гри елементи, механіки інших жанрів.

Отже, перейдімо до визначення жанру шутер. Це - гра, яка вимагає від гравця виконання двох задач - стріляти по супротивникам і уникати шкоди від супротивника [5].

Крім цього, в даному жанрі від гравця часто потребується тактика, стратегічне планування, робота в команді з іншими гравцями з метою перемогти супротивника(ів). Таким піджанром є “**тактичний шутер**”. Прикладом тактичного шутеру можна навести найбільш популярну гру в світі - Counter Strike: Global Offensive. В ній команді гравців поставлена задача обіграти команду суперника, застосовуючи різні види озброєння та плануючи удари з різних напрямів ігрової карти.

З визначення жанру шутера очевидно, що супротивники гравця також мають задачу - знищити гравця, а разом з тим, можливо, старатися уникати шкоди від гравця. При цьому, для забезпечення приємного досвіду гравця, супротивники не повинні бути занадто сильними, інакше користувач втратить інтерес до ігрового застосунку. Тут постає проблема складності гри, яка полягає в тому, що прості ігри можуть швидко набридати, а від складних - користувачі морально виснажуються, дратуються та перестають користуватися продуктом. Складність гри часто корелює з порогом входження у гру - часом, потрібним гравцю для освоєння механік гри та навчання раціональній поведінці в грі.

Кожній складності та жанру гри відповідає своя аудиторія. Гравців поділяють на наступні основні типи [6, 7]:

- Казуальні гравці надають перевагу іграм, в які просто та весело грати. Наприклад, Candy Crush, Animal Crossing. Серед шутерів, такою грою є Slime rancher 2.
- Core-гравці - мають ширшу область інтересів ніж казуальні гравці та можуть грати в різні види ігор, але не мають стільки ж ентузіазму і не витрачають на гру стільки ж зусиль, як хардкорні гравці. Прикладами відповідних ігор є Witcher 3, Cyberpunk 2077 та Half-Life 2, останні 2 з яких - шутери від першої особи.
- Хардкорні гравці надають перевагу складним іграм з високим порогом входження і отримують задоволення від подолання важких випробувань або оволодіння складними навичками. Приклади ігор: Dark Souls 3 та OSU!. До складних шутерів можна віднести гру Squad, насамперед через те, що звичайний гравець не стане витрачати багато часу на освоєння механік та складностей такої, відносно, реалістичної гри.
- Гравці, які полюбляють змагання, беруть участь у турнірах проти інших геймерів, або просто отримують задоволення від PvP-сутичок з іншими гравцями. Приклади шутерів: Counter Strike: Global Offensive, Player Unknown Battlegrounds. Інші ігри: Dota 2, Chess.com, тощо.
- Соціальні гравці - отримують задоволення від спілкування з іншими у процесі гри, знаходженні нових друзів у онлайн грі. Приклади: Among Us, Minecraft, VR Chat, Roblox. Серед шутерів - Dying Light.
- Гравці які використовують ігри для навчання, саморозвитку. В іграх можуть займатися конструюванням, вивченням математики, програмування, логіки або навіть певній природній мові, тощо. Приклади: NandGame, Stormworks: Build and Rescue. Приклад шутера - Linguist FPS - The Language Learning FPS.

Отже, у жанр шутер може грати будь-який з розглянених типів гравців, в залежності від складностей та особливостей самої гри, а складність гри визначається силою супротивника та порогом входження - складністю ігрових механік.

Ігрова механіка - певний аспект гри та правила, якими послуговується гравець під час гри [8]. Жанр шутер вирізняють механіки стрільби, влучності та віддачі. Наприклад, якщо стріляти довго, не відпускаючи гачок, або рухатись під час стрільби, точність стрільби зменшується, а в положенні сидячи або лежачи - збільшується. Таке правило існує в багатьох шутерах.

Як було згадано вище, іншою причиною складності гри є супротивники, які протистоять гравцю. Що, або хто ж такий супротивник?

Супротивником гравця може бути інший гравець, або ж так званий бот - ШІ-гравець, яким керує програма.

Гра, в яку спільно грають два або більше гравців, називають багатокористувацькою (multiplayer) грою, а гра, в яку грає лише один гравець (та, можливо, певна кількість ШІ-гравців), називається однокористувацькою.

У загальному розумінні ботом (ШІ-гравцем, ігровим ШІ) є неігровий персонаж (NPC), який притримується поведінки, визначеної методами ШІ, або простим алгоритмом у вигляді state-машини, набору правил, дерева поведінки, тощо. Тобто, ігровий ШІ не обов'язково може використовувати нейронні мережі чи інші методи машинного навчання. Від обраного методу реалізації супротивника може залежати сприйняття гравцем його як іншого, справжнього розумного гравця, або розчарування у простоті чи безглуздості його поведінки, чи навпаки, розуміння простої поведінки супротивника дозволить гравцю передбачати його дії, а тому отримати задоволення від того, що він зрозумів їх поведінку та навчився їх перемагати (звісно, в такому разі треба підтримувати різноманітність супротивників, щоб гравець не переставав вчитися у грі, а гра для нього не стала нудною).

Реалістичність та непередбачуваність поведінки супротивників може зацікавити гравця, спонукати до її вивчення, зробити ігровий процес різноманітнішим та затримати гравця на більший час. Саме тому в роботі буде зроблена спроба зробити гру, в персонажів якої буде поведінка, схожа з поведінкою людини-гравця.

Задача імітації ботом людиноподібної поведінки може бути вирішеною за допомогою технологій штучного інтелекту. В такому випадку ігровий бот розглядається як інтелектуальний агент.

Інтелектуальний агент — це сутність (програма), яка може приймати рішення або виконувати дії на основі даних з навколишього середовища, введених даних користувача або власного досвіду [9].

Популярними методами реалізації поведінки інтелектуального агента є дерева рішень, створення системи зі скінченною кількістю станів (finite state machine), а також застосування нейронних мереж. В іграх нейронні мережі зустрічаються рідше. Докладніше розповім на прикладах в наступному підрозділі.

Отже, ми щойно розглянули основні поняття, потрібні для розуміння предметної області. Тепер слід розглянути існуючі рішення - ігри жанру шутер та порівняти їх між собою, знайти їх переваги, недоліки та визначити застосовані методи ігрового ШІ.

1.2 Огляд відеоігор у жанрі шутер

В цьому розділі розглянемо різні ігри-представники жанру шутер, його піджанрів - FPS, TPS, а також ігри, які також поєднують в собі елементи різних жанрів.

1.2.1 Counter Strike: Global Offensive (CS:GO)

CS:GO - це гра-представник жанру тактичний шутер. В її основному режимі існує 2 команди гравців - терористи, задача яких прорвати оборону, закласти бомбу та захищати її, а також спецпризначенці, задача яких - захищати стратегічні цілі від вибуху бомби.

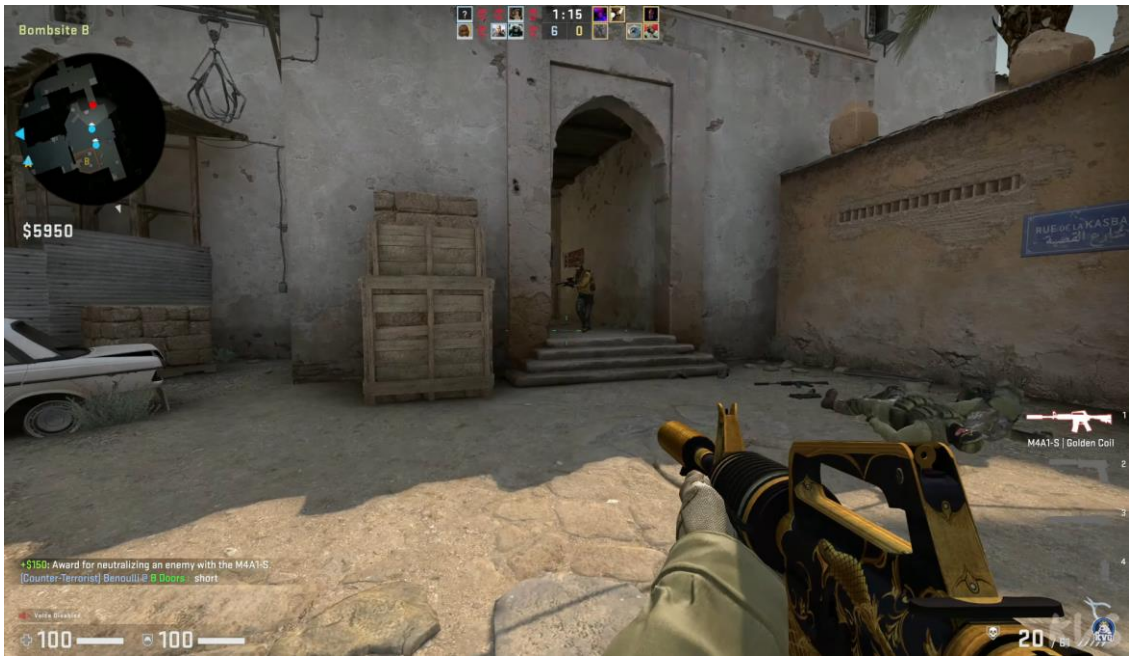


Рисунок 1.1 - Скріншот з гри CS:GO

Ця гра є нащадком серії ігор Counter Strike, історія якої починається ще з 1999 року. Тому в грі присутня стара і проста механіка стрільби, яка стала класичною для ігор жанру.

Механіка стрільби у даній грі має декілька факторів, які впливають на перемогу того чи іншого гравця в момент їх сутички:

- Обрана гравцем зброя визначає точність і швидкість стрільби, завдану шкоду та мобільність гравця.
- Рух гравця в момент стрільби є ще одним чинником, що впливає на його точність. Нерухомий гравець стріляє точніше, ніж той, який рухається. Гравець, має підвищену точність, якщо знаходиться у стані сидіння, а у стані стрибка - знижену точність.
- Кількість попередніх пострілів та час, що пройшов після них визначають віддачу, що негативно впливає на точність гравця.
- Кут і напрямок похибки стрільби (розкид стрільби) визначається випадковим чином та з урахуванням згаданих вище чинників.
- Шкода, завдана супротивнику зменшується, якщо куля перед влучанням також пройшла крізь інші ігрові об’єкти (стіну, укриття, перешкоди, тощо).

При цьому особисті навички гравця визначають його швидкість наведення на супротивника та прийняття рішення про стрільбу. Всі ці фактори безпосередньо впливають на шанс перемоги гравця над супротивником. Також існують додаткові фактори, як-от вигідність позиції гравця, стан осліплення персонажа, увага самого гравця людини, тощо, але вони вже не стосуються механіки стрільби у грі.

Серед переваг даної гри над іншими представниками жанру є наступні особливості:

- Низький поріг входження у гру.
- Інтуїтивність та простота гри - керування персонажем та суть гри зберігаються схожими із класичною серією попередніх популярних ігор та у загальному розумінні простими, а тому гравець не докладає багато зусиль для навчання, і зайшовши у гру, вже розуміє, що йому робити.

- Наявність змагального режиму, в якому, в процесі гри, можна підвищувати своє “звання” та, за допомогою системи матчмейкінгу, отримувати складніших супротивників-гравців.
- В платній версії під час гри гравцям часто “випадають” ігрові предмети, якими можна обмінюватися з іншими або продати за гроші. Таким чином гравець може окупити гроші, витрачені на гру. Звісно, такі грошові надходження надходять за рахунок інших гравців, які купують ці предмети, а компанія-розробник гри бере за це комісію. В будь-якому разі гравці позитивно оцінюють дану можливість.

Але в грі є і наступні недоліки:

- Прості механіки гри, повторюваність геймплею та відсутність певних змін, розвитку у процесі гри іноді спричиняють те, що гравцю набридає дана гра.
- Низька реалістичність стрільби та завданої шкоди. Симуляція стрільби відбувається виключно методом Raycast, тобто проведенням променів.

1.2.2 Player Unknown Battlegrounds (PUBG)

Дана гра поєднує в собі жанри FPS, TPS та жанр королівська битва. Також її можна віднести до жанру MMO, бо в грі одночасно беруть участь до 100 гравців. Керувати персонажем в грі можна як з виду від 3-ї, так і від 1-ї особи.

Рисунок 1.2 - Скріншот з гри PUBG

В цій відеогрі є наступні відмінності (порівняно з попереднім прикладом):

- Персонаж може стріляти в положенні лежачи або сидячи в машині
- В командній грі бере участь велика кількість команд (до 25 при розмірі команди в 4 гравця), при чому в кінці виживає лише одна
- В грі наявна система кастомізації зброї - під час матчу можна знаходити модифікації вогнепальної зброї, які впливатимуть на точність, віддачу озброєння, кількість патронів, тощо

Серед переваг гри:

- Покращена (порівняно з Counter Strike) реалістичність стрільби, яка досягається достатньо цікавим способом - при стрільбі на великі відстані застосовується фізика польоту кулі, а при стрільбі на малі відстані застосовується метод Raycast, який дозволяє зекономити обчислювальні ресурси.
- Наявність у шутері легкових машин, мотоциклів, які дозволяють швидко пересуватися по великій мапі.
- Притаманний “королівським битвам” режим гри викликає особливо сильні емоції та напругу в гравця в кінці матчу, коли він повинен вижити єдиним, одним серед сотні гравців. Тому виграш в матчі стає бажаним для багатьох гравців і забезпечує велику популярність режиму.

Серед недоліків:

- PUBG відомий своєю поганою оптимізацію.
- Вид від 3-ї особи дозволяє заглядати гравцям за перешкоди та рельєф місцевості, що дозволяє їм бачити інших гравців, при цьому сховавшись самому, через що мають велику перевагу. Це спонукає гравців ховатися та грати менш активно, через що активні гравці сильно страждають, дратуються і перестають грати у гру, коли часто помирають від супротивника, який “вискочив нізвідки”.
- Слабка anticheat-система. Якщо в грі трапляється 1 нечесний гравець (чітер), то він псує матч одночасно 99 іншим гравцям.

1.2.3 SQUAD

Цю гру приведено як приклад реалістичного тактичного шутера.

Рисунок 1.3 - Скріншот з гри SQUAD

Дана гра поєднує в собі реалізм та велику кількість екшену, що ставить її між симуляторами та екшн-іграми. Проте, через характерні для симулятора особливості, гра все ще має високий поріг входження, і через це має меншу аудиторію гравців.

Переваги:

- Наявність та різноманіття бойової техніки та авіації, якою можна керувати.
- Реалістичність стрільби та управління транспортом
- Якісна графіка та звукові ефекти
- Механіка взаємодії загону гравців, що стимулює командну роботу

Недоліки:

- Погана оптимізація
- Високий поріг входження

1.3 Огляд ігрового ШІ у відеоіграх

1.3.1 Finite-state machine у грі Half-Life

Half-Life - це гра у жанрі шутер, розроблена у 1998 році і орієнтована на однокористувацьку гру (проте має багатокористувацький міні-режим). Half-Life була надзвичайно високо оцінена критиками (оцінка 96 балів за версією Metacritic) та здобула ряд нагород, зокрема, гра року 1998 [11].

Рисунок 1.4 - Скріншот з гри Half-Life

Ця гра є гарним прикладом застосування FSM як моделі поведінки ШІ-персонажів в іграх.

Finite state machine (FSM) - це модель поведінки, в якій явно задані стани агента, дії, які він виконує, знаходячись у цьому стані та переходи між цими станами. Різні стани пов'язані між собою певними умовами. Якщо виконуються певні умови, модель переходить з одного стану в інший. Усередині стану можна виконувати певні дії або реалізовувати алгоритми поведінки. NPC має бути всередині стану у кожен момент часу [10].

Переваги FSM:

- Простота - складні моделі поведінки можливо створити, представивши у відносно простому вигляді. Переходи між станами, та самі стани є чітко визначеними, що дозволяє легко переглядати структуру.
- Передбачуваність - модель діє однозначно при однакових умовах.

Недоліки:

- Відсутність пам’яті - модель не пам’ятає минулі стани (проте, цю можливість легко додати, модифікувавши алгоритм).
- При частій зміні умов, поведінку буває складно контролювати через переходи між станами.

В Half-Life кожна істота (крім гравця та персонажів, керованих скриптом), керується за допомогою state-машини. Істоти мають набір сприйманої сенсорами інформації, та умови, за якими вони переходять між станами. Кожному стану відповідає своє завдання (Task), яке виконує істота. Набір інформації в Half-Life складається з 32 бітів, що разом утворює одне int32 число. Кожен біт є результатом відповідного сенсора.

Хоча звичайні FSM не мають пам’яті, у Half-Life ця проблема вирішена цікавим способом. Довготривала поведінка забезпечується «режимами» (Schedules) та «цілями» (Goals). Режими «зклеюють» завдання між собою, щоб утворити зв’язну поведінку. Наприклад, у режимі «бігти в укриття» персонаж виконує задану послідовність дій, ігноруючи інші задачі (наприклад, стрільбу

по гравцю), поки мета не буде виконана. Вже згадані вище «цілі» допомагають виконувати ще більш складну поведінку - вони дозволяють послідовно виконувати зв’язку різних режимів, щоб досягти мети [12].

1.3.2 Дерево поведінки у Halo 2: Anniversary

Halo 2: Anniversary - також є грою, що належить до жанру шутер. Ця гра є більш новим ремейком оригінальної гри Halo 2, що стала грою року у 2004.

Рисунок 1.5 - Скріншот з гри Halo 2: Anniversary

На відміну від Half-Life, Halo 2 використовує метод дерева поведінки для керування ШІ-персонажами.

Дерева поведінки (Behaviour Trees) як не дивно, мають форму деревовидної структури.

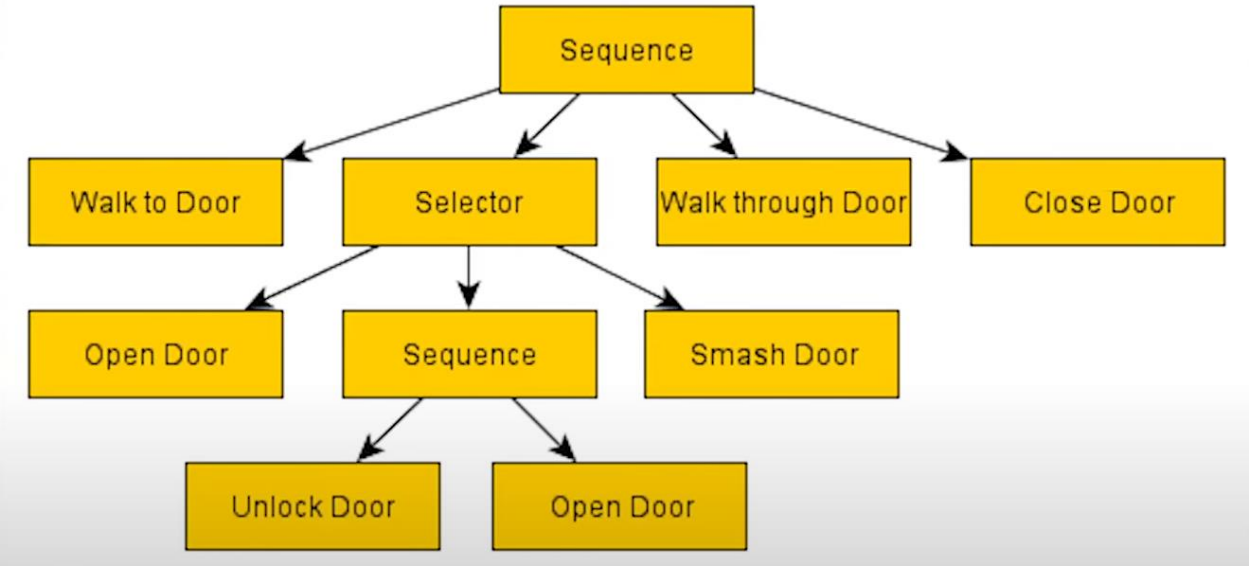


Рисунок 1.6 - Простий приклад дерева поведінки

Таке дерево складається з вузлів. Вузол зазвичай має кілька дочірніх вузлів, які мають свої власні дочірні вузли. Кожен вузол може бути використаний або для задання послідовності дій, однієї конкретної дії, або для застосування логічних операцій, таких як умови та заперечення [13].

Основні вузли дерева наступні [13]:

- Лист (Leaf) - вузол, який не має дочірніх вузлів. Зазвичай він містить певну дію, яку слід виконати.
- Селектор (Selector) - вузол, який використовується для вирішення, які дочірні елементи (або піддерева) слід виконати. Він перевіряє певну умову, і виконує пов’язані вузли в залежності від результату.
- Послідовність (Sequence) - це вузол, дочірні елементи якого повинні бути виконані. Іншими словами, просто піддерево.

Переваги метода дерева поведінки:

- Модульність - складнішу поведінку можна утворювати за допомогою комбінації простішої поведінки. Це також позитивно впливає на масштабованість.
- Краще виконання асинхронної поведінки - за відповідних умов, кілька поведінок можуть виконуватись одночасно, на відміну від FSM.

Недоліки:

- Непередбачуваність - до неї призводять кілька одночасних виконуваних поведінок.
- Метод є складнішим у розумінні, ніж FSM.

У грі Halo 2 наявне наступне дерево поведінки:

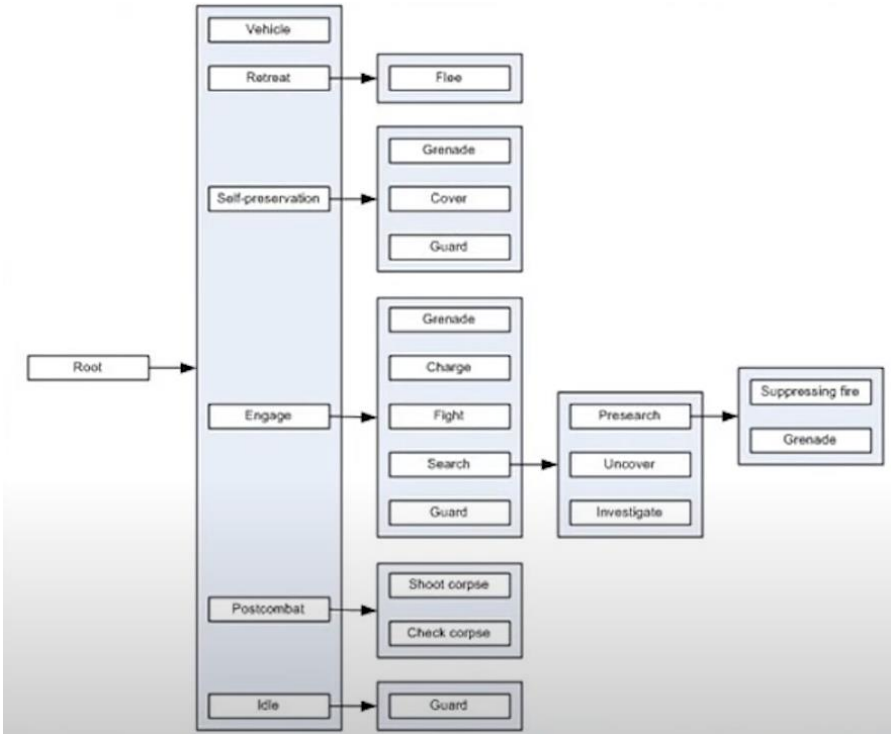


Рисунок 1.7 - Дерево поведінки персонажів у Halo 2

Крім цього, в грі реалізовано ряд розумних систем, які дозволяють покрашити ігровий ШІ.

Серед таких систем - система «Thinking in context». ШІ може приймати специфічні рішення, або вимикати частину дерева поведінки при певному контексті. Контекстом може, наприклад бути, чи персонаж керує машиною, чи він є пасажиром, чи ходить пішки, чи гравець поряд, тощо. Крім цього, використовується система приоритетів, стимулів і затримок, щоб контролювати довготривалу поведінку.

Наприклад, коли гравець сідає у військову машину, супротивники отримують спеціальний стимул, що змінює пріоритети дій і змушує Шукати вільну техніку, сідати в неї і боротися з технікою гравця.

Ще одним цікавим прикладом є зміна приорітетів поведінки, коли ворожий командир знищений. Його підлеглі починають розбігатися від гравця.

Затримка дії стримує персонажа від багаторазового швидкого повторення дій, наприклад, кидання гранати.

1.3.3 Drivatar у Forza Motorsport 6

Однією з ігор, що застосовують машинне навчання є проект Forza Motorsport.

Рисунок 1.8 - Скріншот з гри Forza Motorsport 6

У цій грі використовується система ШІ під назвою «Drivatar». Ця система створена для імітації стилю водіння гравців, щоб створити більш реалістичний і динамічний досвід перегонів.

Ця система збирає дані про те, як гравці їздять у грі. Це стосується того, як вони проходять повороти, наскільки агресивно вони обганяють інших водіїв, як вони реагують на інші автомобілі та як вони працюють під тиском. По суті, будь-яка поведінка, яку демонструє гравець під час водіння, потенційно може бути навчена системою Drivatar.

Після обробки та аналізу даних про реального гравця алгоритмами машинного навчання, система створює ШІ-персонажа, якого можна брати в однокористувацьку гру як бота і змагатися з ним. При цьому його поведінка буде схожою на поведінку конкретної людини, яка грала у гру. Цікавим також є те, що ШІ-водій часто повторює помилки, які допускала людина: може не увійти в поворот і влетіти в перешкоду, врізатися у машину суперника, проїхатися по газону, тощо.

Однією з ключових особливостей Drivatar є те, що вона постійно навчається та оновлюється. Оскільки гравці продовжують грати в гру та їхній стиль водіння розвивається, Drivatar може продовжувати вчитися у них і оновлювати ШІ-водіїв.

Проте, хоча Drivatar створює реалістичніших та непередбачуваних ШІ, вона не ідеальна. ШІ-водії іноді можуть поводитися так, як гравець-людина не буде, і системі важко точно відтворити стиль водіння дуже досвідчених гравців. Тим не менш, це цікавий приклад того, як машинне навчання можна використовувати для покращення ШІ-персонажів у відеоіграх.

Отже, переваги:

- Імітація поведінки справжніх супротивників-людей в однокористувацькому режимі робить його цікавішим, непередбачуванішим, різноманітнішим і створює ілюзію гри зі справжніми людьми, що додає емоцій гравцю.
- Однокористувацька гра дозволяє виконувати балансування гри лише для одного гравця, що спрощує проблему складності гри. Таким чином, розробники додали у гру модифікації та підсилювачі, які можуть зробити гру простіше для гравця, і при цьому не завдати шкоди іншим гравцям-людям. Одночасно з цим, для більш хардкорних гравців, в грі є посилення складності (яке відбирає перевагу у гравця), яке гравець може обрати з метою отримання більшої кількості очків.

Недоліки:

- Ступінь подібності ШІ-водія на людину гравця залежить від кількості зібраних даних про гравця.
- Система все ще не є ідеальною і не здатна точно відтворити поведінку гравця

ВИСНОВОК ДО РОЗДІЛУ 1

В даному розділі було виконано короткий огляд предметної області, а після цього розглянено існуючі рішення ігрових застосунків жанру шутер та таких, що використовують технології ШІ, щоб покращити ігровий досвід користувача.

Серед розглянених програм було виділено використання методів ШІ у грі Forza Motorsport. Вирішено застосувати в проекті методи машинного навчання з подібною метою, тобто для для імітації людиноподібної поведінки суперників, але гра буде відноситися до жанру шутер.

Іншими занайденими перевагами, які було обрано для реалізації в проекті, є наявність в шутері керованої військової техніки (за зразком гри SQUAD). Тому в проекті планується додати броньовані машини, танки та інший транспорт і якісно реалізувати фізику їх руху, що урізноманітнить геймплей та зробить його більш цікавим.

Серед знайдених недоліків у програмах, часто можна побачити погану оптимізацію у грі. Це спонукає використати технології, що покращують швидкодію програми. Через це варто використати технологію Level Of Detail, що дозволить програмі дуже суттєво зменшити кількість полігонів на відображуваній сцені. Інше рішення, яке буде застосовано - моделювання ігрових об'єктів вручну (на відміну від альтернативи - використання готових моделей). Це дозволить контролювати складність геометрії моделей, а отже і навантаження на графічний процесор. Детальніше про інструменти та технології в наступних розділах.

РОЗДІЛ 2. АНАЛІЗ ЗАСОБІВ РОЗРОБКИ

2.1 Ігрові рушї

Основою, на якій базується комп’ютерна гра є ігровий рушій.

Ігровий рушій - це програмний засіб, який забезпечує основні функції для створення та запуску відеоігор. Він виконує рендеринг графіки, обробку вводу користувача, керує фізикою, звуком, а також надає інструменти для пришвидшеної розробки різних аспектів гри, що дозволяє розробникам приділити більше уваги створенню контенту та розробці геймплею [14].

Використання готового рушія значно прискорює розробку ігрового застосунку, порівняно з розробкою такого двигуна вручну. Найбільш відомі приклади ігрових рушіїв - Unreal Engine, Unity та Godot [14].

GameMaker Studio розглядати не будемо, бо він підходить для розробки лише 2Д-ігор, а значить, не підходить для 3Д-шутерів [15].

Інші рушії не увійдуть в розгляд через обмеженість чи відсутність 3Д-графіки, малий розмір спільноти (що впливає на кількість навчальних матеріалів, а отже й на складність відладки), ціну або доступність. Розробку власного рушія розглядати не стану через те, що це є недоцільним щодо витраченого часу. Тому розглянемо 3 згадані рушії.

2.1.1 Ігровий рушій Unity

Unity - це один з найбільш популярних безкоштовних рушіїв для розробки кросплатформенних ігор. Рушій працює на понад 17 платформах. 70% найкращих Android-ігор виконуються саме на Unity [16].

Рисунок 2.1 - Інтерфейс рушія Unity

Unity відомий перш за все великою навчальною базою, широким ком'юніті, великою кількістю готового ігрового контенту (моделей, матеріалів, компонентів, тощо) та простим інтерфейсом (порівняно з, наприклад, UnrealEngine). Недоліками вважається оптимізація у великих проектах та графіка, але насправді обидві проблеми легко вирішуються доповненнями.

Проте, Unity має два серйозних конкуренти - Unreal Engine та Godot.

2.1.3 Unreal Engine

Unreal Engine відомий своїми проривними технологіями в графіці, і особливо корисним він є для великих 3D-проектів з фотореалістичною графікою. Оптимізація також добре себе показує у проектах із великими сценами, великою кількістю об’єктів, тощо. В плані графіки Unreal Engine є найкращим рушієм [17].

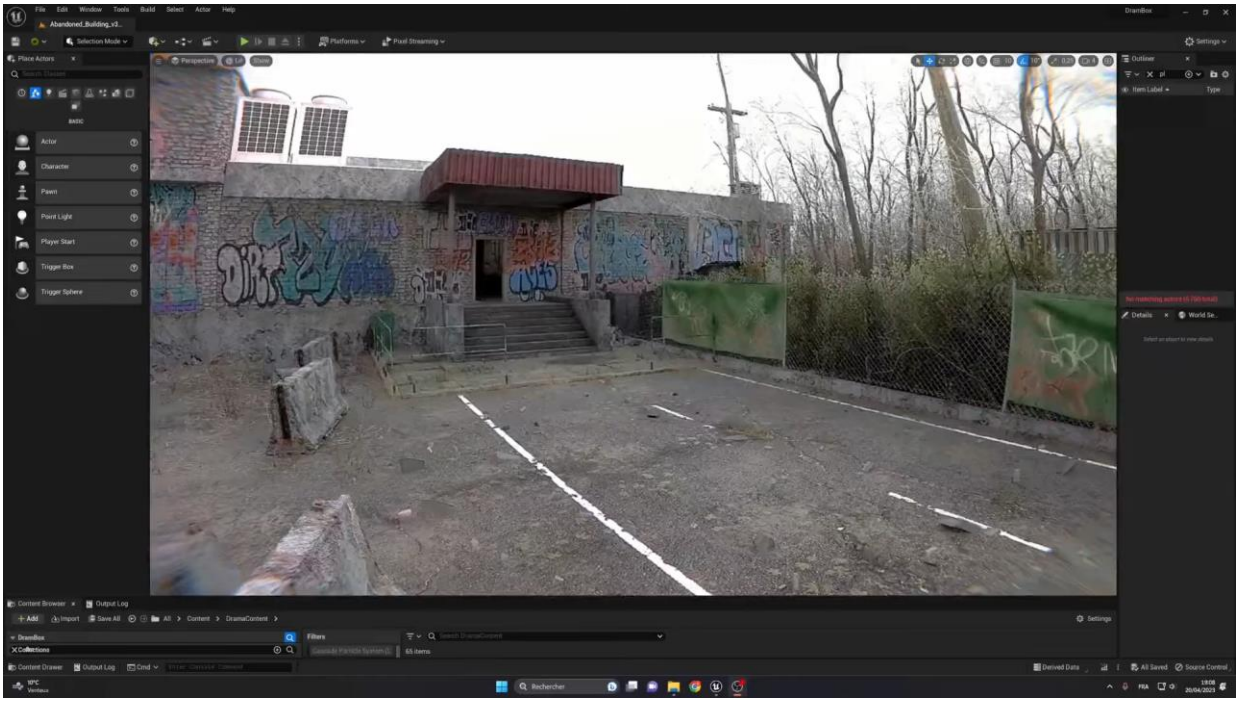


Рисунок 2.2 - Unreal Engine 5 та гра Unrecord [18]

На наведеному рисунку можемо побачити, що фотореалізм гри Unrecord, зробленої на Unreal Engine 5, вражає. Ще більше вражають анімації та геймплей цієї гри, демонстрацію яких можна побачити на офіційному сайті.

Але в Unreal Engine є і недоліки. Одним з них є високий поріг входження через складність двигуна. Також розробника може відштовхнути те, що мовою програмування є C++, яка не є найпростішою мовою. Звичайно, є засіб візуального програмування, але його читабельність та оптимізація у великому

проекті сумнівна, тому двигун потребує поєднання візуального програмування та C++. Ще одним недоліком є порівняно більша ресурсомісткість. Наприклад, при виборі двигуна для розробки простої 2д-гри, точно варто обрати інший двигун.

2.1.4 Godot Engine

Godot є безкоштовним кросплатформним рушієм, який орієнтований на простоту у використанні [19]. Він дозволяє обирати зручну для розробника мову програмування - GDScript, C#, C++, або візуальне програмування. GDScript - спеціально розроблена для двигуна об’єктно-орієнтована мова (створена в цілях оптимізації і інтеграції з двигуном), синтаксис якої схожий на Python [20].

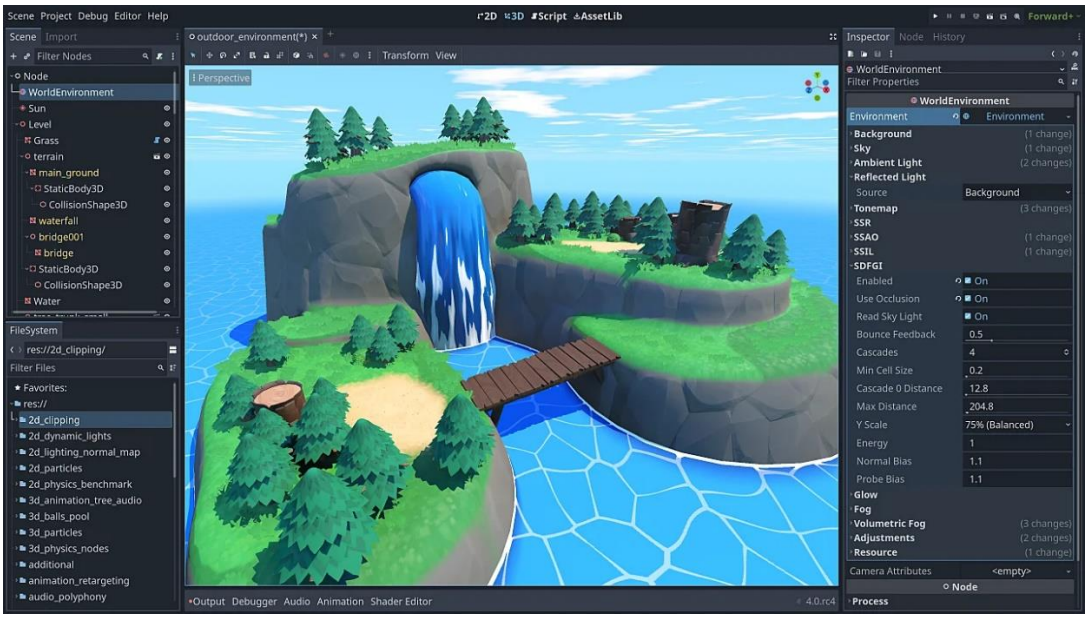


Рисунок 2.3 - Інтерфейс рушія Godot

Недоліком Godot є не найбільше ком’юніті, і, відповідно, не найбільша кількість плагінів та інструментів. І, найбільш важливо, що навчальних матеріалів з а різними тематиками з цього двигуна менше, ніж в Unity та Unreal

Engine. Документація теж іноді може виявитись недостатньо розписаною чи оновленою.

2.1.5 Вибір рушія та його додаткових технологій

Після огляду 3 найбільш популярні з рушіїв, було вирішено обрати рушій Unity. Причини вибору даного рушія наступні.

По-перше, велика кількість навчальних матеріалів та курсів з цього двигуна дозволила мені швидко його опанувати. На це, звісно, впливає і простота самого двигуна, його інтерфейсу користувача.

По-друге, Unity має незліченну кількість доповнень, плагінів, інструментів та готового ігрового контенту, які пришвидшують розробку і відкривають нові можливості, навіть в плані оптимізації та графіки.

Технологія HDRP (High Definition Render Pipeline) із просунутими технологіями освітлення з технологією рейтрейсинга дозволяє досягти рівня графіки, який вже можна порівнювати з Unreal Engine [21]. Але для порівняно невеликих проектів більше підходить unity URP (Universal Render Pipeline), який менше навантажує систему, але не відбирає можливостей для пост-обробки зображення. Саме тому для розробки мого проекту такого розміру, було обрано технологію URP.

Є, також доповнення, що дозволяє досягти шаленої оптимізації з дуже великою кількістю об’єктів (доповнення Unity DOTS. Оптимізація досягається за допомогою автоматичного використання обчислень за допомогою C++ Jobs системи двигуна, а також дуже зручного для процесора розташування пам'яті). Дана технологія може бути застосована в проекті тоді, коли велика кількість об’єктів буде потребувати подібних обчислень, наприклад, політ тисяч куль, ракет, тощо. Проте, цю технологію буде застосовано вже в подальшому розвитку проекту, а не під час даної роботи, бо саме тоді масштаб проекту значно виросте і потребуватиме додаткової оптимізації.

По-третє, Unity дозволяє відносно легко інтегрувати в гру технологію машинного навчання і навчити модель ШІ, наприклад, для супротивників гравця. При цьому можна застосувати імітаційне навчання, клонування поведінки і навчання з підкріпленням, таким чином, модель мозку ШІ-агента може вести себе в грі подібно людині, або перевершити її можливості. Мова йде про технологію Unity ML-Agents, яка працює на базі Pytorch. При цьому, користуючись Unity API з Python, можна застосувати свої методи навчання, або створити модель самому, якщо існує конкретна потреба.

В цьому проекті планується використати методи машинного навчання, тому рушій Unity з технологією ML-Agents підходять якнайкраще.

Отже, ігровий рушій обраний. Тепер слід визначити технології машинного навчання, а також додаткові інструменти, які будуть використані в проекті.

2.1.6 Вибір технологій машинного навчання

Як вже було згадано, відповідною технологією машинного навчання агентів в обраному рушії є Unity ML-Agents. ML-Agents - це open-source проект, що дозволяє використовувати гру як віртуальне середовище для навчання інтелектуальних агентів. Він дозволяє використовувати імплементовані новітні ШІ алгоритми на базі PyTorch, що дозволяють легко тренувати агентів у 2Д та 3Д середовищах [22].

Існує велика кількість методів навчання інтелектуального агента. Вони поділяються на навчання під наглядом (supervised learning), без нагляду (unsupervised learning) та навчання з підкріпленням [25]. В роботі буде використано навчання з підкріпленням та імітаційне навчання, що є навчанням під наглядом (навчання агента відбувається на заздалегідь зібраних даних).

За замовчуванням, ML-Agents дозволяє використовувати алгоритм PPO (Proximal Policy Optimization). PPO - це метод RL (reinforcement learning) загального призначення, перевагою якого є стабільність у порівнянні з багатьма іншими RL алгоритмами [23]. PPO намагається встановити баланс між простотою імплементації, складністю зразків, простотою відладки. Стабільність забезпечується тим, що під час оновлення моделі та мінімізації функції втрат, алгоритм забезпечує відносно мале відхилення політики поведінки від попередньої [24].

Альтернативою до PPO є SAC, який працює ефективніше з меншою кількістю зразків вибірки, тому краще підходить для складних проектів, де крок симуляції займає довше часу. В порівнянні з PPO, часто потребує в 5-10 раз менше зразків [23].

Серед supervised-методів навчання ML Agents дозволяє використати імітаційне навчання та клонування поведінки. Кожен з цих методів може бути самостійним, або використаним в поєднанні з PPO та іншими методами.

Імітаційне навчання за допомогою GAIL передбачає створення додаткової нейронної мережі-критика (дискримінатора), яка, отримавши записи поведінки справжнього гравця та моделі, намагається визначити, чи перед нею справжній гравець чи ШІ модель [24]. Простими словами, вона оцінює ШІ модель поведінки та видає йому нагороду за дії, які схожі на дії гравця. Задачею моделі постає оптимізація нагороди, тому вона когегує свою поведінку і навчається діяти, як справжній гравець.

Клонування поведінки - навчання агента діяти в точній відповідності із записаними демонстраціями. Воно працює гірше з невідомими станами середовища, але дозволяє швидко навчити агента наслідувати поведінку гравця, тому є корисним в поєднанні з іншими методами. Також є найбільш ефективним, якщо в демонстраціях присутні дані про всі (або більшість) можливих станів агента та середовища [24].

Цікавим методом також є видача агенту нагород за його допитливість. В цьому методі створюються дві додаткові нейронні мережі - зворотня та пряма.

Зворотня модель кодує попередню та наступну інформацію з сенсорів агента та намагається передбачити дії, які він виконав, щоб перейти до нового стану.

Друга модель намагається передбачити наступний стан агента у середовищі (інформацію з його сенсорів), маючи попередній стан та дії, які він виконав [24]. Нагородою агента встановлюється похибка передбачення наступного стану (з додаванням наогроди з середовища). Іншими словами, агент отримує нагороду, якщо бачить щось нове. Це є дуже корисним для середовищ, де агент рідко отримує нагороду, а тому гірше вчиться - за допомогою цього методу він прагне виявляти нові стани, а тому краще досліджує середовище і частіше отримує нагороду, що прискорює навчання.

В роботі навчання буде почато з методу PPO через його стабільність, і для початку буде створено просте середовище для навчання. Поступово буде збільшена складність середовища та включатиму інші методи, та, за необхідості, збільшу розмір моделі. Про виконану роботу детальніше в 3 розділі, а зараз, слід визначити джерела контенту, що буде використано для створення комп’ютерної гри.

2.2 Вибір джерел контенту гри

Відомо, що рендер двигуна 3Д-гри здатен відображати як 3Д-моделі, так і 2Д-текстури (наприклад, інтерфейс користувача, billboard-текстури, пласкі текстури трави, гілок дерев, тощо). Такий контент можна або імпортувати у готовому вигляді (придбавши або завантаживши з відкритого доступу), або створивши самому. Те саме стосується іншого контенту - звуків, матеріалів, skybox-матеріалів, анімацій, персонажів, ефектів, чи навіть готових ігрових сцен.

Асет - це термін в розробці ігор, що означає будь-який контент, який може бути використаний для створення ігрового світу та його ігрового процесу.

В кожного розробника постає проблема, які асети звідки дістати, і що робити самому. Зазвичай у 2Д-іграх контент створювати простіше, а у 3Д-іграх складніше, тому інді-розробники часто надають перевагу готовому контенту, якщо вони розробляють 3Д-гру. Основним джерелом готових асетів у випадку з двигуном Unity є Unity Asset Store, де можна придбати асети або завантажити безкоштовно. Але в кожного магазина ігрового контенту є проблема - створені іншою людиною асети зазвичай не будуть відповідати бажанню людини, якщо вона не була замовником. З іншої сторони, створення власних асетів часто потребує від людини суттєвих зусиль та часу, в сукупності з навичками використання відповідних інструментів.

В такому випадку, якщо в малій команді розробки немає дизайнерів та художників, частіше за все доцільнішим варіантом є обрати в магазині асети, які сподобаються більше всього, і з ними розробляти ігровий застосунок. Це зазвичай негативно впливає на унікальність гри (як мінімум, її візуального вигляду), бо такий самий контент можуть використовувати інші розробники.

Також слід згадати, що новим інструментом створення контенту є генерація за допомогою нейронних мереж. Існують різні генеративні моделі, але у відкритому доступі найбільш поширеними поки є Stable Diffusion, ControlNet, різні inpainting та outpainting моделі.

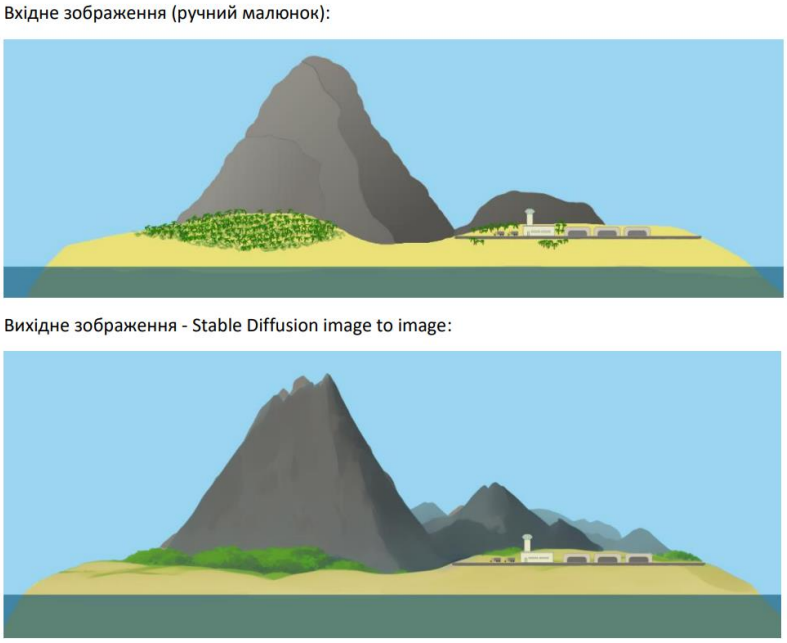


Рисунок 2.4 - Генерація контенту з Stable Diffusion в іншій моїй грі «Planes»

Як можемо побачити на рисунку (створеному мною), ШІ можна використати для генерації картинок або зміни стилю. Художник спочатку малює спрощений вигляд бажаної текстури, а потім нейронна мережа перетворює його (методом image-to-image), щоб стиль відповідав заданому текстовому запиту. З переваг даної технології є те, що вона дозволяє людині створити кращі малюнки за коротший час. З недоліків - те, що на жаль, вона більше підходить для 2Д-проектів, ніж для 3Д, бо 3Д об’єкти зазвичай потребують геометрії, щоб відображатись у тривимірній грі. Дана технологія може бути використана для поліпшення розгорток моделей, проте, це не замінить малювання мною вручну, а додасть ще один етап процесу створення контенту.

Отже, після розгляду різних способів отримання ігрового контенту, було обрано створення абсолютної більшості контенту вручну - моделювання,

анімація та малювання текстур за допомогою відповідних інструментів. На мій вибір вплинули наступні чинники:

- Маю навички моделювання та малювання, тому здатен створювати моделі та текстури, зокрема моделі та розгортку персонажа.
- В попередніх розділах було визначено, що в цілях оптимізації буде створено власні низькополігональні моделі.
- В цьому розділі було визначено, що використання готового ігрового контенту погано впливає на унікальність гри. Але моєю метою є максимально виокремити мій проект серед інших ігор.

Проте, малу частину контенту гри всеодно доведеться імпортувати з відкритого доступу. Це ті асети, які створити сам я не в змозі, бо ще не здобув відповідних начичок. Таким контентом є, насамперед, звуки та музика. Докладніше у розділі з описом процесу розробки.

Отже, моделювання, анімацію та художній дизайн гри було вирішено робити власноруч. Звісно, геймдизайн, програмування та решту роботи так само.

Тепер слід обрати інструменти для створення власного ігрового контенту.

2.3 Вибір засобів створення контенту

Інструмент 3Д-моделювання - це засіб, який дозволяє створювати графічні моделі, що складаються з полігонів. Часто інструмент дозволяє створювати для моделей розгортку та анімувати модель.

Розгортка - це текстура, що накладається на об’ємний об’єкт графічним рушієм, а анімація - дозволяє об’єкту трансформуватися з ходом часу.

Сучасні застосунки для 3Д-графіки дозволяють, як мінімум, моделювати, фарбувати розгортки та анімувати, що і є потрібним для створення контенту в

нашому випадку. Серед найбільш популярних інструментів такого типу: 3DS Max, Maya, Blender.

Blender вважається найкращим безкоштовним ПЗ для 3Д-моделювання. Він надає весь функціонал, необхідний для моделювання, анімації, створення скелетів, симуляції, рендерингу, композиції та motion-дизайну [26].

Переваги:

- Повністю безкоштовний
- Функціонал охоплює весь пайплайн роботи
- Велика спільнота

Недоліки:

- Обмежений функціонал для генерування та візуалізації кривих і поверхонь.

Autodesk 3DS Max - найкраще підходить для 3Д-моделювання реалістичного середовища або персонажів. Популярний серед розробників ігор та дизайнерів інтер'єру [26].

Переваги:

- Добре підходить для розробки ігор
- Містить дуже корисні плагіни

Недоліки:

- Дорогий (235$ за місяць підписки) [27]
- Буває ненадійним, якщо використовувати багато плагінів [26]

Maya - інструмент, що широко використовується для створення та анімації персонажів. Надає потужний функціонал для симуляції рослин, води, вогню, вибухів, а також симуляції волосся та шерсті, сніжного шторму, тощо. Також є порівняно дуже простим у використанні [26, 28].

Переваги:

- Простота використання
- Широкий функціонал

- Процедурні ефекти

  Недоліки:

- Потребує потужної локальної машини
- Дорогий (235$ за місяць підписки, так само як 3DS Max) [28]

Також варто згадати, що ще одним відомим інструментом є Zbrush, перевагами якого є широкі можливості для скульптинга та малювання. Ще однією перевагою є якісна підтримка роботи з графічним планшетом, що дозволяє досвідченим користувачам працювати ефективніше. Проте, поріг входження в даного інструменту великий, і крім того, він є платним (32.69 $ / міс.).

В результаті було обрано Blender, насамперед по тій причині, що він є єдиним безкоштовним open-source додатком серед розглянутих. До того ж, він є зручним та містить всі потрібні функції для моєї задачі, а також має велику спільноту та безліч матеріалів для навчання.

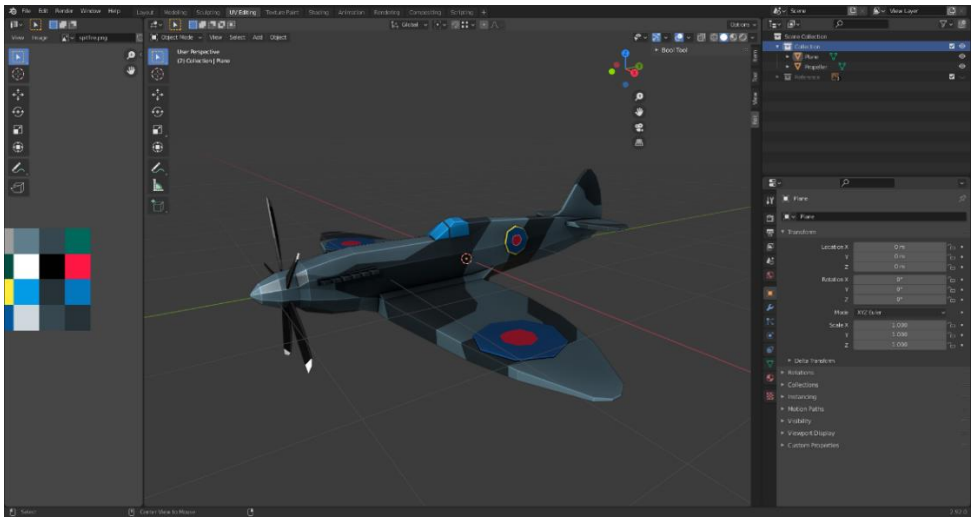


Рисунок 2.5 - Приклад створеної мною низькополігональної моделі в Blender

Як інструмент для малювання растрової графіки для гри, я обираю Adobe Photoshop. Його перевагою є велика кількість корисних графічних інструментів

та можливість використання графічного пера, з урахуванням сили натиску та нахилу (девайс наявний). Потужним розширенням є плагін Stable Art, що дозволяє оброблювати або генерувати зображення за допомогою моделі нейронної мережі. Даний плагін я потенційно зможу використати для покращення намальованих мною зображень, щоб зробити графічний стиль гри якіснішим.

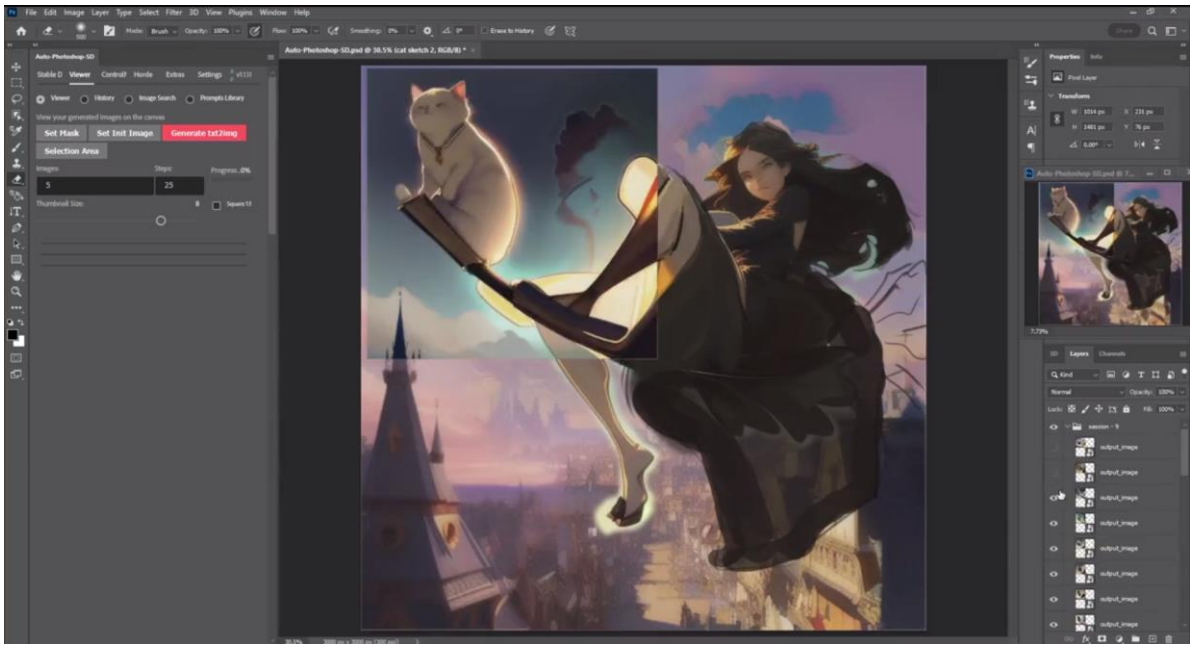


Рисунок 2.6 - Приклад роботи з Stable Diffusion плагіном в Adobe Photoshop

Недоліком даного інструменту є вищий поріг входження, ніж в альтернативних інструментів. Альтернативами є GIMP, SAI, Krita та інші. Але Photoshop є потужнішим за кількістю різних графічних інструментів. До того ж, за кількістю навчального матеріалу у вільному доступі Photoshop лідирує, бо є дуже популярним.

ВИСНОВОК ДО РОЗДІЛУ 2

В даному розділі було розглянено різні інструменти та технології, які буде використано під час розробки гри. Графічним рушієм було обрано Unity через його простоту, велику кількість навчальних матеріалів та наявну технологію ML-Agents, що допоможе інтегрувати ШІ-агентів у гру. Як мову програмування, буде використано C#, який інтегрований з рушієм Unity.

Як інструмент для 3Д-моделювання було обрано Blender, через те, що він є безкоштовним і при цьому дозволяє виконати повний пайплайн створення контенту. За допомогою нього будуть створені моделі, скелети, розгортки, анімації.

Як інструмент для дизайну текстур та матеріалів, було обрано Adobe Photoshop через хорошу інтеграцію графічного планшета, велику кількість фільтрів та можливість встановлення ШІ-плагінів.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

3.1 Створення ігрового контенту

Як я вже зазначив у попередньому розділі, у розробці ігрового застосунку, окрім реалізації програмного коду, дуже важливим є ігровий контент (асети), який надає користувачу змогу сприймати віртуальний світ гри візуально та за допомогою звуків.

У тривимірній грі жанру шутер, користувач зазвичай керує ігровим персонажем, який представлений 3Д-моделлю. Логічним є почати створювати гру саме з персонажа, а також з реалізації механік керування ним у коді програми.

Процес створення 3Д-моделі зазвичай починають з малювання бажаного вигляду цієї моделі в одній або декількох проекціях. Якщо коротко, то художник спочатку збирає набір графічних референсів, і, в поєднанні з власним досвідом та фантазією, малює власного персонажа. Для моделі персонажа зазвичай роблять передню та бокову проекцію, проте, замість останньої можна використати існуючі референси анатомії персонажа. Такі референси також можуть бути застосовані при створенні будь-яких частин тіла, їх проекцій, або ж анімацій 3Д-моделей. Сам процес малювання докладно описувати не стану, бо спеціалізація дипломної роботи все ж більше відноситься до написання програмного коду, про який буде далі. Проте, це не зменшує цінність багатьох витрачених зусиль і часу на створення ігрового контенту, бо це є невід'ємною складовою створення ігрового програмного застосунку.

3.1.1 Малювання проекцій 3Д-моделей персонажів

За допомогою графічного планшета та визначеного у попередньому розділі графічного інструмента Adobe Photoshop, я намалював зовнішній вигляд першого персонажа гри:

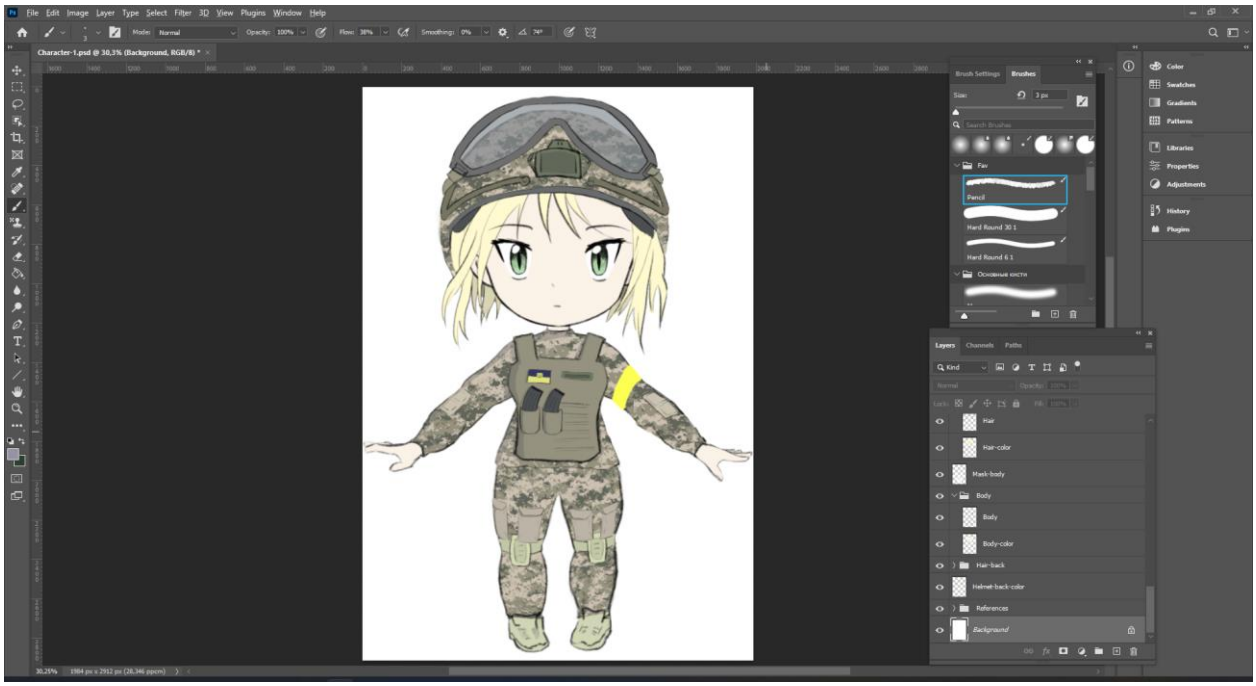


Рисунок 3.1 - Малювання прототипу головного персонажа в Adobe Photoshop

Я використовую велику кількість слоїв для окремих елементів одягу, частин тіла та структурую їх, щоб мати змогу сформувати окремі знімки шарів проекції, які я накладатиму на різні ділянки розгортки 3Д-моделі. Іншими словами, приховування шарів одягу, наприклад, дозволяє сформувати знімок проекції, в якому капелюх (шолом) персонажа не заважатиме накласти текстуру волоссся на модель, не зважаючи на те, що волоссся знаходиться під шоломом на малюнку. Приховування шару тіней дозволить сформувати чистіші текстури, які будуть готові до накладання на модель.

Наступний персонаж слугуватиме прототипом вигляду супротивників, яких повинен знищити або витіснити гравець (рис. 3.2).

Рисунок 3.2 - Малюнок прототипа персонажа-супротивника в Adobe Photoshop

В моєму проекті передбачається не тільки керування персонажами у вигляді людей, а й ігровою технікою - танками, БМП та бронемашинами.

Але для створення таких моделей, я використовуватиму існуючі креслення у трьох проекціях та референси-фотографії. Тепер час перейти до моделювання.

3.1.2 Моделювання та анімація персонажів

Створені зображення зовнішнього вигляду персонажа імпортую до Blender і встановлюю у площині XZ - таким чином я маю змогу порівнювати і підправляти модель у відповідності з малюнком (у площині YZ - референси анатомії, які я зазвичай приховую, коли перестали бути необхідними).

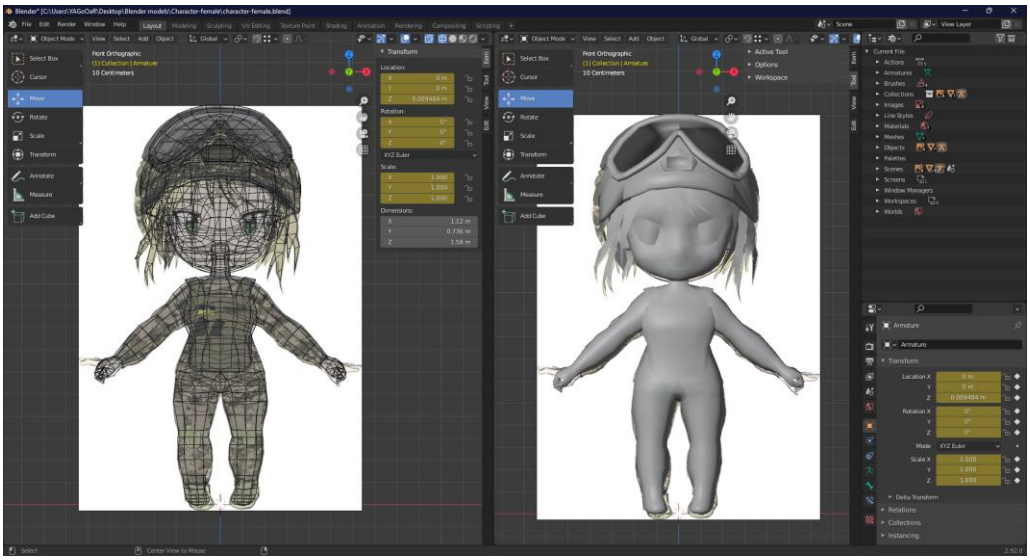


Рисунок 3.3 - Попередній вигляд моделі без текстур в Blender

Моделювати починаю з примітивів, форму яких згодом підганяю до вигляду частин тіла чи одягу персонажа.

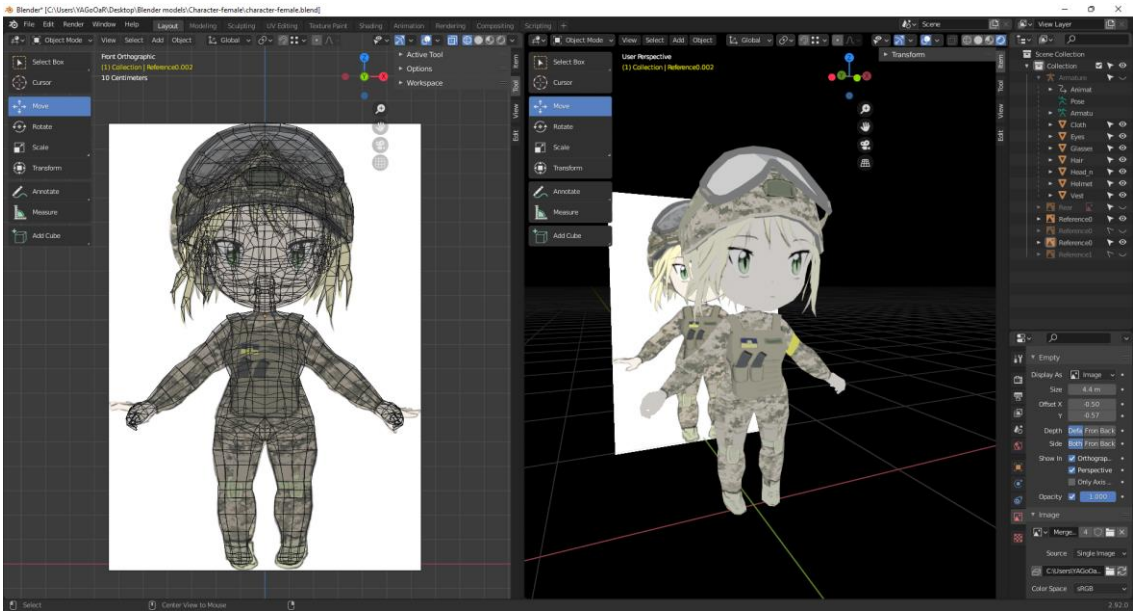


Рисунок 3.4 - Вигляд створеної моделі

Засіб Blender дозволяє зручно проеціювати створений малюнок на полігони розгортки (рис. 3.5).

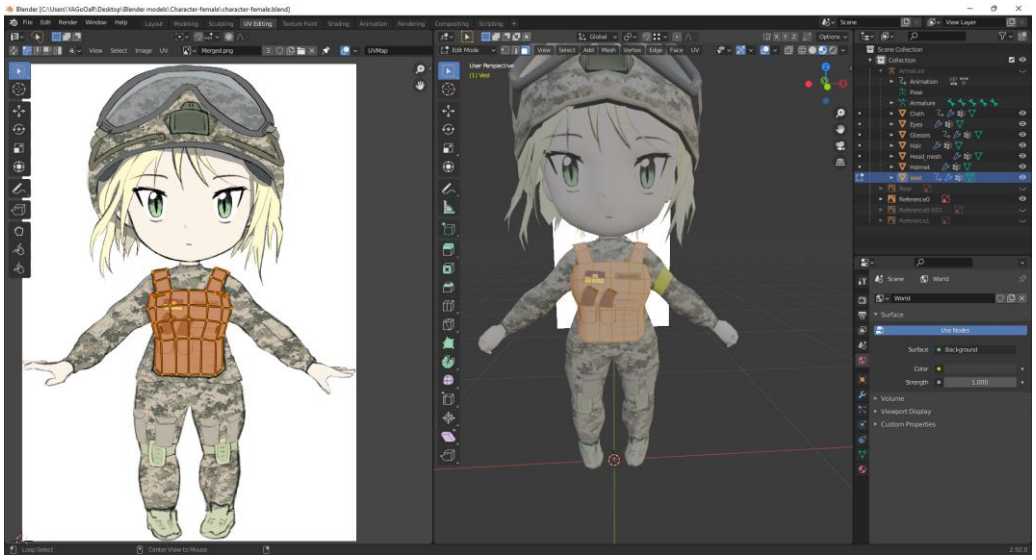


Рисунок 3.5 - Процес текстурування

Текстури встановлюю окремо для кожного об’єкту та шару проекції.

Також, у відповідності з малюнком, моделюю персонажа-супротивника:

Рисунок 3.6 - Моделювання персонажа-супротивника в Blender

Створивши модель, необхідно також призначити персонажу скелет. Процес створення такого скелету називають ріґингом. Скелет дозволяє анімувати персонажа і вирішує проблему руху та повороту різних частин тіла-моделі. Часто доводиться вручну корегувати ваги для вершин сітки моделі, щоб вирішити проблеми з анімацією. Процес роботи з даним інструментом (призначення вагів анімації) - на наступному малюнку.

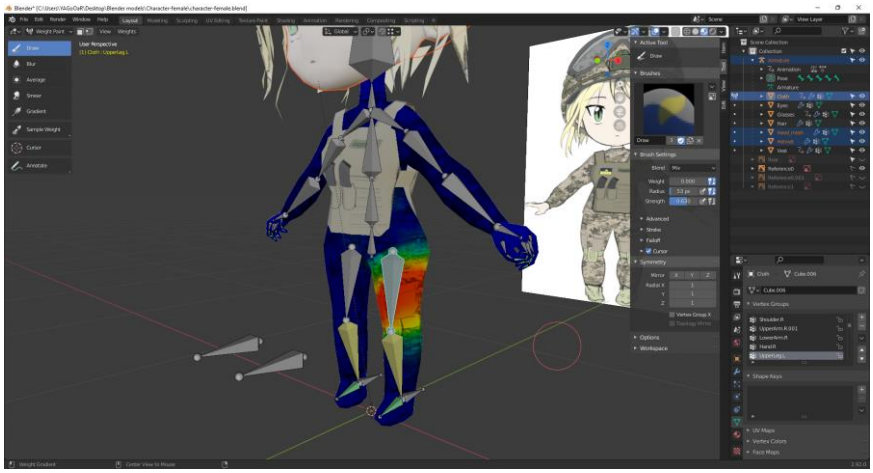


Рисунок 3.7 - Рігинг та призначення вагів для анімації скелета персонажа

Отримавши скелет, персонаж може бути анімований. Я вручну створив по 15 різних анімацій для кожного персонажа: біг та ходіння в різні сторони, стрибок, цілення з гвинтівки, кидання гранати, присідання, визирання із-за укриття, тощо.

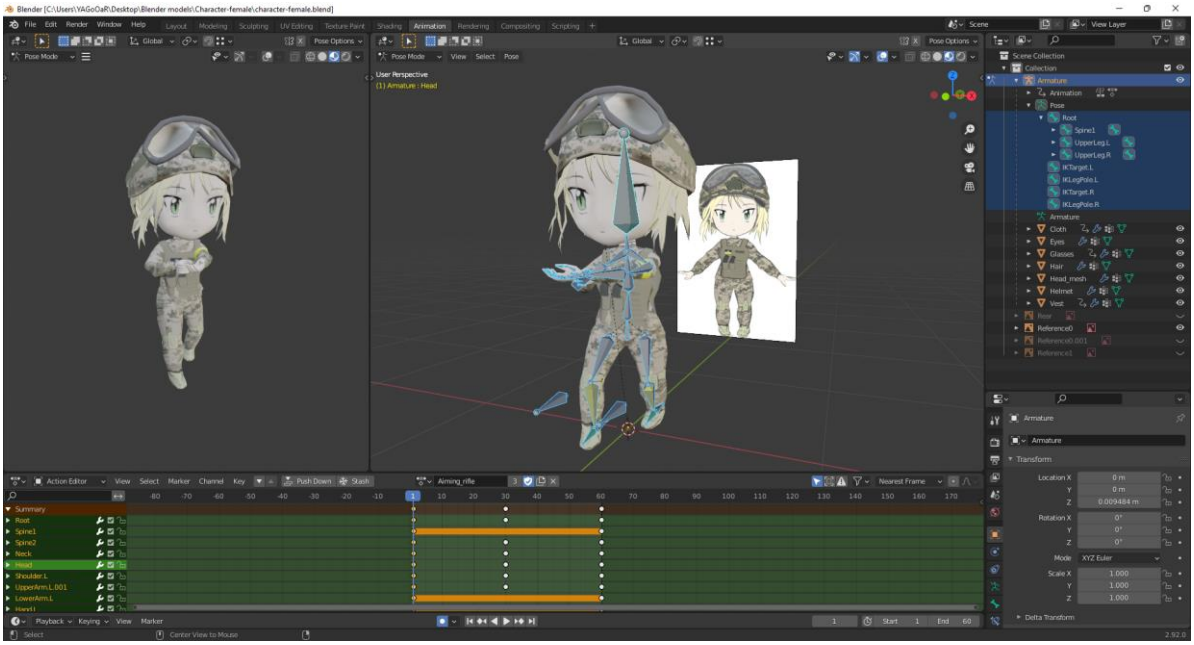


Рисунок 3.8 - Одна з створених анімацій персонажа - поза стрільби

Кожна з цих анімацій передбачає рух персонажа - навіть якщо він стоїть і не здвигається з місця, то він так чи інакше дихає, хитається, тощо. При цьому в грі будуть додані процедурні анімації та плавні переходи між ними - ці функції дозволить реалізувати ігровий двигун, а також написаний код.

На рис. 3.9 - анімація бігу супротивника зі зброєю в руках. Сама зброя змодельована пізніше в ході розробки гри.

Рисунок 3.9 - Одна з анімацій персонажа-супротивника

Тепер маємо готові моделі та анімації персонажів. Можна приступити до моделювання ігрової техніки.

3.1.2 Моделювання ігрової техніки

Ігрову техніку в моєму проекті я моделюю із використанням креслень у трьох-чотирьох проекціях. Відповідні рухливі частини моделей як-от колеса, башта танка, кулемет та гармата, є окремими об’єктами і у грі матимуть свою точку, навколо якої вони повертатимуться за визначеною віссю.

Далі можемо побачити процес розробки прототипу ігрової техніки - машини HMMWV.

На рис. 3.10 видно, що текстури є тимчасово призначеними прямо з креслення машини, щоб краще оцінити візуальний вигляд. Пізніше текстура креслення буде розмальована, щоб модель набула кольору.

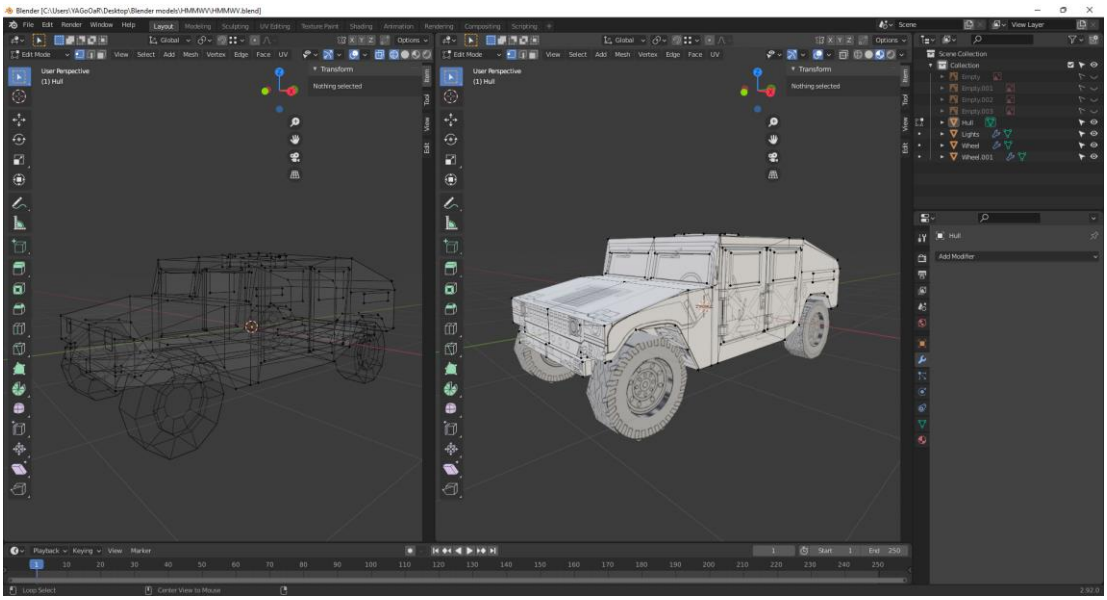


Рисунок 3.10 - Моделювання прототипу HMMWV за кресленням

Техніка, що має гармату або кулемет, у грі має визначену точку, з якої будуть утворюватися кулі та снаряди. Зазвичай я встановлюю це як дуло зброї. БМП-2, наприклад, має автоматичну пушку, кулемет та ПТКР.

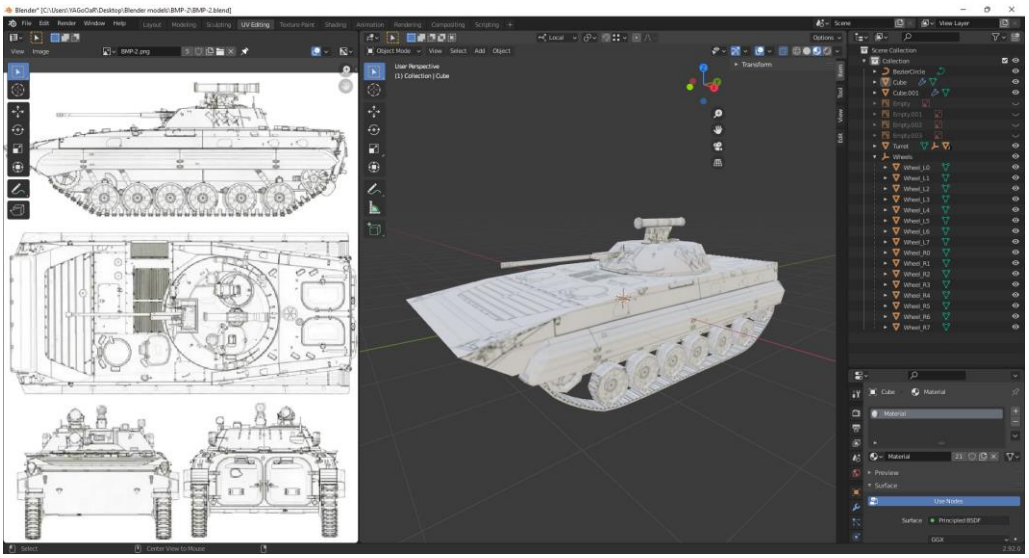


Рисунок 3.11 - Моделювання прототипу БМП-2 за кресленням

Завершена низькополігональна модель танка виглядає наступним чином:

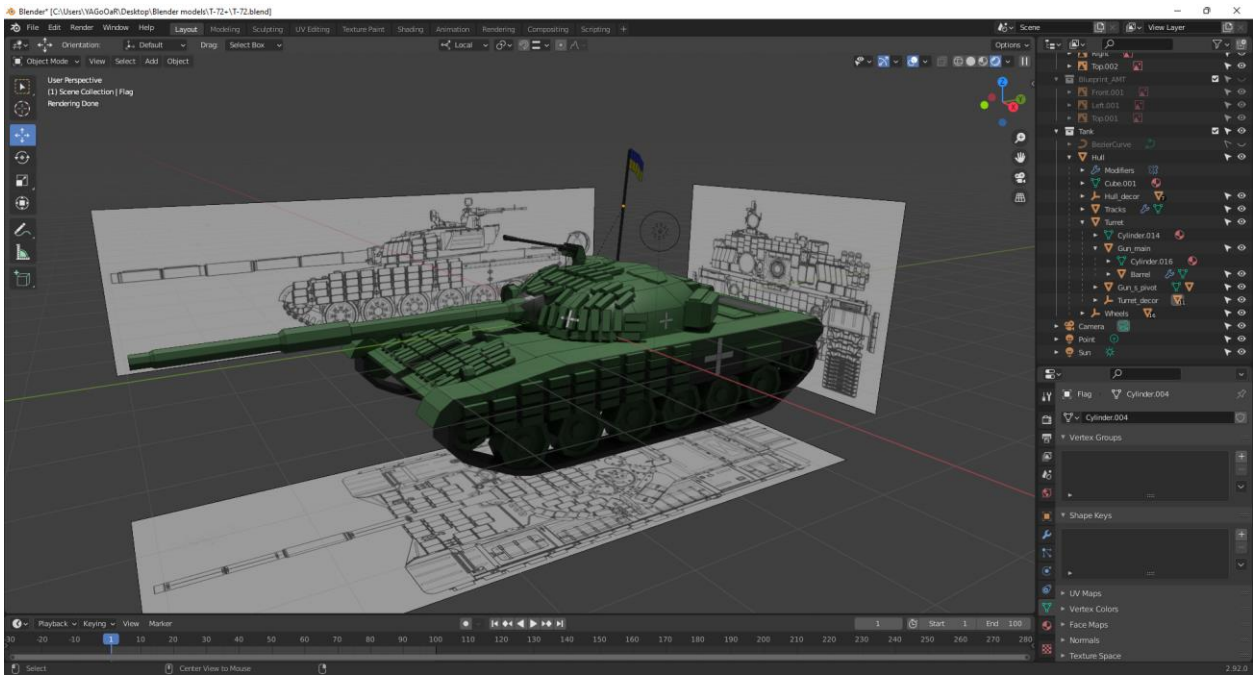


Рисунок 3.12 - Результат моделювання танка Т-72 за кресленням

Звичайно, завжди є над чим попрацювати, але в даному проекті я цілком задоволений створеним мною контентом, який чудово підійде до задуманого візуального стилю гри.

Серед інших моделей, які я створив, є також хатинки, зброя та прості моделі прототипів персонажів та ігрових об’єктів. Тепер можна додати створені моделі у гру.

3.1.3 Створення ігрового оточення в Unity

Поекспериментувавши з шейдерами, можна досягти наступного вигляду створених моделей в рушії в Unity:

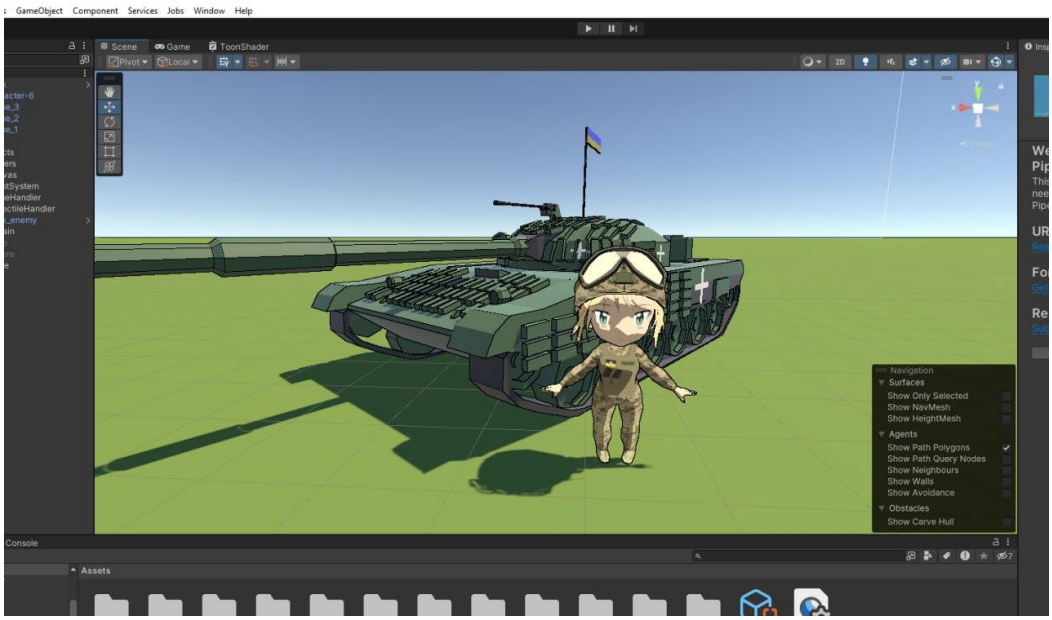


Рисунок 3.13 - вигляд створених моделей у грі

Це є достатньо непоганий візуальний стиль для гри, але в процесі розробки я вирішив надати перевагу більш реалістичному візуальному представленню, тому відмовився від постобробки з доданням контурів. Далі я намалював текстури трави, створив моделі дерев та імпортував матеріали землі та неба з відкритих джерел.

Маючи текстури та моделі ігрового оточення, я створив ігрову карту, що зображена на рис. 3.14.

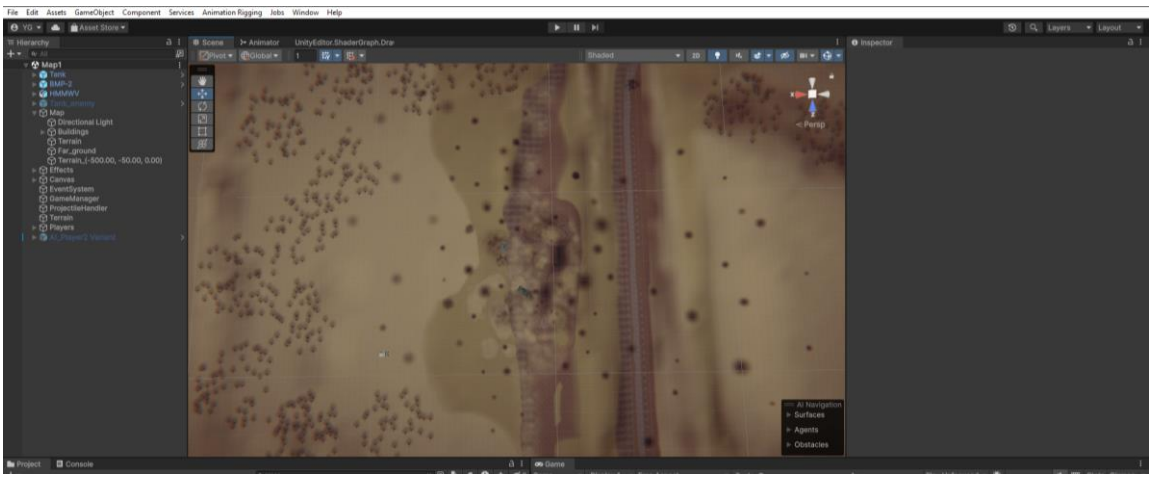


Рисунок 3.14 - Одна з ігрових локацій, вид зверху

Також я додав пост-ефекти для покращення графіки. Маємо результат:

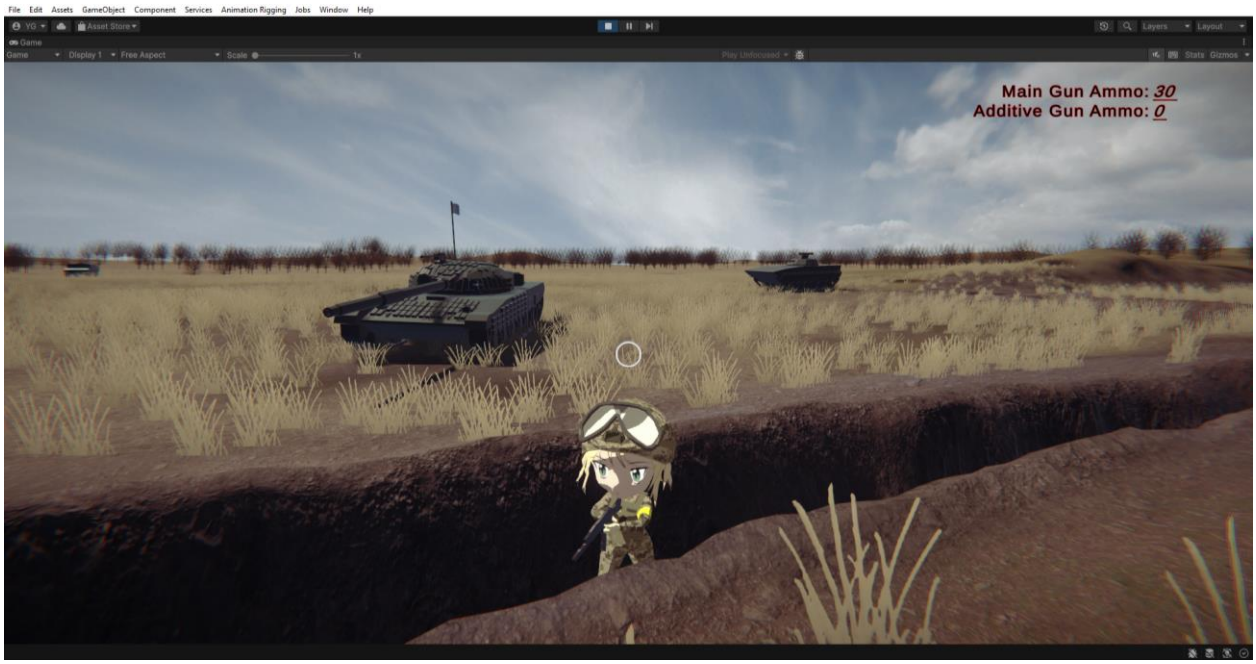


Рисунок 3.15 - Скріншот зі створеної гри

На цьому етапі в мене вже були реалізовані деякі механіки гри. Тепер час розповісти про них.

3.2 Програмна реалізація ігрових механік

3.2.1 Керування персонажем

У шутері віртуальним відображенням гравця є його ігровий персонаж. Те, як він керується і які можливості руху чи дій він має, безпосередньо впливає на відчуття гри (game feel). Керування персонажем повинно мати миттєвий ефект та бути зручним - це позитивно вплине на досвід користувача, та й в цілому, на його бажання грати.

В моєму проекті я реалізував систему руху персонажем, що забезпечує плавне керування на зразок найпопулярніших ігор жанру шутер. Рух персонажа обчислюється з використанням векторної алгебри з метою зручного та

природнього керування вектором швидкості. Опціонально, ця система також дозволяє користуватися механікою bunny hop, що звична багатьом гравцям даного жанру. Фрагмент обрахунку фізики керування наведений на рис. 3.16. Фізика зіткнень персонажа з середовищем обчислюється рушієм.

```
private void FixedUpdate()
{
    if (relativeMovementSpace is null) return;

    bool isOnGround = groundChecker.IsOnGround;

    if (isOnGround != wasOnGround) animatorController.SetJump(!isOnGround);

    UpdatePlayerFriction(isOnGround);

    Vector3 horizontalVelocity = Vector3.ProjectOnPlane(rb.velocity, Vector3.up);

    Quaternion movementRotation = Quaternion.LookRotation(Vector3.ProjectOnPlane(relativeMovementSpace.forward, Vector3.up).normalized);

    Vector3 movementDirection = movementRotation * MathUtils.Vector2To3DSpace(movement);

    Vector3 velocityDirection = horizontalVelocity.magnitude > minVelocity ? horizontalVelocity : transform.forward;
    Vector3 orthogonalControls = Vector3.Project(movementDirection, Vector3.Cross(velocityDirection, Vector3.up));
    Vector3 tangentialControls = Vector3.Project(movementDirection, velocityDirection);

    float velocityAlongControls = Vector3.Dot(rb.velocity, tangentialControls.normalized);

    float lowAccelerationPoint = isOnGround ? maxSpeed : flyMaxSpeed;
    float acceleration = MathUtils.SLikeInterpolation(x: velocityAlongControls, a: lowAccelerationPoint - highAccelerationPoint, b: lowAccelerationPoint);

    float accelerationMultiplier = isOnGround ? moveForce : moveForce * moveForceWhileJumpingMultiplier;

    rb.AddForce(accelerationMultiplier * Time.fixedDeltaTime * (orthogonalControls + tangentialControls * acceleration), ForceMode.VelocityChange);

    Vector3 relativeVelocity = characterTransform.worldToLocalMatrix * rb.velocity;
    Vector2 relativeVelocity2D = MathUtils.Vector3Flatten(relativeVelocity);

    animatorController.SetMovement(character.FirstPerson ? relativeVelocity2D : Vector2.up * relativeVelocity2D.magnitude);

    wasOnGround = isOnGround;
}
```

Рисунок 3.16 - Фрагмент коду контролера руху персонажа

Швидкість та напрямок руху персонажа впливає на його анімації. Під час виду від першої особи, персонаж рухається відносно локального простору камери та рух його голови синхронізований з камерою. Але під часу виду від третьої особи, орієнтація персонажа може змінюватися відносно камери.

Приклади виду від першої особи у розроблюваній грі - на рис. 3.17.

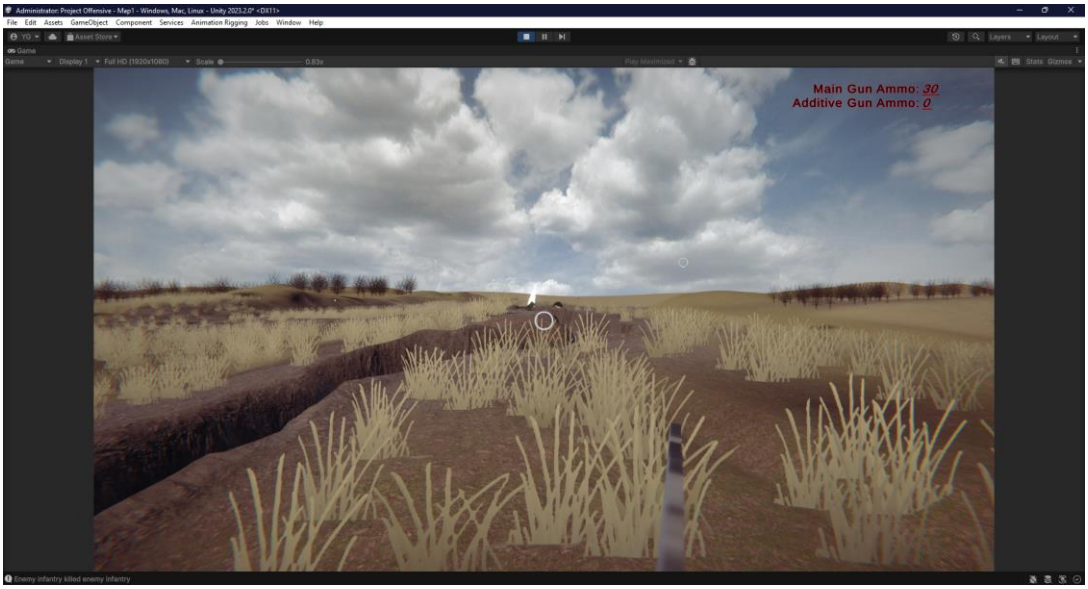


Рисунок 3.17 - Вид від першої особи

На рис. 3.18. - приклад виду гравця від третьої особи. При даному вигляді камера здіймається вверх над персонажем.

Рисунок 3.18 - Вид від третьої особи

Серед дій, доступних гравцю є біг, прицілювання мишею, стрибок, кидання гранати, стрільба, взаємодія з об’єктами (сідання в техніку, тощо), зміна пози, тощо.

```
4 references
public override void SetPlayer(Player player)
{
    base.SetPlayer(player);

    PlayerInput playerInput = player.PlayerInput;
    var actions = playerInput.Infantry.Buttons;
    var axes = playerInput.Infantry.Axes;

    actions[InfantryActions.Jump] = (button) => { if (button) characterController.Jump(); };
    actions[InfantryActions.Shoot] = (button) => {
        gunManager.Shoot(GunType.main, button ? ShootingMode.turnSprayOn : ShootingMode.turnSprayOff);
    };
    actions[InfantryActions.ShootSingle] = (button) => {
        if (button) gunManager.Shoot(GunType.main, ShootingMode.single);
    };

    actions[InfantryActions.ThrowGrenade] = (button) => {
        if (button) gunManager.Shoot(GunType.grenade, ShootingMode.single);
    };

    actions[InfantryActions.AimAbove] = (button) => { if (button) AimAbove(!isAimingAbove); };

    actions[InfantryActions.IgnoreLookPoint] = (on) => animatorController.SetAiming(!on);

    axes[Axes.Horizontal] = (val) => { characterController.SetMovementAxis(0, val); };
    axes[Axes.Vertical] = (val) => { characterController.SetMovementAxis(1, val); };
}
```

Рисунок 3.19 - Фрагмент коду із встановленням дій, доступних персонажу

Створена система вводу дозволяє зручно змінювати та додавати дії персонажа та кнопки контролера. Гнучкість дозволяє створити сторінку налаштувань управління користувача в меню, щоб гравець сам зміг призначити управління, яке йому довподоби.

Приклад структур даних, що містять встановлені кнопки та осі для відповідних дій гравця на рис. 3.20.

```
readonly KeyBindings<InfantryActions> infantryActionBindings = new(
    new()
    {
        { InfantryActions.Shoot, MouseButtonChecker(0) },
        { InfantryActions.AimAbove, MouseButtonChecker(1) },
        { InfantryActions.IgnoreLookPoint, KeyChecker(KeyCode.LeftAlt) },
        { InfantryActions.ThrowGrenade, KeyChecker(KeyCode.G) },
        { InfantryActions.Jump, KeyChecker(KeyCode.Space) },
        { InfantryActions.Crouch, KeyChecker(KeyCode.C) },
        { InfantryActions.EnterVehicle, KeyChecker(KeyCode.F) },
    }
);
readonly KeyBindings<TankActions> tankActionBindings = new(
    new()
    {
        { TankActions.Shoot, MouseButtonChecker(0) },
        { TankActions.ShootAdditive, MouseButtonChecker(1) },
        { TankActions.IgnoreLookPoint, KeyChecker(KeyCode.C) },
        { TankActions.LeaveVehicle, KeyChecker(KeyCode.F) },
    }
);
readonly KeyBindings<GeneralActions> generalBindings = new(
    new()
    {
        { GeneralActions.Zoom, MouseButtonChecker(2) },
    }
);
readonly AxisBindings<Axes> axisBindings = new(
    new()
    {
        { Axes.Horizontal, () => Input.GetAxisRaw("Horizontal") },
        { Axes.Vertical, () => Input.GetAxisRaw("Vertical") },
    }
);
```

Рисунок 3.20 - Фрагмент коду із встановленням клавіш у відповідність діям персонажа

Механіка прицілювання та стрільби працює наступним чином:

- Гравець має курсор (великий кружечок, рис. 3.21), за допомогою якого вказує персонажу точку прицілювання в просторі. (Точка є перетином променя, випущеного з камери в напрямку курсора, з певним фізичним об’єктом)
- Гравець, при управлінні піхотинцем, може стріляти з автомата та кидати гранати.
- Управляючи танком, він може стріляти з гармати та кулемета, автопушки, тощо.

- Персонаж намагається прицілитися зброєю прямо в цю точку, якщо це дозволяє зробити його анатомія.
- Під час руху, хитання персонажа спричиняє відхилення зброї від напрямку на цю точку, тому точніше цілитися стоячи або в позі присідання.
- Після першого пострілу розкид зброї збільшується
- Гравець може натиснути праву кнопку миші, щоб визирнути з-за укріплення, а потім стріляти

Рисунок 3.21 - Прицілювання під час бігу

Процедурні анімації урізноманітнюють рухи персонажа - за допомогою них можна окремо керувати частинами тіла та поєднувати різну поведінку персонажа як-от прицілювання у різні точки та одночасно з цим, біг у різні напрямки. До того ж, можна використовувати інверсивну кінематику, що буде обчислювати положення кісток, щоб персонаж міг тримати зброю кількома руками, коли вона повертається.

На наступній ілюстрації можемо побачити, що персонаж цілиться в одну сторону, а нижня частина тіла направлена в іншу сторону. Процедурна анімація тут повертає руки та голову персонажа, щоб він цілився у задану точку.

Рисунок 3.22 - Процедурна анімація прицілювання

Процедурна анімація налаштована з різними вагами для різних для окремих частин тіла. Зменшені значення вагів повороту за курсором має тіло та голова персонажа, щоб виконувати неповний поворот - якщо курсор сильно відхилений від напрямку вперед (наприклад, 90 градусів), то голова та тулуб повертаються на менший кут так само, як робить людина (решту повороту виконують очі). Руки зі зброєю повертаються точно в напрямку курсору.

3.2 Реалізація ігрового ШІ

Як я вже зазначив у попередніх розділах, в проекті буде використана нейронна мережа для реалізації поведінки NPC-персонажів і для цього використаю технологію Unity ML Agents.

Методом навчання буде навчання з підкріпленням та імітаційне навчання. Крім того, під час тренування моделі будуть змагатися між собою (в командах) та своїми минулими версіями.

Гіперпараметри моделі та методу навчання слід налаштувати у файлі конфігурації поведінки, яку використовуватиме алгоритм навчання ML Agents. Приклад такого файлу я наводжу далі:

```yaml

behaviors:
  InfantryAgentBehaviour:
    trainer_type: ppo
    hyperparameters:
      batch_size: 512
      buffer_size: 8192
      learning_rate: 0.0009
      beta: 0.005
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
      beta_schedule: linear
      epsilon_schedule: linear
    network_settings:
      memory:
        memory_size: 32
        sequence_length: 16
      normalize: true
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      gail:
        strength: 0.7
        demo_path: Assets\Demos\shooting.demo
    behavioral_cloning:
      strength: 0.7
      demo_path: Assets\Demos\shooting.demo
    self_play:
      window: 10
      play_against_latest_model_ratio: 0.5
      save_steps: 50000
      swap_steps: 2000
      team_change: 200000
      initial_elo: 1200.0
    max_steps: 20000000
    time_horizon: 256
    summary_freq: 8000

```

Рисунок 3.23 - Конфігурація поведінки агента

З попереднього рисунку видно, що розмір моделі поведінки агента задана як 2 слої та 128 нейронів у слої. ML Agents також неявно використовує для нейронів функцію активації Swish, тобто аргумент, помножений на синус від нього ж.

Агент діє, базуючись на своїх знаннях та сприйманій інформації з навколишнього світу. Знання агента містяться в параметрах моделі (вагах) та пам'яті (LSTM-елементах). Розмір пам'яті та моделі (кількість шарів, нейронів) визначаються конфігурацією поведінки.

```
0 references
public override void CollectObservations(VectorSensor sensor)
{
    if (me.Dead)
    {
        sensor.AddObservation(new float[101]);
        return;
    }

    Transform myTransform = character.ViewController.transform;

    AmmoManager ammoManager = character.AmmoManager;

    bool hasAmmo = ammoManager.TotalAmmo > 0;

    sensor.AddObservation(myTransform.forward);
    sensor.AddObservation(rb.velocity);
    sensor.AddObservation(hasAmmo);
    sensor.AddObservation(ammoManager.IsReloading);
    sensor.AddObservation(ammoManager.Ammo / 30);
    sensor.AddObservation(ammoManager.GrenadeAmmo > 0);
    sensor.AddObservation(me.HP / me.MaxHP);

    Entity[] enemies = teams.FindClosestToMe(oppositeTeam, myTransform.position, enemiesBufferSensor.MaxObservations);

    Entity closest = enemies.Any() ? enemies[0] : null;

    int enemyObjectsFilled = 0;

    foreach (Entity enemy in enemies)
    {
        Transform target = enemy.transform;
        Vector3 targetPos = target.position + characterCenterOffset;

        if (!character.DoISeeEnemy(targetPos + characterVisionOffset)) continue;

        Vector3 relativeDelta = myTransform.worldToLocalMatrix * (targetPos - myTransform.position);
        Vector3 relativeDir = relativeDelta.normalized;
        float dist = relativeDelta.magnitude;

        Vector3 angleDiff = Quaternion.FromToRotation(Vector3.forward, relativeDir).eulerAngles;

        float angleX = normalizeAngle(Mathf.DeltaAngle(0, angleDiff.y));
        float angleY = normalizeAngle(Mathf.DeltaAngle(0, -angleDiff.x));

        float[] observation = { angleX, angleY, normalizeDistance(dist) };
```

Рисунок 3.24 - Фрагмент коду обчислення значення сенсорів агента

В процесі тренування агента, алгоритм навчання (в моєму випадку алгоритм PPO) намагається оптимізувати метрику нагороди (досягти найвищого значення). Агент виконує дії відповідно до своєї попередньої поведінки з деяким випадковим відхиленням, щоб дослідити результат різних варіантів поведінки та визначити, яка поведінка є ефективнішою. Таким чином, якщо правильно призначати нагороди за результат дій агента, він почне вчитися повторювати та покращувати найліпший отриманий результат. Звісно, агент навчається лише на етапі тренування, а в готовому ігровому застосунку буде застосовуватися вже готова натренована модель. Так працює більшість ШІ рішень в різних сферах інформаційних технологій, наприклад, відомі GPT-моделі - вони є заздалегідь натренованими.

```
1 reference
public void OnMeDied(Entity attacker)
{
    if (attacker == world) return;

    // Death penalty
    AddReward(-0.2f);
}

1 reference
public void OnSomeoneKilled(Entity victim)
{
    if (victim == me || victim.team != oppositeTeam) return;

    // Kill reward
    AddReward(0.4f);
}

1 reference
public void OnEnemyDamaged(Entity victim)
{
    // Damage reward
    AddReward(0.4f);
    // Distance reward
    AddReward(0.4f * Mathf.Clamp01((victim.transform.position - me.transform.position).magnitude/40f));
}

1 reference
public void OnHeal()
{
    // Healing and reloading on base reward
    AddReward(0.4f);
}
```

Рисунок 3.25 - Фрагмент коду видачі нагород агенту

Нагороди за вбивства та покарання за смерть стимулюють агента виконувати ефективні дії ведення бою. Нагорода за спільну перемогу змушують агентів діяти як команда.

Крім вказаних нагород, я додав агенту покарання за кожен кадр симуляції, щоб покарання акумулювалось з часом. Таким чином агенти краще старатимуться виконати епізод тренування якомога швидше, щоб не втрачати отримані нагороди. Також були додані нагороди за точне наведення на ціль та покарання за неправильний напрямок прицілювання, щоб пришвидшити навчання на ранніх етапах, проте я їх вимикаю при подальшому навчанні. Нагорода за виграш надається всій команді агентів, щоб стимулювати командну гру.

Важливо в даному процесі також лишити агенту простір для самостійного розвитку у різних напрямках - не обмежувати його поведінку великою кількістю умовних нагород та покарань, а давати нагороди саме за результат. Інакше код видачі нагород мало чим відрізнятиметься від вручну прописаної поведінки, і тому сенсу застосування ШІ не лишиться, бо агент оптимізуватиме свою поведінку під задані умови, а не під головну ціль та її результат.

Наприклад, в ході навчання може виявитись, що наведення персонажа прямо на найближчу ціль не завжди є найефективнішим способом перемоги. Наприклад, ефективнішим може стати стрільба по супротивникам з найменшою кількістю здоров’я (щоб швидше усувати небезпеку), навіть якщо він стоїть поза своїми більш здоровими учасниками команди.

Ще одним прикладом є прицілювання вгору від супротивника та кидання гранати на велику відстань, що дозволить перекинути певну перешкоду. Саме тому не слід обмежувати агента конкретними нагородами та покараннями за дії, а краще давати нагороди за сам результат. Проте, такі нагороди все ж можна використовувати на ранніх етапах навчання, щоб його пришвидшити.

Прикладами креативної поведінки агента можу навести наступні ілюстрації (рис. 3.26, рис. 3.27). На них можна побачити одне з середовищ

тренування агентів, які я створив. Тут наявні окопи та перешкоди, серед яких агенти повинні навчитися воювати та перемагати супротивників.

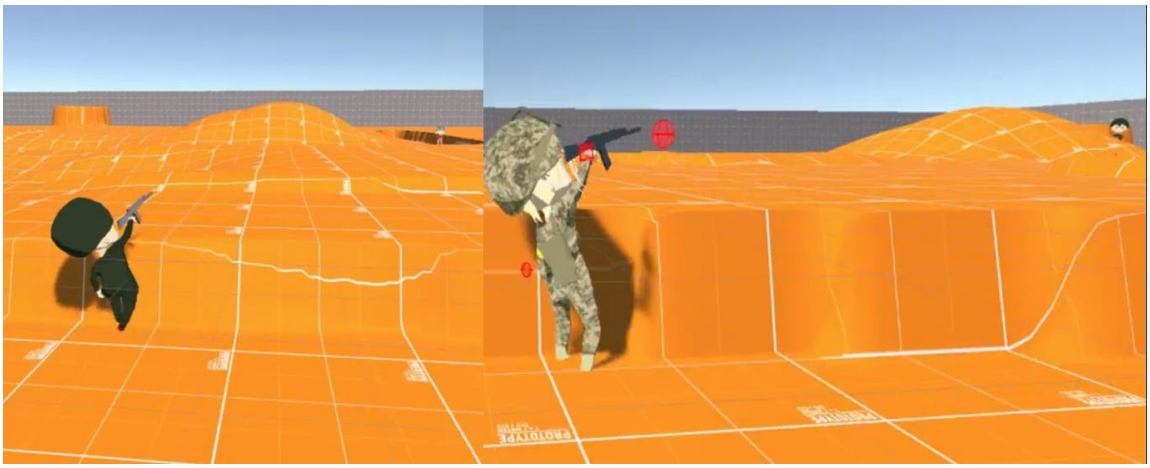


Рисунок 3.26 - Агенти визирають з укриття та стріляють по своєму супротивнику

Агенти вміють ховатися за перешкодами, щоб в них було складніше попасти - це збільшує їх ефективність. На рис. 3.27 один з агентів ховається, щоб не отримати покарання за смерть. Таку поведінку я усунув, додавши покарання агентам за проведений час гри.

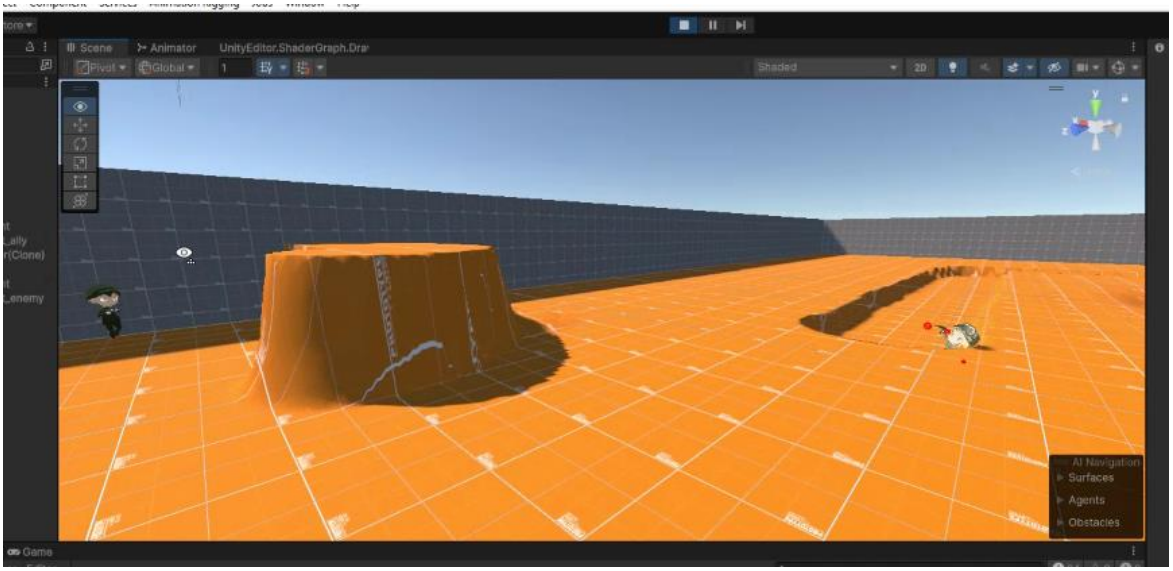


Рисунок 3.27 - Агенти ховаються та очікують супротивника

Для зменшення навантаження на локальну машину, я додав спрощені моделі персонажів, щоб мати змогу навчати велику кількість агентів одночасно. На рис. 3.28, зправа можна побачити гранату, яку успішно закинула червона команда, врезультаті чого одразу знищила декілька синіх гравців.

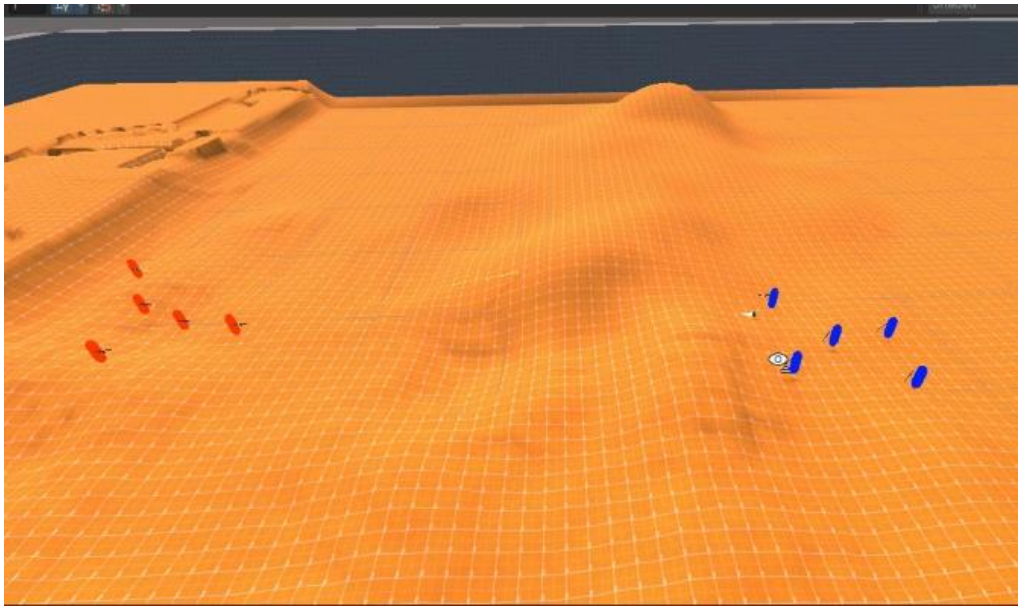


Рисунок 3.28 - Командний бій агентів

Після цього синя команда робить свою спробу (рис. 3.29).

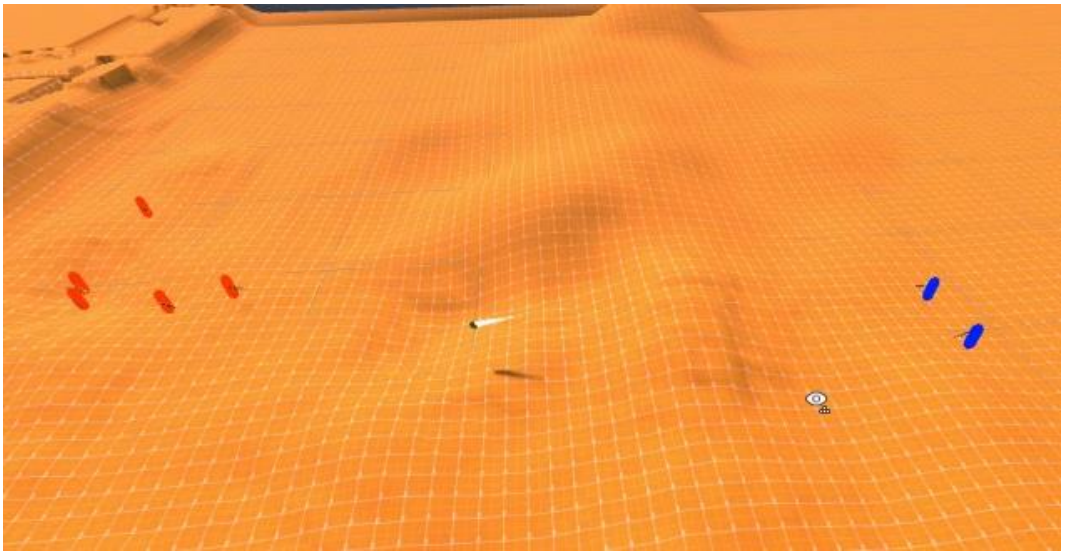


Рисунок 3.29 - Кидання гранати агентами

Для збирання даних для імітаційного навчання та клонування поведінки ML Agents дозволяє використати компонент запису демонстрації поведінки агента - Demonstration Recorder.

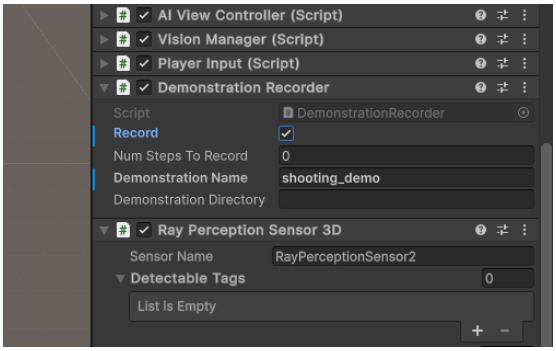


Рисунок 3.30 - Компонент запису демонстрацій агента для ML Agents

Під час запису демонстрації, я виконую роль агента та «показую йому, як себе слід поводити». Тобто просто збираю дані для датасета поведінки.

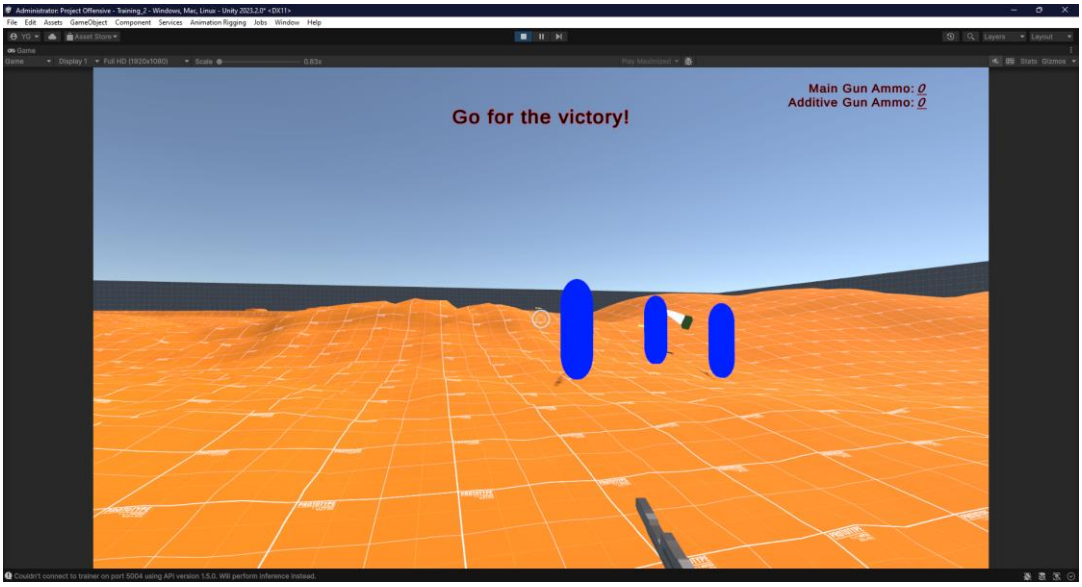


Рисунок 3.31 - Взяття на себе ролі агента для запису демонстрації

Тренування агентів я виконую одночасно в багатьох середовищах, що суттєво пришвидшує процес. Нижче я надаю приклад тренування агентів стрільбі по цілям одночасно в 16 середовищах.

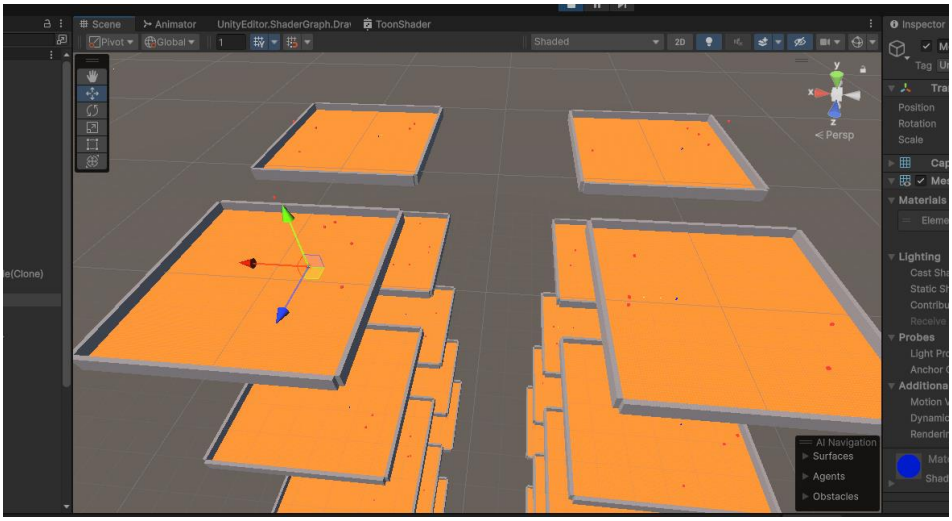


Рисунок 3.32 - Середовища тренування агентів

З метою навчання моделей Unity спілкується з тренером ML Agents, що використовує Pytorch. Прийняття рішень агента під час та за межами тренування обчислюється за допомогою механізму прийняття рішень - Barracuda.

Рисунок 3.33 - Вигляд консольної програми навчання агентів

3.3 Робота над оптимізацією гри

Як я вже говорив раніше, однією з причин використання зроблених мною вручну 3Д моделей є контроль над їх складністю геометрії. Коли розробник використовує запозичені моделі, він може ненавмисно обрати високоякісні моделі, які сильно навантажуватимуть графічний процесор - з цієї причини страждає оптимізація деяких існуючих ігор.

Отже, першим методом, що я застосовую, є створення low-poly моделей - низькополігональних моделей з відносно малою кількістю геометрії, яка використовує мало ресурсів відеокарти і при цьому підходить під візуальний стиль гри.

Рисунок 3.34 - Порівняння моєї простої моделі та високодеталізованої моделі, зробленої іншою людиною

Високодеталізовані моделі можуть застосовуватись разом з використанням технології Level Of Detail (LOD) як перший рівень деталізації, проте всеодно потребуватимуть потужний комп’ютер для гри. Часто налаштування графіки у іграх впливає на рівні деталізації та складність геометрії ігрових об’єктів. Проте, створення одразу низькополігональних моделей є значно простішим рішенням для імплементації, тому я його обрав.

Другим методом, що я застосовую, є саме технологія Level Of Detail.

Рисунок 3.35 - Рівні деталізації дерева

Модель дерева змінює кількість полігонів та врешті решт перетворюється на пласку картинку, по мірі віддалення камери від неї. Для цього я створював та спрощував модель дерева, а потім робив рендер його вигляду як картинку (billboard). Білборд відображається за допомогою спеціального шейдера. Решту роботи виконує технологія Unity LOD.

Таким чином можна деталізувати близькі до гравця об’єкти і зекономити на деталях об’єктів, які персонаж бачить вдалині. Це дуже значно економить ресурси відеокарти. Таку деталізацію слід також застосувати до танка та персонажів, коли масштаб проекту збільшиться (зараз в цьому не так багато сенсу, бо одночасних персонажів та техніки на ігровій карті далеко не так багато, як дерев, тому вплив на оптимізацію мінімальний).

ВИСНОВОК ДО РОЗДІЛУ 3

В цьому розділі було описано хід роботи над проектом - реалізацію ігрових механік, створення ігрового контенту, особливості застосування машинного навчання та роботу над оптимізацією гри.

Створення контенту полягало в малюванні текстур, моделюванні персонажів та техніки, ріґингу, анімації. Із створеним контентом було налаштовано ігрові сцени, на яких проводиться бій між гравцем та ШІ-персонажами.

За допомогою рушія Unity та мови програмування C# було реалізовано ігрові механіки прицілювання персонажа (зокрема, з укриття), водіння технкіки, руху та стрільби з різної зброї, тощо.

За допомогою технології ML Agents було створено модель поведінки агента та навчено модель за допомогою методу навчання з підкріпленням, імітаційного навчання та клонування поведінки.

РОЗДІЛ 4. ОГЛЯД ТА ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОДУКТУ

4.1 Огляд розробленої гри

Запустивши ігровий застосунок, після заставки користувач бачить головне меню гри.

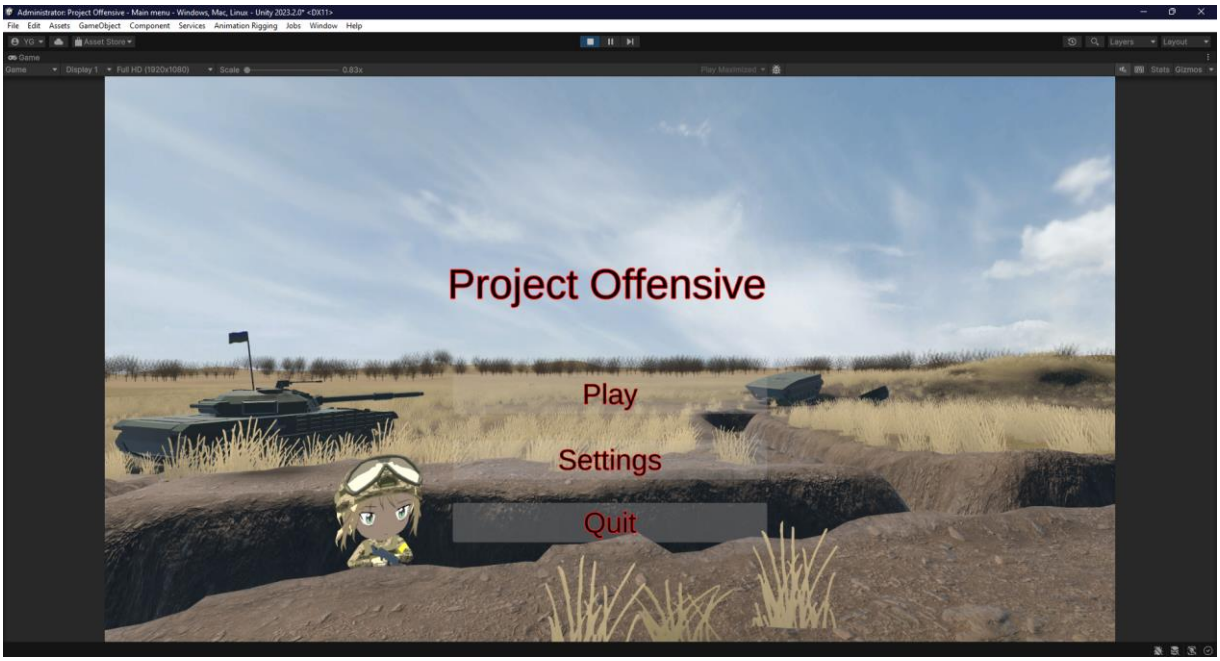


Рисунок 4.1 - Головне меню гри

Користувач може почати гру, відкрити налаштування звуку/управління або вийти з гри. Гра складається з декількох рівнів, під час проходження кожного з яких, ціллю гравця є зайняти позиції супротивника, знищивши або витіснивши супротивників. Гравець може штурмувати позиції, будучи піхотинцем, або керуючи танком. На рис. 4.2 можемо побачити приклад такої позиції.

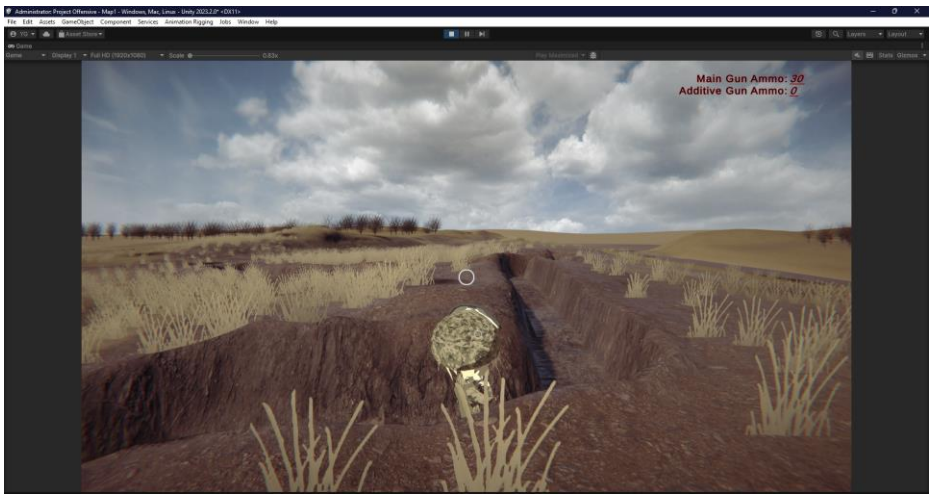


Рисунок 4.2 - Приклад позицій супротивника - траншеї

Персонаж може сідати в наявну бронетехніку та вилазити з неї. Для цього гравець повинен натиснути відповідну кнопку контролера (чи клавіатури). За замовчуванням, це є клавіша «F».

Залізши в танк, гравець може керувати баштою та кулеметом ігровим курсором (великий кружечок). Поточний напрямок головної зброї відмічено малим кружечком. Тобто, так само, як з піхотинцем, що було описано в попередньому розділі на етапі розробки. Стрільба з головної зброї виконується внаслідок натискання лівої кнопки миші за замовчуванням. Для кулемета - права кнопка миші за замовчуванням.

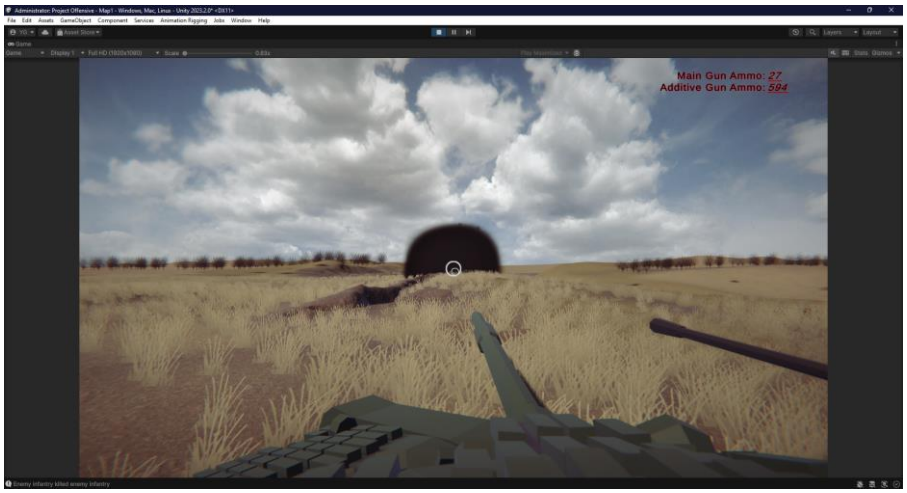


Рисунок 4.3 - Стрільба з гармати танка по укріпленнях

Снаряд танка, влучивши в землю, має визначений радіус дії, в якому всім ігровим об’єктам, що мають компонент Entity, наноситься шкода. Згаданий компонент я створив з метою додання системи здоров’я персонажам та іншим об’єктам.

Кулемет наносить шкоду об’єктам лише при прямому попаданні.

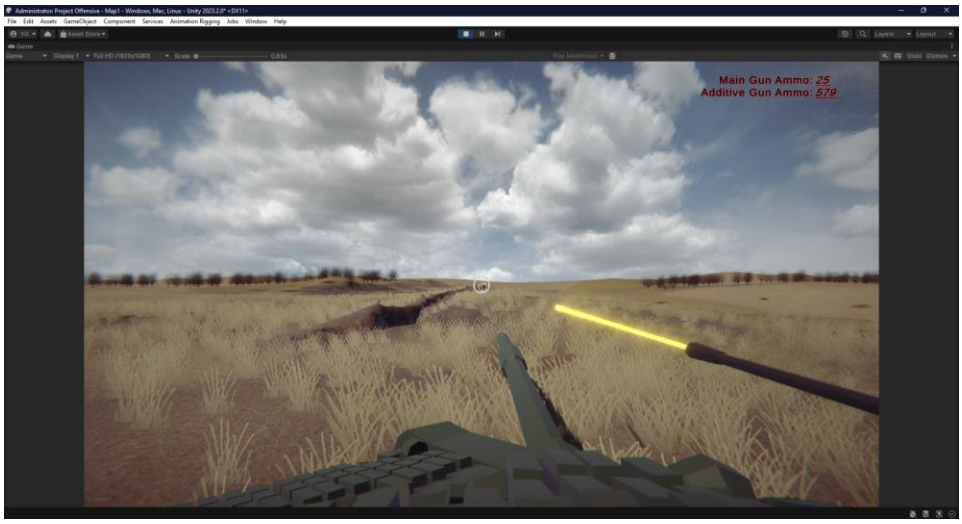


Рисунок 4.4 - Стрільба з кулемета танка по укріпленнях

Подібно кулемету танка, автомат піхотинця завдає шкоду лише при прямому влученні в ціль. Шкода, завдана автоматом та його точність є меншою, ніж відповідні параметри в кулемета танка.

Рисунок 4.5 - Стрільба з автомата по укріпленнях

Проте, в піхотинця є свої переваги. По-перше, він є менш помітною та пріоритетною ціллю, а по-друге, він може засідати в траншеях та окопах і вести вогонь з них.

Знаходячись в траншеї, гравець може піднімати автомат вгору, що дозволить йому стріляти через укриття. Кнопкою даної дії за замовчуванням є права кнопка миші.

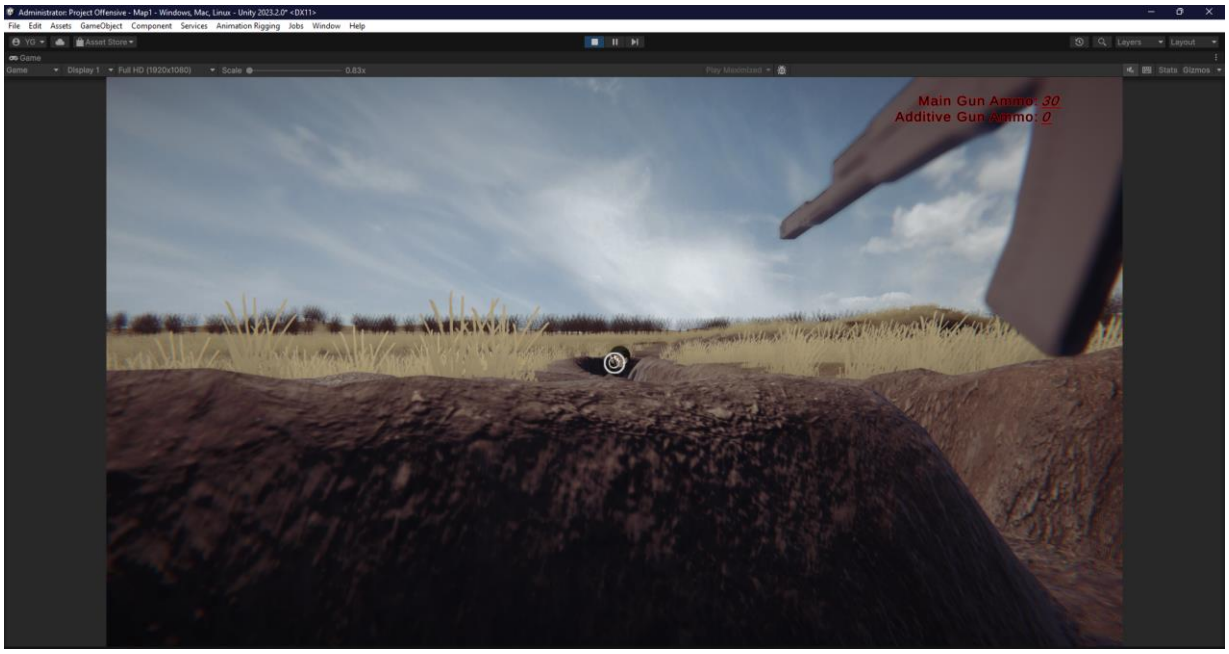


Рисунок 4.6 - Бій в траншеях

Гравець, як і інші піхотинці, може кидати гранату. Кнопкою даної дії за замовчуванням є клавіша «G». Далі (рис. 4.7) я наводжу приклад кидання такої гранати. Як ігрова умовність, граната має білий трасер, щоб її було простіше побачити.

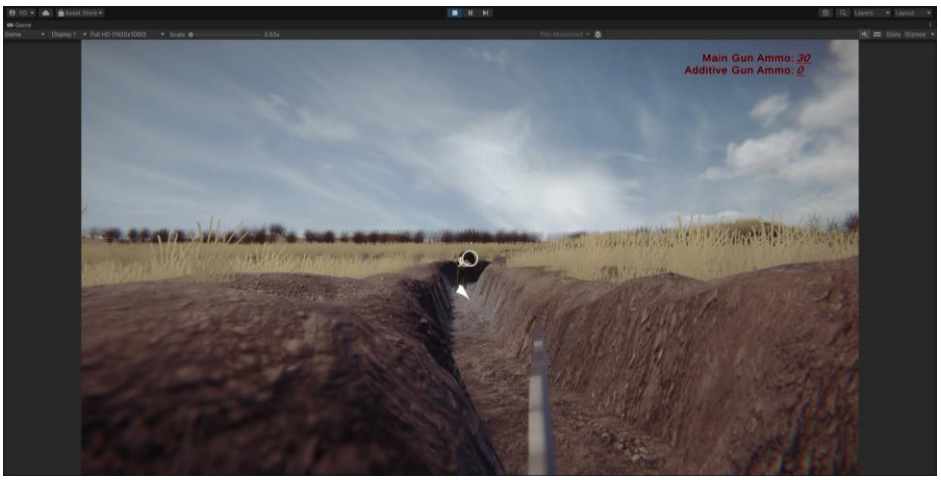


Рисунок 4.7 - Кидання гранати в супротивника

На рис. 4.8 - результат кидання гранати - вибух та знищення супротивника.

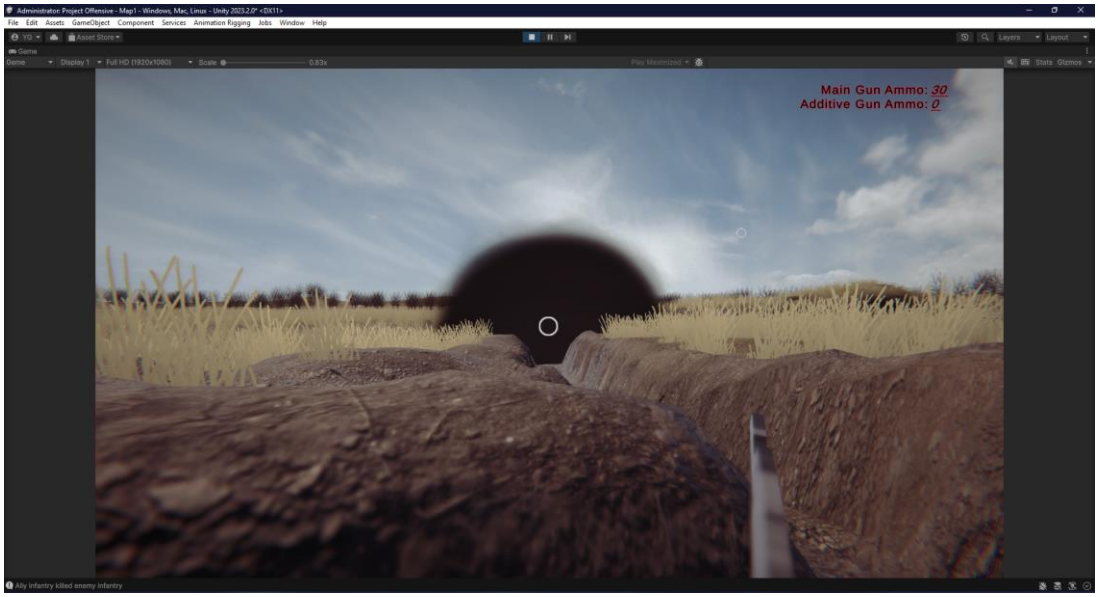


Рисунок 4.8 - Супротивник знищений гранатою

Шкода, нанесена гранатою, залежить від відстані персонажа до вибуху.

4.2 Тестування інтерфейсу гри

Для завершення роботи, слід протестувати стани програми, до яких може перейти користувач (див. Додаток 3), щоб бути впевненим, що застосунок працює коректно.

Першим, вже відомим станом є головне меню гри.

Рисунок 4.9 - Перехід до налаштувань

Спробуємо перейти на сторінку налаштувань:

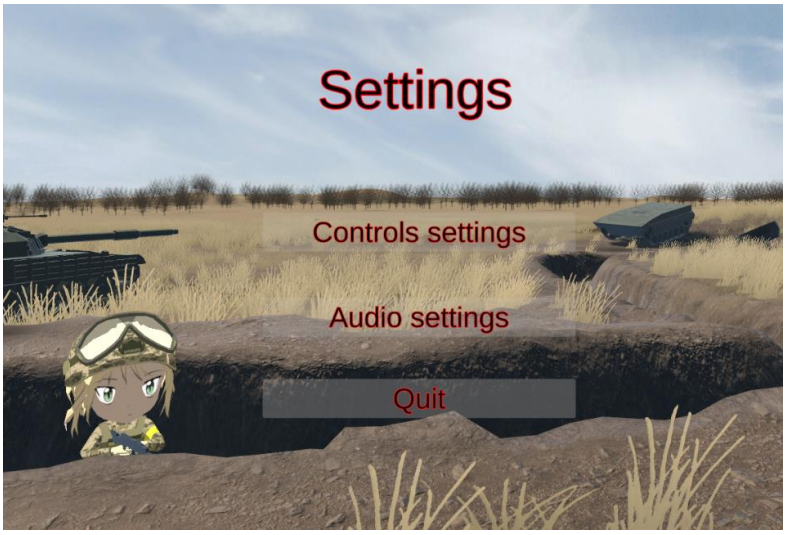


Рисунок 4.10 - Перехід до налаштувань гри

Тут ми бачимо сторінку налаштувань гри, та можливості налаштування аудіо або управління. Переходимо до налаштувань аудіо. Порухавши слайдер на сторінці, можна побачити, що він працює і регулює гучність звуку.

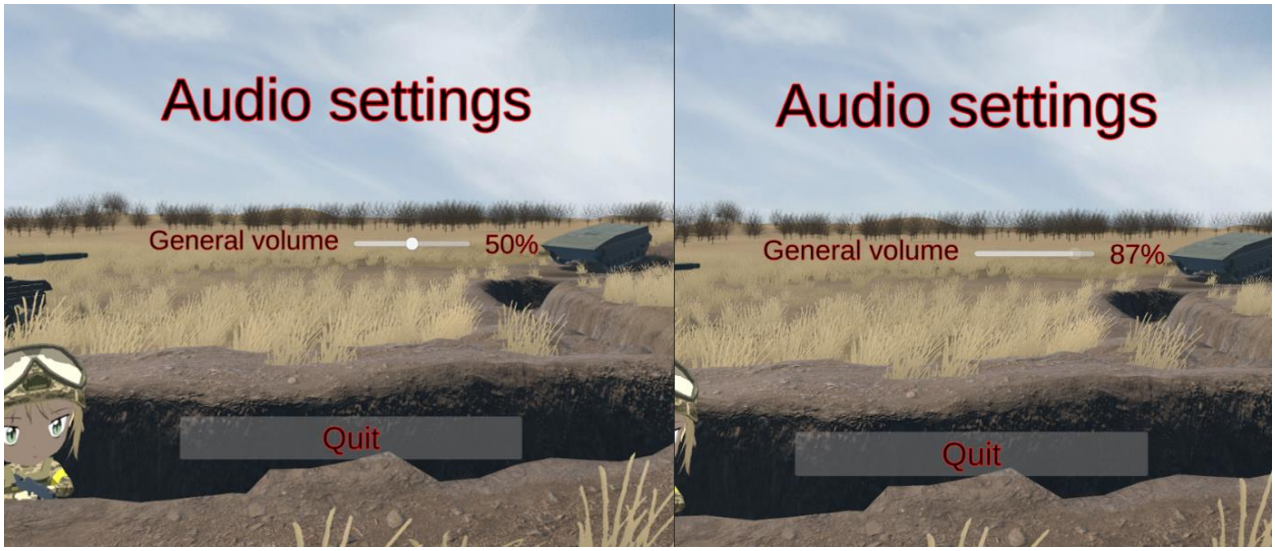


Рисунок 4.11 - Налаштування аудіо

Звідси можемо повернутися в налаштування гри і відкрити сторінку налаштувань управління.

Рисунок 4.12 - Налаштування управління

Звідси можна повернутися до меню налаштувань та головного меню. Проте, ми їх вже бачили, тому ілюструвати знов я не буду.

З меню гри користувач може натиснути кнопку виходу. В такому випадку програма завершується за допомогою визову функції Application.Quit().

При натисканні кнопки «Play», відкривається наступн а сторінка, що дозволяє завантажити обраний рівень.

Рисунок 4.13 - Меню вибору рівня

Коли користувач робить вибір, завантажується відповідний рівень гри. В процесі гри, користувач може натиснути паузу. Цей стан виглядає наступним чином:

Рисунок 4.14 - Сторінка паузи гри

Аналогічно до головного меню, можна відкрити налаштування. Також можна повернутися до головного меню, або продовжити гру.

Рисунок 4.15 - Сторінка налаштувань під час паузи

Рисунок 4.16 - Сторінка налаштувань аудіо під час паузи

Із загальних налаштувань можна відкрити налаштування управління або аудіо. На малюнку вище, наприклад, сторінка аудіо, відкрита під час паузи.

Отже, було розглянуто всі стани ігрового інтерфейсу системи. Вони відповідають схемі у додатку 3.

ВИСНОВОК ДО РОЗДІЛУ 4

В цьому розділі був зроблений короткий огляд створеної гри та перевірено роботу доданих механік гри. Також було перевірено роботу інтерфейсу користувача, а саме головного меню, паузи та їх сторінок. В поточних версіях програми непередбаченої поведінки не виявлено, бо вона одразу була виправлена і доведена до задуманної, по мірі знаходження невідповідності між наявною та цільовою.

Проте це ніколи не означає, що недоліків зовсім нема. Як і в будь-якого проекту, розробник має обмежені ресурси – час та гроші, розмір команди, тощо. Тому масштаб та популярність гри часто виявляються кращими у великих команд. Але існує так звана категорія інді-ігор, що завжди має свою аудиторію. Такі ігри розробляються малою командою, або ж одним розробником та зазвичай є невеликими, але часто цікавими проектами. Оскільки єдиним розробником гри є я, то моя гра належить до даної категорії. У порівнянні з такими іграми, з досвіду можу сказати, що проект має великі перспективи подальшої розробки.

Звичайно, створену гру вже можна вважати готовим продуктом, проте, вона може і буде мати подальший розвиток. Наприклад, додання нових рівнів збільшить об’єм геймплею гри, а додавання нових видів техніки дозволить гравцям використовувати більше різних цікавих тактик, щоб пройти рівень, а також розширить ігрову аудиторію.

Через застосування нових технологій при розробці мною гри, комерційні проекти-аналоги, що використовують нейронні мережі в подібних цілях і при цьому мають спільний жанр (шутер), відсутні. Варто зауважити, що є аналогічні застосування агентів у різних модифікаціях існуючих ігор, але їх я не враховую, бо оригінальна гра не використовувала нейронні мережі, тому не підлягає

розгляду. Саме тому гра може бути виділена з-поміж інших конкурентів нестандартним підходом до реалізації супротивників у грі, що зацікавить користувача. Іншою вагомою особливістю є власноруч створений візуальний стиль гри та контент, а також увага, приділена простій, але, на мою суб'єктивну оцінку, красивій графіці.

ВИСНОВКИ

В процесі написання дипломної роботи було створено комп'ютерну гру. Вона належить до жанру шутер та має ігрові механіки, притаманні даному жанру. Відповідно до функціональних вимог, також була реалізована поведінка персонажів-суперників у грі.

Перед початком розробки, було зроблено огляд предметної області та існуючих рішень. Розглянуті існуючі застосунки я дослідив та визначив їх переваги та недоліки. Також було виділено плюси та мінуси використання різних методів ШІ. Обраними методами навчання агентів був метод навчання з підкріпленням, а саме PPO, як найбільш універсальний та стабільний. Його недолік в потребі великої кількості зразків компенсувався створенням великої кількості паралельних тренувальних середовищ. Додатково було використано імітаційне навчання та клонування поведінки, щоб пришвидшити тренування та надати поведінці агента ознаки гри людини.

Основний ігровий контент – текстури, моделі, анімації для своєї гри, було створено власноруч і застосовано для цього інструменти Adobe Photoshop та Blender. В результаті отримано низькополігональні моделі з накладеними текстурами, що добре підходять стилю навколишнього середовища.

Ігровим рушієм було обрано Unity, насамперед як двигун, що є найбільш зручним для реалізації ігрового ШІ, бо має технології ML Agents та Unity Barracuda, а також, на мою думку, є найбільш простим в освоєнні через те, що має безліч навчального матеріалу та велику спільноту. Мова C# була обрана, як єдина така, що підтримується сучасною версією Unity.

Із використанням створеного контенту та ігрового рушія, було побудовано ігрові карти (сцени), що є середовищами для гри користувача та/або ШІ-агентів.

Використовуючи обрані методи, я навчив ШІ агентів виконувати поставлені цілі – знищувати персонажа-гравця, захоплювати й утримувати стратегічні позиції на ігровій карті.

Також гра має головне меню, сторінку паузи та налаштування гри, що задовольняє відповідну вимогу.

Виділення проекту серед конкурентів забезпечуються унікальністю підходу до імплементації ігрового ШІ та візуальним стилем гри. Перспективи подальшої розробки забезпечуються вдалою платформою для створення контенту та простором для розвитку функціоналу, що може бути зручно доданий завдяки створеній системі.

Проект задовольняє всі поставлені функціональні вимоги, які були поставлені в роботі.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The Video Game Explosion: A History from PONG to Playstation and Beyond [Електронний ресурс] – 2008. – Режим доступу до ресурсу: https://books.google.com.ua/books?id=XiM0ntMybNwC&printsec=frontcover
2. What Makes a Good Video Game? 4 Key Elements [Електронний ресурс] // Pluralsight – 2014. – Режим доступу до ресурсу: https://www.pluralsight.com/blog/film-games/what-makes-a-great-game-the-key-elements-of-successful-games
3. Computer Games [Електронний ресурс] // Nanyang Technological University – Режим доступу до ресурсу: https://www3.ntu.edu.sg/home/asschui/Computer%20Games.PDF
4. Genre and game studies: Toward a critical approach to video game genres [Електронний ресурс] // University of Melbourne – 2006. – Режим доступу до ресурсу: https://journals.sagepub.com/doi/pdf/10.1177/1046878105282278
5. Action games [Електронний ресурс] // AllGame – Режим доступу до ресурсу: https://web.archive.org/web/20141209041709/http://allgame.com/genre.php?id=20
6. Uncovering the Top 5 Types of Gamers in Today's Gaming Community [Електронний ресурс] // Ganknow – Режим доступу до ресурсу: https://ganknow.com/blog/types-of-gamers/
7. Deeper and Wider [Електронний ресурс] // Nintendo E3 – 2011. – Режим доступу до ресурсу:

https://web.archive.org/web/20110608122917/http://iwataasks.nintendo.com/interviews/index.html?disableNav=true/#/e32011/newhw/0/6

8. GameDesigning - Video Game Mechanics for Beginners [Електронний ресурс] // Gamedesigning – 2023. – Режим доступу до ресурсу: https://www.gamedesigning.org/learn/basic-game-mechanics/
9. Intelligent agent definition [Електронний ресурс] // TechTarget – 2019. – Режим доступу до ресурсу: https://www.techtarget.com/searchenterpriseai/definition/agent-intelligent-agent
10. Finite State Machine in Game Development [Електронний ресурс] – 2021. – Режим доступу до ресурсу: https://www.researchgate.net/publication/355518086_Finite_State_Machine_in_Game_Development
11. Half-Life [Електронний ресурс] // Metacritic – Режим доступу до ресурсу: https://www.metacritic.com/game/pc/half-life?ftag=MCD-06-10aaa1f
12. The AI from Half-Life's SDK in Retrospective [Електронний ресурс] // AiGameDev – 2008. – Режим доступу до ресурсу: https://web.archive.org/web/20170208220517/http://aigamedev.com/open/article/halflife-sdk/
13. Outsmarting The Covenant: The AI of Halo 2 [Електронний ресурс] // Medium – 2017. – Режим доступу до ресурсу: https://web.archive.org/web/20191208062134/https://medium.com/the-cube/theaiofhalo2-33e824209a4c
14. What Is a Game Engine? [Електронний ресурс] // How-To Geek – 2023. – Режим доступу до ресурсу: https://www.howtogeek.com/888619/what-is-a-game-engine/

15. GameMaker Studio. Офіційний сайт. [Електронний ресурс] – Режим доступу до ресурсу: https://gamemaker.io/en
16. Welcome to unity [Електронний ресурс] // Unity – Режим доступу до ресурсу: https://unity.com/our-company
17. Unreal Engine 5. Офіційний сайт. [Електронний ресурс] // Unreal Engine – Режим доступу до ресурсу: https://www.unrealengine.com/en-US/unreal-engine-5
18. Unrecord. Сторінка у соц. мережах. [Електронний ресурс] // Twitter – Режим доступу до ресурсу: https://twitter.com/unrecordgame
19. Godot Engine. Офіційна документація. [Електронний ресурс] // Godot Engine – Режим доступу до ресурсу: https://docs.godotengine.org/en/stable/about/introduction.html#about-godot-engine
20. GDScript reference. Офіційна документація. [Електронний ресурс] // Godot Engine – Режим доступу до ресурсу: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html
21. Unity 2023.1 Beta - Feature Highlights [Електронний ресурс] // Unity Forum – 2023. – Режим доступу до ресурсу: https://forum.unity.com/threads/unity-2023-1-beta-feature-highlights.1388922/
22. ML Agents Toolkit [Електронний ресурс] // GitHub – Режим доступу до ресурсу: https://github.com/Unity-Technologies/ml-agents
23. ML Agents overview [Електронний ресурс] // GitHub – Режим доступу до ресурсу:

https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/ML-Agents-Overview.md#deep-reinforcement-learning

24. Proximal Policy Optimization [Електронний ресурс] // OpenAI – Режим доступу до ресурсу: https://web.archive.org/web/20230401063534/https://openai.com/research/openai-baselines-pp
25. Total Environment Research Themes - Assessing the potential of machine learning methods to study the removal of pharmaceuticals from wastewater using biochar or activated carbon [Електронний ресурс] // ScienceDirect – 2022. – Режим доступу до ресурсу: https://www.sciencedirect.com/science/article/pii/S2772809922000016?via%3Dihub
26. Best 3D modeling software and rendering software [Електронний ресурс] // Techradar – 2023 – Режим доступу до ресурсу: https://www.techradar.com/best/best-3d-modelling-software
27. Autodesk 3ds Max: Create massive worlds and high-quality designs [Електронний ресурс] // Autodesk – Режим доступу до ресурсу: https://www.autodesk.com/products/3ds-max/overview
28. Autodesk Maya: Create expansive worlds, complex characters, and dazzling effects [Електронний ресурс] // Autodesk – Режим доступу до ресурсу: https://www.autodesk.com/products/maya/overview

ДОДАТОК 1

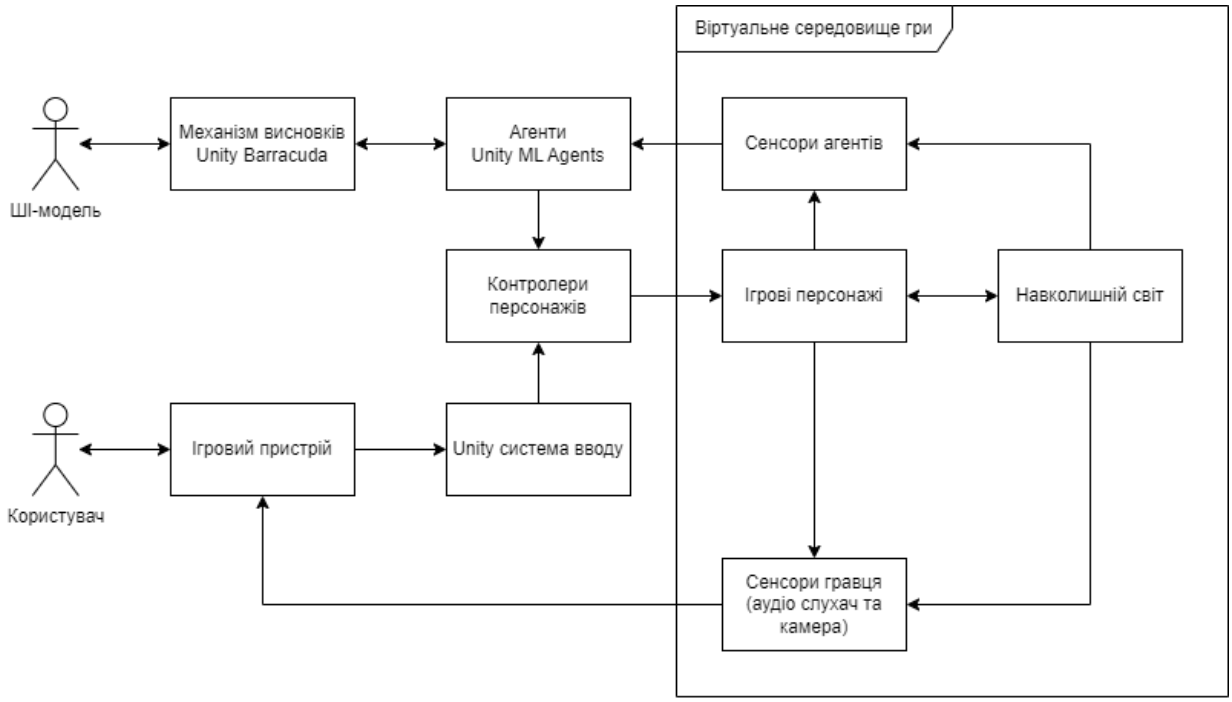
Комп’ютерна гра у жанрі шутер з використанням машинного навчання

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Грибенко Є. В.			**Комп'ютерна гра у жанрі шутер з використанням машинного навчання**	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Таран В.І.					
					Структурна схема системи	КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93	
Н. Контр.		Волокита А.М.					
Затвердив							

ДОДАТОК 2

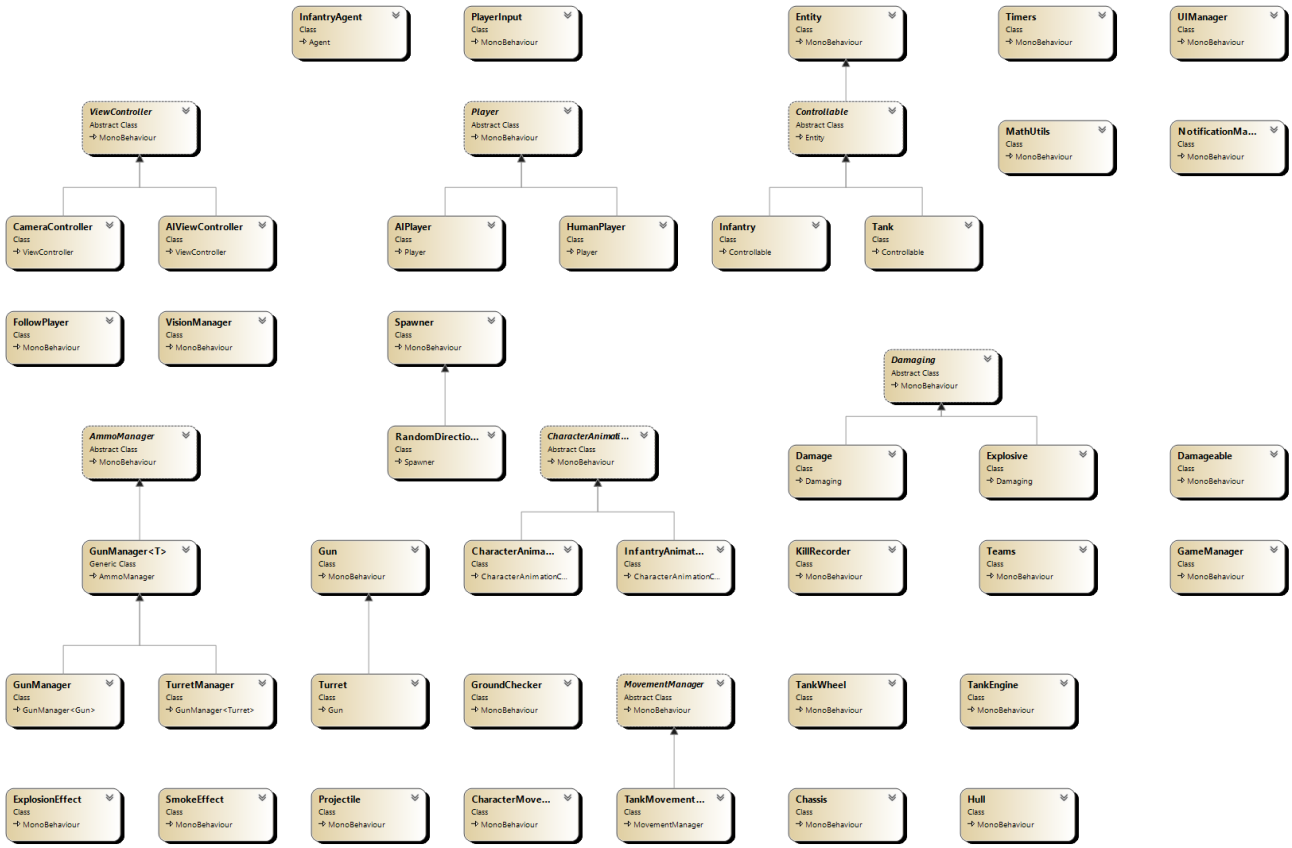
Комп’ютерна гра у жанрі шутер з використанням машинного навчання

Діаграма класів (функціональна схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Грибенко Є.В.			**Комп'ютерна гра у жанрі шутер з використанням машинного навчання** Діаграма класів (функціональна схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Таран В.І.					
Н. Контр.		Волокита А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93	
Затвердив							

ДОДАТОК 3

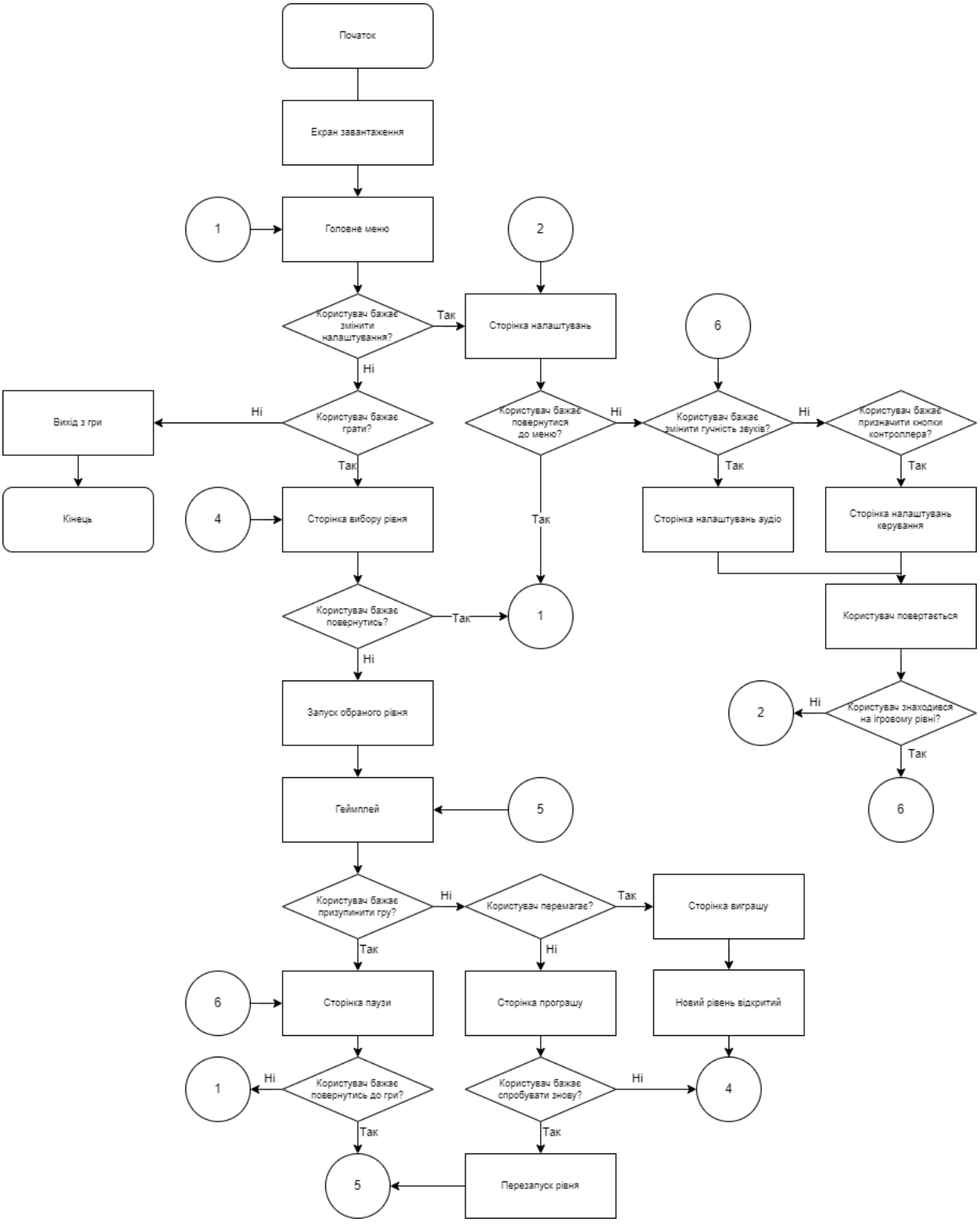
Комп’ютерна гра у жанрі шутер з використанням машинного навчання

Алгоритм дій програмного забезпечення (принципова схема)

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.006 Д3		
		№ докум.	Підпис	Дата			
Розробив		Грибенко Є.В.			**Комп'ютерна гра у жанрі шутер з використанням машинного навчання** Алгоритм дій програмного забезпечення (принципова схема)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Таран В.І.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93	
Н. Контр.		Волокита А.М.					
Затвердив							

ДОДАТОК 4

Комп’ютерна гра у жанрі шутер з використанням машинного навчання

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 43

Київ 2023 р

```
// Player.cs
using UnityEngine;
using static PlayerInput;

// Абстрактний клас гравця
public abstract class Player : MonoBehaviour
{
    [SerializeField] protected PlayerInput playerInput;
    [SerializeField] protected Spawner mySpawner;
    [SerializeField] protected ViewController viewController;

    protected Controllable character;

    public PlayerInput PlayerInput { get => playerInput; }

    // Призначення персонажа гравцю
    public virtual void SetCharacter(Controllable character)
    {
        this.character = character;
        character.SetPlayer(this);

        switch (character)
        {
            case Tank tank:
                playerInput.Tank.Buttons[TankActions.LeaveVehicle] = (button) => {
                    if (button) character = LeaveVehicle(tank);
                };
                break;
            case Infantry infantry:
                playerInput.Infantry.Buttons[InfantryActions.EnterVehicle] = (button)
=>
                {
                    if (
                        !button
                        || !(
                            viewController.RaycastItem(out GameObject obj)
                            && obj.TryGetComponent(out Hull hull)
                            && TryEnterVehicle(infantry, hull.Vehicle)
                        )
                    ) return;
                    character = hull.Vehicle;
                };
                break;
        }
    }

    // Вимкнення управління персонажа
    public virtual void RemoveCharacter(Controllable character)
    {
        character.RemovePlayer();
    }

    // Вхід персонажем у техніку
    public virtual bool TryEnterVehicle(Infantry character, Tank vehicle)
    {
        if (vehicle.Crew != null) return false;

        character.RemovePlayer();
        character.gameObject.SetActive(false);
```

```
        playerInput.Infantry.Reset();

        vehicle.Crew = character;
        SetCharacter(vehicle);
        return true;
    }

    // Покидання техніки персонажем
    public virtual Controllable LeaveVehicle(Tank tank)
    {
        foreach (var axis in new Axes[] { Axes.Horizontal, Axes.Vertical })
        {
            playerInput.Tank.Axes.ResetAxis(axis);
        }

        Infantry crew = tank.Crew;
        tank.Crew = null;

        crew.transform.position = tank.ExitPoint.position;
        crew.gameObject.SetActive(true);

        tank.RemovePlayer();

        playerInput.Tank.Reset();

        SetCharacter(crew);
        return crew;
    }
}

// PlayerInput.cs

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

// Клас, що відповідає за ввід користувача
public class PlayerInput : MonoBehaviour
{
    private readonly InfantryControls infantry = new();
    private readonly TankControls tank = new();
    private readonly GeneralControls general = new();

    public InfantryControls Infantry { get => infantry; }
    public TankControls Tank { get => tank; }
    public GeneralControls General { get => general; }

    // Ігрові осі контроллера
    public enum Axes
    {
        Horizontal,
        Vertical,
    }
    // Дії персонажа-піхотинця
    public enum InfantryActions
    {
        Shoot,
```

```
        ShootSingle,
        Jump,
        AimAbove,
        Crouch,
        IgnoreLookPoint,
        ThrowGrenade,
        EnterVehicle,
    }
    // Дії персонажа-танка
    public enum TankActions
    {
        Shoot,
        IgnoreLookPoint,
        LeaveVehicle,
        ShootAdditive,
    }
    // Загальні дії
    public enum GeneralActions
    {
        Zoom,
    }
    // Клас, що тримає callback-функції для дій
    public class KeyBindings<T>
    {
        public Dictionary<T, Func<bool, bool>> Actions { get; }

        public KeyBindings(Dictionary<T, Func<bool, bool>> actions)
        {
            Actions = actions;
        }
    }
    // Клас, що тримає callback-функції для осей
    public class AxisBindings<T>
    {
        public Dictionary<T, Func<float>> Actions { get; }

        public AxisBindings(Dictionary<T, Func<float>> actions)
        {
            Actions = actions;
        }
    }
    // Клас, що відповідає за кнопки або осі контролера
    public class CharacterActionGroup<T1, T2>
    {
        protected Dictionary<T1, Action<T2>> dict = new();
        public Type actionsType;

        static readonly Action<T2> doNothing = (_) => { };

        public CharacterActionGroup()
        {
            foreach (T1 action in Enum.GetValues(typeof(T1))) dict.Add(action,
doNothing);
            actionsType = typeof(T1);
        }

        public Action<T2> this[T1 index]
        {
            get => dict[index];
```

```
            set => dict[index] = value;
        }

        public void Reset()
        {
            foreach (var key in dict.Keys.ToList()) dict[key] = doNothing;
        }
    }

    public class Buttons<T> : CharacterActionGroup<T, bool> { }
    public class Axes<T> : CharacterActionGroup<T, float>
    {
        public void ResetAxis(T axis)
        {
            dict[axis].Invoke(0);
        }
    }
    // Клас, що відповідає за категорію дій персонажа
    public class Category<T1, T2>
    {
        public Axes<T1> Axes { get; }
        public Buttons<T2> Buttons { get; }
        public Category()
        {
            Axes = new();
            Buttons = new();
        }

        public void Reset()
        {
            Axes.Reset();
            Buttons.Reset();
        }
    }

    public class InfantryControls : Category<Axes, InfantryActions> { }
    public class TankControls : Category<Axes, TankActions> { }
    public class GeneralControls : Category<Axes, GeneralActions> { }

}

// HumanPlayer.cs
using System;
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
using static PlayerInput;

// Клас, що є відображенням гравця-людини у грі
public class HumanPlayer : Player
{
    [SerializeField] CameraController cameraController;
    [SerializeField] Volume volume;
    DepthOfField DoF;

    bool zoom = false;
    // Призначення клавіш для дій піхотинця
    readonly KeyBindings<InfantryActions> infantryActionBindings = new(
        new()
```

```
        {
            { InfantryActions.Shoot, MouseButtonChecker(0) },
            { InfantryActions.AimAbove, MouseButtonChecker(1) },
            { InfantryActions.IgnoreLookPoint, KeyChecker(KeyCode.LeftAlt) },
            { InfantryActions.ThrowGrenade, KeyChecker(KeyCode.G) },
            { InfantryActions.Jump, KeyChecker(KeyCode.Space) },
            { InfantryActions.Crouch, KeyChecker(KeyCode.C) },
            { InfantryActions.EnterVehicle, KeyChecker(KeyCode.F) },
        }
    );
    // Призначення клавіш для дій танка
    readonly KeyBindings<TankActions> tankActionBindings = new(
        new()
        {
            { TankActions.Shoot, MouseButtonChecker(0) },
            { TankActions.ShootAdditive, MouseButtonChecker(1) },
            { TankActions.IgnoreLookPoint, KeyChecker(KeyCode.C) },
            { TankActions.LeaveVehicle, KeyChecker(KeyCode.F) },
        }
    );
    // Призначення клавіш для загальних дій
    readonly KeyBindings<GeneralActions> generalBindings = new(
        new()
        {
            { GeneralActions.Zoom, MouseButtonChecker(2) },
        }
    );
    // Призначення осей
    readonly AxisBindings<Axes> axisBindings = new(
        new()
        {
            { Axes.Horizontal, () => Input.GetAxisRaw("Horizontal") },
            { Axes.Vertical, () => Input.GetAxisRaw("Vertical") },
        }
    );
    // Метод, що визивається на початку роботи програми
    private void Awake()
    {
        volume.profile.TryGet(out DoF);
        mySpawner.OnSpawn.AddListener((characterEntity) =>
SetCharacter(characterEntity.GetComponent<Controllable>()));
    }
    // Метод, що визивається на початку, другою
    private void Start()
    {
        playerInput.General.Buttons[GeneralActions.Zoom] = (button) => { if (button)
SetZoom(!zoom); };
    }

    // Зміна виду камери - FPS/TPS
    void SetZoom(bool on)
    {
        zoom = on;
        cameraController.SetZoom(zoom);
        character.FirstPerson = zoom;
    }
    // Призначення гравця
    public override void SetCharacter(Controllable character)
    {
```

```
        base.SetCharacter(character);

character.gameObject.GetComponentInChildren<Entity>().OnDeath.AddListener(OnDestroyPla
yer);

character.gameObject.GetComponentInChildren<GunManager>().MainGun.OnShot.AddListener(c
ameraController.ShootShake);

        character.SetMoveRelativeToTransform(cameraController.transform);

        cameraController.Follow = character;
        cameraController.DeactivateOnZoom = character.DeactivateOnZoom;
    }
    // Перевірка клавіш миші
    static Func<bool, bool> MouseButtonChecker(int button)
    {
        return (up) => up ? Input.GetMouseButtonUp(button) :
Input.GetMouseButtonDown(button);
    }
    // Перевірка клавіш клавіатури
    static Func<bool, bool> KeyChecker(KeyCode key)
    {
        return (up) => up ? Input.GetKeyUp(key) : Input.GetKeyDown(key);
    }
    // Виконання callback-функцій кнопок
    void InvokeActionEvents<T>(Category<Axes, T> category, KeyBindings<T> bindings)
    {
        var buttons = category.Buttons;
        foreach (var item in bindings.Actions)
        {
            var actionEvent = buttons[item.Key];
            if (item.Value(false)) actionEvent.Invoke(true);
            else if (item.Value(true)) actionEvent.Invoke(false);
        }
    }
    // Виконання callback-функцій осей
    void InvokeAxisEvents<T1, T2>(Category<T1, T2> category, AxisBindings<T1>
bindings)
    {
        var buttons = category.Axes;
        foreach (var item in bindings.Actions)
        {
            var actionEvent = buttons[item.Key];
            actionEvent.Invoke(item.Value());
        }
    }

    // Метод, що визивається кожен кадр гри
    void Update()
    {
        switch (character)
        {
            case Infantry:
                InvokeAxisEvents(playerInput.Infantry, axisBindings);
                InvokeActionEvents(playerInput.Infantry, infantryActionBindings);
                break;
            case Tank:
                InvokeAxisEvents(playerInput.Tank, axisBindings);
                InvokeActionEvents(playerInput.Tank, tankActionBindings);
```

```
                break;
        }

        InvokeActionEvents(playerInput.General, generalBindings);
    }
    // Метод, що визивається кожен кадр гри, другою
    void LateUpdate()
    {
        if (character == null) return;
        Vector3 focus = cameraController.RaycastCursor();

        character.LookAt(focus);

        DoF.focusDistance.value = (focus -
cameraController.transform.position).magnitude;

        Vector2 look = new Vector2(Input.GetAxis("Mouse X"), Input.GetAxis("Mouse
Y"));
        cameraController.RotateLook(look);

        Vector3 gunFocus = character.GetCursorPoint();
        cameraController.SetAdditiveCursor(gunFocus);
    }
    // Callback-функція після смерті персонажа
    private void OnDestroyPlayer(Entity attacker)
    {
        string attackerStr = attacker == character ? "yourself" :
attacker.SpecialName;
        NotificationManager.Instance.SendNotification($"You were killed by
{attackerStr}.", 5);
    }
    // Вхід в техніку
    public override bool TryEnterVehicle(Infantry character, Tank vehicle)
    {
        if (base.TryEnterVehicle(character, vehicle))
        {
            cameraController.Follow = vehicle;
            SetZoom(false);
            return true;
        }
        return false;
    }
    // Вихід з техніки
    public override Controllable LeaveVehicle(Tank tank)
    {
        Controllable character = base.LeaveVehicle(tank);
        cameraController.Follow = character;
        cameraController.EnableOccludingMeshes(true);
        SetZoom(false);
        return character;
    }
}

// AIPlayer.cs
using UnityEngine;
using static PlayerInput;

// Клас, що відповідає за ШІ-гравця
public class AIPlayer : Player
```

```
{
    AIViewController aiViewController;
    AmmoManager ammoManager;
    VisionManager visionManager;
    Rigidbody rb;

    public Controllable Character { get => character; }
    public AIViewController ViewController { get => aiViewController; }
    public Rigidbody Rb { get => rb; }
    public AmmoManager AmmoManager { get => ammoManager; }
    // Призначення гравця
    public override void SetCharacter(Controllable character)
    {
        base.SetCharacter(character);
        character.SetMoveRelativeToTransform(viewController.transform);
        character.FirstPerson = true;
        ammoManager = character.AmmoManager;
        rb = character.GetComponentInChildren<Rigidbody>();
    }

    void Awake()
    {
        aiViewController = GetComponentInChildren<AIViewController>();
        visionManager = GetComponentInChildren<VisionManager>();
    }

    private void Update()
    {
        if (character == null) return;
        viewController.transform.position = character.CameraPivot.position +
character.CameraPivot.rotation * character.CameraZoomOffset;
    }
    // Керування камерою
    public void RotateLook(Vector2 rotation)
    {
        aiViewController.RotateLook(rotation);
        character.LookAt(aiViewController.RaycastLook());
    }
    // Керування рухом
    public void Move(Vector2 movement)
    {
        playerInput.Infantry.Axes[Axes.Horizontal].Invoke(movement.x);
        playerInput.Infantry.Axes[Axes.Vertical].Invoke(movement.y);
    }
    // Стрибок
    public void Jump() =>
playerInput.Infantry.Buttons[InfantryActions.Jump].Invoke(true);
    // Стрільба
    public void Shoot() =>
playerInput.Infantry.Buttons[InfantryActions.ShootSingle].Invoke(true);
    // Кидання гранати
    public void ThrowGrenade() =>
playerInput.Infantry.Buttons[InfantryActions.ThrowGrenade].Invoke(true);
    // Перевірка точки на видимість
    public bool DoISeeEnemy(Vector3 enemy) => visionManager.DoISeeIt(enemy);
    // Підняття зброї
    public void AimAbove(bool on) => character.AimAbove(on);
    // Перевірка на те, чи знаходиться персонаж за укріпленням
    public bool AmIObstructed() => character.AmIObstructed();
```

```
}

// InfantryAgent.cs
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
using UnityEngine.Events;
using System.Linq;
using static Teams;
using System;

// Клас, що відповідає за інтелектуального агента
public class InfantryAgent : Agent
{
    [SerializeField] Transform spawnPoint;

    [SerializeField] AIPlayer character;

    [SerializeField] Vector3 characterCenterOffset = Vector3.up;
    [SerializeField] Vector3 characterVisionOffset = Vector3.up * 0.5f;

    [SerializeField] Spawner mySpawner;

    [SerializeField] Team oppositeTeam;

    readonly int entityTypesCount = Enum.GetNames(typeof(Entity.Type)).Length;

    public UnityEvent onBegin = new();

    Entity me;

    Rigidbody rb;

    Teams teams;

    const float lookSensitivity = 5;

    CustomBufferSensor enemiesBufferSensor;

    CustomBufferSensor teammatesBufferSensor;

    const float maxPerceptionDist = 200;

    bool isHeuristic = false;

    Vector2 heuristicLook = Vector2.zero;
    int heuristicFramesCounter = 0;

    float lastClosestDist = 0;

    Entity world;

    // Клас, що дозволяє працювати з сенсором кількох векторів
    class CustomBufferSensor
    {
        public int MaxObservations { get; }
        public int ObservationSize { get; }
```

```
        public CustomBufferSensor(int maxObservations, int observationSize)
        {
            MaxObservations = maxObservations;
            ObservationSize = observationSize;
        }

        public void FillEmpty (VectorSensor sensor, int filledObservationsCount)
        {
            sensor.AddObservation(new float[(MaxObservations -
filledObservationsCount) * ObservationSize]);
        }
    }
    // Метод, що визивається на початку
    public void Start()
    {
        world = GameManager.Instance.World;
        teams = Instance;

        enemiesBufferSensor = new(maxObservations: 5, observationSize: 3 +
entityTypesCount);
        teammatesBufferSensor = new(maxObservations: 5, observationSize: 3 +
entityTypesCount);

        teams.onWin.AddListener((teamThatWon) =>
        {
            if (teamThatWon != oppositeTeam) AddReward(0.6f);
            else if (teamThatWon == oppositeTeam) AddReward(-0.6f);

            EndEpisode();
        });
    }
    // Метод, що визивається кожен кадр гри
    private void Update()
    {
        if (!isHeuristic) return;
        heuristicLook += new Vector2(Input.GetAxisRaw("Mouse X"),
Input.GetAxisRaw("Mouse Y"));
        heuristicFramesCounter++;
    }
    // Метод ручного керування агентом
    public override void Heuristic(in ActionBuffers actionsOut)
    {
        if (!isHeuristic) isHeuristic = true;

        Vector2 look = heuristicLook / heuristicFramesCounter * lookSensitivity;

        heuristicLook = Vector2.zero;
        heuristicFramesCounter = 0;

        Vector2 movement = new Vector2(Input.GetAxisRaw("Horizontal"),
Input.GetAxisRaw("Vertical"));

        ActionSegment<float> continuousActions = actionsOut.ContinuousActions;
        continuousActions[0] = look.x;
        continuousActions[1] = look.y;

        ActionSegment<int> discreteAction = actionsOut.DiscreteActions;

        discreteAction[0] = (int)movement.x switch {
```

```
            > 0 => 1,
            < 0 => 2,
            0 => 0,
        };

        discreteAction[1] = (int)movement.y switch
        {
            > 0 => 1,
            < 0 => 2,
            0 => 0,
        };

        discreteAction[2] = Input.GetMouseButton(0) ? 1 : 0;
        discreteAction[3] = Input.GetKey(KeyCode.G) ? 1 : 0;
        discreteAction[4] = Input.GetMouseButton(1) ? 1 : 0;
    }

    float normalizeDistance(float val) => Mathf.Min(val, maxPerceptionDist) /
maxPerceptionDist;

    float normalizeAngle(float val) => val / 180;

    // Метод збору значень сенсорів
    public override void CollectObservations(VectorSensor sensor)
    {
        if (me.Dead)
        {
            sensor.AddObservation(new float[101]);
            return;
        }

        Transform myTransform = character.ViewController.transform;

        AmmoManager ammoManager = character.AmmoManager;

        bool hasAmmo = ammoManager.TotalAmmo > 0;

        sensor.AddObservation(myTransform.forward);
        sensor.AddObservation(rb.velocity);
        sensor.AddObservation(hasAmmo);
        sensor.AddObservation(ammoManager.IsReloading);
        sensor.AddObservation(ammoManager.Ammo / 30);
        sensor.AddObservation(ammoManager.GrenadeAmmo > 0);
        sensor.AddObservation(me.HP / me.MaxHP);

        Entity[] enemies = teams.FindClosestToMe(oppositeTeam, myTransform.position,
enemiesBufferSensor.MaxObservations);

        Entity closest = enemies.Any() ? enemies[0] : null;

        int enemyObjectsFilled = 0;

        foreach (Entity enemy in enemies)
        {
            Transform target = enemy.transform;
            Vector3 targetPos = target.position + characterCenterOffset;

            if (!character.DoISeeEnemy(targetPos + characterVisionOffset)) continue;
```

```
            Vector3 relativeDelta = myTransform.worldToLocalMatrix * (targetPos -
myTransform.position);
            Vector3 relativeDir = relativeDelta.normalized;
            float dist = relativeDelta.magnitude;

            Vector3 angleDiff = Quaternion.FromToRotation(Vector3.forward,
relativeDir).eulerAngles;

            float angleX = normalizeAngle(Mathf.DeltaAngle(0, angleDiff.y));
            float angleY = normalizeAngle(Mathf.DeltaAngle(0, -angleDiff.x));

            float[] observation = { angleX, angleY, normalizeDistance(dist) };

            sensor.AddObservation(observation);
            sensor.AddOneHotObservation((int)enemy.EntityType, entityTypesCount);

            enemyObjectsFilled++;

            float angleError = Vector3.Angle(relativeDir, Vector3.forward);

            if (enemy == closest)
            {
                lastClosestDist = dist;
            }

            if (hasAmmo && enemy.EntityType != Entity.Type.grenade)
            {
                if (angleError < 0.3f) AddReward(0.005f);
                else if (angleError < 0.6f) AddReward(0.002f);
                else if (angleError < 1.2f) AddReward(0.001f);
                else if (angleError < 2f) AddReward(0.0005f);
                else if (enemy == closest)
                {

                    float angleReward = 0.001f - Mathf.Min(0.002f, angleError * 0.002f
/ 180);
                    AddReward(angleReward);

                }
            }
        }
        enemiesBufferSensor.FillEmpty(sensor, enemyObjectsFilled);

        Entity[] teammates = teams.FindClosestToMe(oppositeTeam, myTransform.position,
teammatesBufferSensor.MaxObservations);

        int teammatesObjectsFilled = 0;

        foreach (Entity teammate in teammates)
        {
            Transform target = teammate.transform;
            Vector3 targetPos = target.position + characterCenterOffset;
            Vector3 relativeDelta = myTransform.worldToLocalMatrix * (targetPos -
myTransform.position);
            Vector3 relativeDir = relativeDelta.normalized;
            float dist = relativeDelta.magnitude;

            Vector3 angleDiff = Quaternion.FromToRotation(Vector3.forward,
relativeDir).eulerAngles;
```

```
            float angleX = normalizeAngle(Mathf.DeltaAngle(0, angleDiff.y));
            float angleY = normalizeAngle(Mathf.DeltaAngle(0, -angleDiff.x));

            float[] observation = { angleX, angleY, normalizeDistance(dist) };

            sensor.AddObservation(observation);
            sensor.AddOneHotObservation((int)teammate.EntityType, entityTypesCount);

            teammatesObjectsFilled++;
        }

        teammatesBufferSensor.FillEmpty(sensor, teammatesObjectsFilled);

        float elevation = myTransform.rotation.eulerAngles.x;
        if (Mathf.Abs(Mathf.DeltaAngle(0, elevation)) > 60) AddReward(-0.001f);
    }
    // Метод переводу виводу агента у вісь
    public int DiscreteToDir(int val) => val switch
    {
        1 => 1,
        2 => -1,
        _ => 0,
    };

    // Метод, що дозволяє виконувати дії агента
    public override void OnActionReceived(ActionBuffers actions)
    {
        if (character.Character == null || me.Dead) return;
        if (float.IsNaN(actions.ContinuousActions[0]) ||
float.IsNaN(actions.ContinuousActions[1])) return;

        Vector2 lookVector = new(actions.ContinuousActions[0],
actions.ContinuousActions[1]);

        character.RotateLook(lookVector);

        Vector2 movement = new(
            DiscreteToDir(actions.DiscreteActions[0]),
            DiscreteToDir(actions.DiscreteActions[1])
        );

        character.Move(movement);

        if (actions.DiscreteActions[2] == 1)
        {
            character.Shoot();
            AddReward(-0.0001f);
        }
        if (actions.DiscreteActions[3] == 1)
        {
            character.ThrowGrenade();
            AddReward(-0.01f);
            if (character.AmmoManager.GrenadeAmmo <= 0) AddReward(-0.01f);
            if(lastClosestDist > 25) AddReward(-0.01f);
        }
    }

    private void FixedUpdate()
```

```
{
    if (character.Character == null) return;

    AddReward(-0.00005f); // time reward
}

// Метод, що спрацьовує на початку епізоду навчання агента
public override void OnEpisodeBegin()
{
    RespawnCharacter();
    onBegin.Invoke();
}

// Скидання стану персонажа - його знищення і створення
public void RespawnCharacter()
{
    if (character.Character != null )
    if (character.Character != null) Destroy(character.Character.gameObject);

    if (me != null) me.Kill(world);

    me = mySpawner.Spawn();
    me.OnDeath.AddListener(OnMeDied);
    me.OnHeal.AddListener(OnHeal);

    KillRecorder killRecorder = me.gameObject.AddComponent<KillRecorder>();

    killRecorder.OnKill.AddListener(OnSomeoneKilled);

    killRecorder.OnDamage.AddListener(OnEnemyDamaged);

    character.SetCharacter(me.GetComponent<Controllable>());

    character.Character.transform.position = spawnPoint.position;
    character.ViewController.ResetTransform();

    rb = character.Rb;
}

public void OnMeDied(Entity attacker)
{
    if (attacker == world) return;

    // Death penalty
    AddReward(-0.2f);
}

public void OnSomeoneKilled(Entity victim)
{
    if (victim == me || victim.team != oppositeTeam) return;

    // Kill reward
    AddReward(0.4f);
}

public void OnEnemyDamaged(Entity victim)
{
    // Damage reward
    AddReward(0.4f);
```

```
        // Distance reward
        AddReward(0.4f * Mathf.Clamp01((victim.transform.position -
me.transform.position).magnitude/40f));
    }

    public void OnHeal()
    {
        // Healing and reloading on base reward
        AddReward(0.4f);
    }

    public void OnWin()
    {
        EndEpisode();
    }
}

// Entity.cs
using UnityEngine;
using UnityEngine.Events;

// Клас, що відповідає за ігрові сутності та їх здоров'я
public class Entity : MonoBehaviour
{
    [SerializeField] int hp = 4;
    [SerializeField] int maxhp = 4;
    [SerializeField] string specialName;
    [SerializeField] Type entityType = Type.target;

    public UnityEvent<Entity> OnDeath = new();
    public UnityEvent<Entity> OnDamage = new();
    public UnityEvent OnHeal = new();

    KillRecorder killRecorder = null;

    public Teams.Team team = Teams.Team.Enemies;

    public bool dieable = true;

    public bool Dead { get => dead; }
    bool dead = false;

    public int HP { get => hp; }
    public int MaxHP { get => maxhp; }
    public string SpecialName { get => specialName; }
    public KillRecorder KillRecorder { get => killRecorder; set => killRecorder =
value; }
    public Type EntityType { get => entityType; set => entityType = value; }
    // Тип сутності
    public enum Type
    {
        infantry,
        tank,
        lightVehicle,
        target,
        teamBase,
        grenade,
    }
```

```
    // Метод, що викликається на початку
    void Start()
    {
        if (string.IsNullOrWhiteSpace(specialName)) specialName = transform.name;
        Teams.Instance.AddObject(this);
    }
    // Метод, що викликається при знищенні сутності
    void OnDestroy()
    {
        Teams.Instance.RemoveObject(this);
    }
    // Метод, що викликається при нанесенні шкоди сутності, з врахуванням команди
    public void Damage(int damageAmount, Teams.Team team, Entity attacker)
    {
        if (this.team == team) return;
        Damage(damageAmount, attacker);
    }
    // Метод, що викликається при нанесенні шкоди сутності без врахуванням команди
    public void Damage(int damageAmount, Entity attacker)
    {
        hp -= damageAmount;
        OnDamage.Invoke(attacker);
        if (hp <= 0 && !dead) {
            if (dieable) Die(attacker);
            return;
        }
        if (attacker.killRecorder != null)
attacker.killRecorder.OnDamage.Invoke(this);
    }
    // Метод, що вбиває сутність
    public void Kill(Entity attacker)
    {
        Damage(HP, attacker);
    }
    // Метод, через який помирає сутність
    void Die(Entity attacker)
    {
        dead = true;

        string message = attacker != this ? $"{attacker.SpecialName} killed
{specialName}" : $"{specialName} commited suicide.";
        message = message[0].ToString().ToUpper() + message[1..];

        if (attacker != this) Debug.Log(message);

        Destroy(gameObject);

        OnDeath.Invoke(attacker);
        if (attacker.killRecorder != null) attacker.killRecorder.OnKill.Invoke(this);
    }
    // Метод що лікує сутність
    public void Heal()
    {
        if (hp == maxhp) return;
        hp = maxhp;
        OnHeal.Invoke();
    }
}
```

```
// Controllable.cs
using System.Collections.Generic;
using UnityEngine;
using static Gun;

// Клас контрольованих сутностей
public abstract class Controllable : Entity
{
    [SerializeField] Vector3 cameraOffset;
    [SerializeField] Vector3 cameraZoomOffset;
    [SerializeField] Transform cameraPivot;
    [SerializeField] List<Component> deactivateOnZoom = new();
    [SerializeField] protected AmmoManager ammoManager;

    protected Player player;
    protected PlayerInput playerInput;

    protected Transform moveRelativeToTransform;
    protected bool zoomed = false;

    public bool FirstPerson { set => zoomed = value; get => zoomed; }
    public Transform CameraPivot { get => cameraPivot; }
    public Vector3 CameraOffset { get => cameraOffset; }
    public Vector3 CameraZoomOffset { get => cameraZoomOffset; }
    public List<Component> DeactivateOnZoom { get => deactivateOnZoom; }
    public AmmoManager AmmoManager { get => ammoManager; }

    /// <summary>
    /// Global direction vector
    /// </summary>
    public abstract void LookAt(Vector3 Direction);

    public abstract Vector3 GetCursorPoint();

    public virtual void RelativeLook(Vector2 angles) { }

    public virtual void SetMoveRelativeToTransform(Transform cam) =>
moveRelativeToTransform = cam;
    // Встановлення гравця для сутності
    public virtual void SetPlayer(Player player)
    {
        this.player = player;
        playerInput = player.PlayerInput;
    }
    // Відв’язування гравця від сутності
    public void RemovePlayer()
    {
        player = null;
        playerInput = null;
    }

    public virtual bool AimAbove(bool on) => false;

    public virtual void SetReloadAnim(GunType type, bool on) { }

    public virtual bool AmIObstructed() => false;

    public virtual void Awake()
    {
```

```
        ammoManager = GetComponent<AmmoManager>();
    }
}

// Infantry.cs
using UnityEngine;
using static Gun;
using static PlayerInput;

// Клас піхотинця
public class Infantry : Controllable
{
    [SerializeField] CharacterAnimationController animatorController;
    [SerializeField] CharacterMovementController characterController;

    [SerializeField] Transform character;

    [SerializeField] LayerMask raycastMask;
    [SerializeField] float aimCoef = 10f;

    [SerializeField] Transform targetMarker;
    [SerializeField] Transform gunPivot;

    [SerializeField] GunManager gunManager;

    float spreadInMovementMultiplier = 4f;

    GroundChecker groundChecker;

    Vector3 characterCenterOffset = Vector3.up * 0.5f;

    public Vector3 TargetRotation { get; private set; }

    Timers.CooldownTimer aimAboveCooldown = new(1f);

    bool isAimingAbove = false;

    Rigidbody rb;

    public override void Awake()
    {
        base.Awake();
        rb = GetComponent<Rigidbody>();
        groundChecker = characterController.GroundChecker;
    }
    // Метод, що викликається на початку
    void Start()
    {
        if (moveRelativeToTransform is not null) characterController.MoveRelativeTo =
moveRelativeToTransform;
    }
    // Метод, що викликається кожен кадр фізики
    private void FixedUpdate()
    {
        if (!groundChecker.IsOnGround && gunManager != null)
gunManager.Shoot(GunType.main, ShootingMode.turnSprayOff);
    }
    // Метод, що викликається кожен кадр
    private void Update()
```

```
    {
        TargetRotation = Vector3.ProjectOnPlane(targetMarker.position -
transform.position, Vector3.up).normalized;
        gunManager.SetMovementAccuracyCoef(Mathf.Clamp01(rb.velocity.magnitude /
characterController.MaxSpeed) * spreadInMovementMultiplier);
    }
    // Керування камерою
    public override void LookAt(Vector3 point)
    {
        Vector3 aimDelta = point - gunPivot.position;

        targetMarker.position = gunPivot.position +
            Quaternion.Lerp(
                Quaternion.LookRotation(targetMarker.position - gunPivot.position),
                Quaternion.LookRotation(Vector3.Angle(aimDelta, character.forward) <
130 ? aimDelta : character.forward),
                aimCoef * Time.deltaTime
            ) * Vector3.forward;
    }
    // Отримання курсору зброї
    public override Vector3 GetCursorPoint()
    {
        return gunManager is not null ? gunManager.RayCastGun() : Vector3.zero;
    }
    // Підняття зброї
    public override bool AimAbove(bool on) {
        if (!aimAboveCooldown.Check()) return false;
        aimAboveCooldown.Reset();
        animatorController.SetAimingAbove(on);
        gunManager.SetSprayAccuracyCoef(on ? 1.7f : 1);
        isAimingAbove = on;
        return true;
    }
    // Відносне керування камерою
    public override void RelativeLook(Vector2 angles)
    {
        targetMarker.position = gunPivot.position + Quaternion.Euler(angles.y,
angles.x, 0) * Vector3.forward;
    }

    // Встановлення відносного простору руху
    public override void SetMoveRelativeToTransform(Transform cam)
    {
        characterController.MoveRelativeTo = cam;
        base.SetMoveRelativeToTransform(cam);
    }

    bool obstructed = false;
    Vector3 contact = Vector3.zero;
    // перевірка на укриття
    public override bool AmIObstructed()
    {
        return Physics.Raycast(transform.position + characterCenterOffset,
TargetRotation, 6, raycastMask);
    }
    // Візуалізація попередньої перевірки
    private void OnDrawGizmosSelected()
    {
        if (!obstructed)
```

```
        {
            Gizmos.color = Color.yellow;
            Vector3 me = transform.position + characterCenterOffset;
            Gizmos.DrawLine(me, me + TargetRotation * 2);
        } else
        {
            Gizmos.color = Color.yellow;
            Vector3 me = transform.position + characterCenterOffset;
            Gizmos.DrawLine(me, contact);
        }

    }
    // Встановлення гравця
    public override void SetPlayer(Player player)
    {
        base.SetPlayer(player);

        PlayerInput playerInput = player.PlayerInput;
        var actions = playerInput.Infantry.Buttons;
        var axes = playerInput.Infantry.Axes;

        actions[InfantryActions.Jump] = (button) => { if (button)
characterController.Jump(); };
        actions[InfantryActions.Shoot] = (button) => {
            gunManager.Shoot(GunType.main, button ? ShootingMode.turnSprayOn :
ShootingMode.turnSprayOff);
        };
        actions[InfantryActions.ShootSingle] = (button) => {
            if (button) gunManager.Shoot(GunType.main, ShootingMode.single);
        };

        actions[InfantryActions.ThrowGrenade] = (button) => {
            if (button) gunManager.Shoot(GunType.grenade, ShootingMode.single);
        };

        actions[InfantryActions.AimAbove] = (button) => { if (button)
AimAbove(!isAimingAbove); };

        actions[InfantryActions.IgnoreLookPoint] = (on) =>
animatorController.SetAiming(!on);

        axes[Axes.Horizontal] = (val) => { characterController.SetMovementAxis(0,
val); };
        axes[Axes.Vertical] = (val) => { characterController.SetMovementAxis(1, val);
};
    }

    public override void SetReloadAnim(GunType type, bool on) =>
animatorController.SetReload(on);
}

// CharacterAnimationController.cs

using UnityEngine;

// Клас, що відповідає за анімацію персонажа
public abstract class CharacterAnimationController: MonoBehaviour
{
    public abstract void SetMovement(Vector2 movement);
```

```
    public abstract void SetJump(bool on);
    public abstract void SetAiming(bool on);
    public abstract void SetAimingAbove(bool on);
    public abstract void SetReload(bool on);
}

// InfantryAnimatorController.cs

using UnityEngine;
using UnityEngine.Animations.Rigging;
// Клас, що відповідає за анімацію піхотинця
public class InfantryAnimatorController : CharacterAnimationController
{
    [SerializeField] Animator animator;
    [SerializeField] Rig aimRig;
    [SerializeField] Rigidbody rb;
    [SerializeField] float runSpeed = 3.48f;
    [SerializeField] TwoBoneIKConstraint leftHand;
    [SerializeField] Gun gun;
    [SerializeField] Controllable character;

    float transitionSmoothing = 6;

    float aimingLayerWeightTarget = 0f;
    float aimingLayerWeight = 0f;

    float aimingAboveWeightTarget = 0f;
    float aimingAboveWeight = 0f;

    float aimRigWeightTarget = 1f;
    float aimRigWeight = 1f;

    int Speed;
    int isAiming;
    int aimingLayer;
    int isJumping;
    int isReloading;
    int isAimingAbove;

    int SpeedX;
    int SpeedY;

    bool IsIgnoringLookPoint { get => aimRigWeightTarget < 0.5f; }

    bool ignoreBeforeJump = false;

    public enum State
    {
        idle,
        walking,
        running,
    }

    private void Awake()
    {
        Speed = Animator.StringToHash("Speed");
        SpeedX = Animator.StringToHash("SpeedX");
        SpeedY = Animator.StringToHash("SpeedY");
```

```
        isAiming = Animator.StringToHash("isAiming");
        aimingLayer = animator.GetLayerIndex("AimingLayer");
        isJumping = Animator.StringToHash("isJumping");
        isReloading = Animator.StringToHash("isReloading");
        isAimingAbove = Animator.StringToHash("AimingAbove");
        ignoreBeforeJump = IsIgnoringLookPoint;
        aimingLayerWeightTarget = 1;
    }

    private void Update()
    {
        SetSpeed(rb.velocity.magnitude / runSpeed);

        aimingLayerWeight = Mathf.Lerp(aimingLayerWeight, aimingLayerWeightTarget,
transitionSmoothing * Time.deltaTime);
        aimingAboveWeight = Mathf.Lerp(aimingAboveWeight, aimingAboveWeightTarget,
transitionSmoothing * Time.deltaTime);
        aimRigWeight = Mathf.Lerp(aimRigWeight, aimRigWeightTarget,
transitionSmoothing * Time.deltaTime);

        animator.SetLayerWeight(aimingLayer, aimingLayerWeight);

        animator.SetFloat(isAimingAbove, aimingAboveWeight);
        aimRig.weight = aimRigWeight;

        leftHand.weight = 1 - aimingAboveWeight;
    }

    void SetSpeed(float speed)
    {
        animator.SetFloat(Speed, speed);
    }

   public override void SetMovement(Vector2 movement)
    {
        animator.SetFloat(SpeedX, movement.x);
        animator.SetFloat(SpeedY, movement.y);
    }

    public override void SetJump(bool on)
    {
        if (on)
        {
            if (animator.GetBool(isJumping)) return;
            ignoreBeforeJump = IsIgnoringLookPoint;
        }

        animator.SetBool(isJumping, on);

        if (animator.GetBool(isReloading)) return;

        float aimingWeights = on ? 0 : (ignoreBeforeJump ? 0 : 1);


        aimRigWeightTarget = aimingWeights;
        aimingLayerWeightTarget = aimingWeights;
    }

    public override void SetAiming(bool on)
```

```
    {
        ignoreBeforeJump = !on;
        animator.SetBool(isAiming, on);
        if (animator.GetBool(isJumping)) return;

        float aimingWeights = on ? 1 : 0;

        aimRigWeightTarget = aimingWeights;
    }

    public override void SetAimingAbove(bool on)
    {
        aimingAboveWeightTarget = on ? 1 : 0;
    }

    public override void SetReload(bool on)
    {
        if (animator == null) return;

        ignoreBeforeJump = on;
        animator.SetBool(isReloading, on);
        character.RelativeLook(Vector2.zero);
        SetAiming(!on);

        float aimingWeights = on ? 1 : 0;

        aimingLayerWeightTarget = aimingWeights;
    }
}



// CharacterMovementController.cs

using UnityEngine;
// Клас, що відповідає за рух персонажа
public class CharacterMovementController : MonoBehaviour
{
    [Header("Settings")]
    [SerializeField] float movementSmoothing = 0.05f;

    [SerializeField] float moveForce = 20f;
    [SerializeField] float moveForceWhileJumpingMultiplier = 0.5f;
    [SerializeField] float maxSpeed = 13f;
    [SerializeField] float flyMaxSpeed = 20f;
    [SerializeField] float highAccelerationPoint = 3f;
    [SerializeField] float jumpPower = 8f;
    [Tooltip("Velocity threshold that affects movement control")]
    [SerializeField] float minVelocity = 0.1f;
    [SerializeField] float playerMaterialFriction = 4f;
    [SerializeField] CharacterAnimationController animatorController;

    [SerializeField] CapsuleCollider capsuleCollider;

    [SerializeField] GroundChecker groundChecker;

    [SerializeField] Infantry character;

    [SerializeField] Transform characterTransform;
```

```
    Transform relativeMovementSpace;
    Rigidbody rb;

    Vector2 movement = Vector2.zero;

    Timers.CooldownTimer jumpCoolDownTimer;

    public Transform MoveRelativeTo { set => relativeMovementSpace = value; }
    public float MaxSpeed { get => maxSpeed; }
    public GroundChecker GroundChecker { get => groundChecker; }
    public Vector2 Movement { get => movement; set => movement = value; }
    // Метод, що визивається на початку
    void Awake()
    {
        rb = GetComponent<Rigidbody>();
        jumpCoolDownTimer = new Timers.CooldownTimer(0.5f);
    }

    bool wasOnGround = true;
    // Метод, що визивається кожен кадр
    private void Update()
    {
        if (character.FirstPerson)
        {
            Vector3 targetRotation =
Vector3.ProjectOnPlane(relativeMovementSpace.forward, Vector3.up).normalized;
            characterTransform.rotation = Quaternion.Lerp(characterTransform.rotation,
Quaternion.LookRotation(targetRotation), movementSmoothing);
        }
        else
        {
            Vector3 velocity2D = Vector3.ProjectOnPlane(rb.velocity, Vector3.up);
            characterTransform.rotation =
                velocity2D.magnitude > 0.3f
                    ? Quaternion.Lerp(characterTransform.rotation,
Quaternion.LookRotation(velocity2D.normalized), movementSmoothing)
                    : characterTransform.rotation;
        }
    }
    // Метод, що визивається кожен кадр фізики
    private void FixedUpdate()
    {
        if (relativeMovementSpace is null) return;

        bool isOnGround = groundChecker.IsOnGround;

        if (isOnGround != wasOnGround) animatorController.SetJump(!isOnGround);

        UpdatePlayerFriction(isOnGround);

        Vector3 horizontalVelocity = Vector3.ProjectOnPlane(rb.velocity, Vector3.up);

        Quaternion movementRotation =
Quaternion.LookRotation(Vector3.ProjectOnPlane(relativeMovementSpace.forward,
Vector3.up).normalized);

        Vector3 movementDirection = movementRotation *
MathUtils.Vector2To3DSpace(movement);
```

```
        Vector3 velocityDirection = horizontalVelocity.magnitude > minVelocity ?
horizontalVelocity : transform.forward;
        Vector3 orthogonalControls = Vector3.Project(movementDirection,
Vector3.Cross(velocityDirection, Vector3.up));
        Vector3 tangentialControls = Vector3.Project(movementDirection,
velocityDirection);

        float velocityAlongControls = Vector3.Dot(rb.velocity,
tangentialControls.normalized);

        float lowAccelerationPoint = isOnGround ? maxSpeed : flyMaxSpeed;
        float acceleration = MathUtils.SLikeInterpolation(x: velocityAlongControls, a:
lowAccelerationPoint - highAccelerationPoint, b: lowAccelerationPoint);

        float accelerationMultiplier = isOnGround ? moveForce : moveForce *
moveForceWhileJumpingMultiplier;

        rb.AddForce(accelerationMultiplier * Time.fixedDeltaTime * (orthogonalControls
+ tangentialControls * acceleration), ForceMode.VelocityChange);

        Vector3 relativeVelocity = characterTransform.worldToLocalMatrix *
rb.velocity;
        Vector2 relativeVelocity2D = MathUtils.Vector3Flatten(relativeVelocity);

        animatorController.SetMovement(character.FirstPerson ? relativeVelocity2D :
Vector2.up * relativeVelocity2D.magnitude);

        wasOnGround = isOnGround;
    }

    void UpdatePlayerFriction(bool isOnGround) =>
capsuleCollider.material.dynamicFriction = isOnGround ? playerMaterialFriction : 0;

    public void SetMovementAxis(int axis, float value)
    {
        movement[axis] = value;
    }

    public void Jump()
    {
        if (!groundChecker.IsOnGround || !jumpCoolDownTimer.Check()) return;
        jumpCoolDownTimer.Reset();
        rb.AddForce(Vector3.up * jumpPower, ForceMode.VelocityChange);
    }
}


// Tank.cs
using UnityEngine;
using static Gun;
using static PlayerInput;
// Клас танка
public class Tank : Controllable
{
    TankEngine engine;
    Infantry crew;
    [SerializeField] Transform exitPoint;
    TurretManager turretManager;
```

```
    public Infantry Crew { get => crew; set => crew = value; }
    public Transform ExitPoint { get => exitPoint; }
    // Метод, що визивається на початку
    void Start()
    {
        engine = GetComponentInChildren<TankEngine>();
        turretManager = GetComponentInChildren<TurretManager>();
        OnDeath.AddListener(_) => { if (playerInput != null)
playerInput.Tank.Reset(); });
    }
    // Встановлення гравця
    public override void SetPlayer(Player player)
    {
        base.SetPlayer(player);

        PlayerInput playerInput = player.PlayerInput;
        var actions = playerInput.Tank.Buttons;
        var axes = playerInput.Tank.Axes;

        axes[Axes.Horizontal] = (val) => { engine.SetMovementAxis(0, val); };
        axes[Axes.Vertical] = (val) => { engine.SetMovementAxis(1, val); };

        actions[TankActions.Shoot] = (button) => {
            turretManager.Shoot(GunType.main, button ? ShootingMode.turnSprayOn :
ShootingMode.turnSprayOff);
        };

        actions[TankActions.ShootAdditive] = (button) => {
            turretManager.Shoot(GunType.additive, button ? ShootingMode.turnSprayOn :
ShootingMode.turnSprayOff);
        };

        actions[TankActions.IgnoreLookPoint] = (button) => {
turretManager.IgnoreLookPoint(button); };
    }

    public override void LookAt(Vector3 Direction) => turretManager.LookAt(Direction);

    public override Vector3 GetCursorPoint() => turretManager.RayCastGun();

    public override void RelativeLook(Vector2 angles) => turretManager.Rotate(angles,
GunType.main);

    public void SetBrakes(bool on) => engine.Brake = on;
}

// TankEngine.cs
using UnityEngine;
// Клас двигуна танка
public class TankEngine : MonoBehaviour
{
    [SerializeField] float motorForce;
    [SerializeField] float zeroForceSpeed = 10;
    [SerializeField] float maxForceSpeed = 8;
    [SerializeField] float brakeTorque = 300;

    [SerializeField] float brakeSpeedThreshold = 1;
```

```
    [SerializeField] Chassis chassis;
    TankWheel[] wheels;

    bool brake;

    public bool Brake { set => brake = value; }

    Vector2 controls;
    // Метод, що визивається на початку
    void Start()
    {
        wheels = chassis.Wheels;
    }
    // Метод, що визивається кожен кадр фізики
    void FixedUpdate()
    {
        float leftControls = Mathf.Clamp(controls.y + controls.x, -1, 1);
        float rightControls = Mathf.Clamp(controls.y - controls.x, -1, 1);

        Vector2 trackSpeed = chassis.TrackVelocity / Time.fixedDeltaTime;

        foreach (TankWheel wheel in wheels)
        {
            float speed = wheel.type == TankWheel.WheelType.Left ? trackSpeed.x :
trackSpeed.y;
            float wheelControls = brake ? 0 : (wheel.type == TankWheel.WheelType.Left
? leftControls : rightControls);

            bool breaking = wheelControls == 0 || speed * Mathf.Sign(wheelControls) <=
-brakeSpeedThreshold;
            wheel.Wheel.brakeTorque = (breaking || brake) ? brakeTorque : 0;

            wheel.Wheel.motorTorque = GetMotorForce(speed * Mathf.Sign(wheelControls),
wheelControls);
        }
    }
    // Отримання сили мотора
    float GetMotorForce(float speed, float controls)
    {
        float SFunctionRes = speed < maxForceSpeed ? 1
            : speed > zeroForceSpeed ? 0
            : .5f + .5f * Mathf.Cos((speed - maxForceSpeed) / (zeroForceSpeed -
maxForceSpeed) * Mathf.PI);

        return SFunctionRes * motorForce * controls;
    }

    public void SetMovementAxis(int axis, float value)
    {
        controls[axis] = value;
    }
}

// Chassis.cs
using System.Collections.Generic;
using UnityEngine;
// Клас шасі танка
public class Chassis : MonoBehaviour
{
```

```
    [Tooltip("A substring to determine if a wheel is left(should be included in wheel
transform's name)")]
    [SerializeField] string wheelLeftSubstring = "_L";

    [SerializeField] GameObject wheelPrefab;

    [Tooltip("Chassis object that should contain all wheel meshes")]
    [SerializeField] Transform wheelMeshContainer;

    [Tooltip("Hull object transform (needed to fix physical wheels rotation if hull is
rotated).")]
    [SerializeField] Transform rootObject;

    [Tooltip("Track virtual position to track movement")]
    [SerializeField] Transform leftTrack;
    [SerializeField] Transform rightTrack;

    [Tooltip("A place to store physic wheels")]
    [SerializeField] Transform wheelHandler;

    [SerializeField] Vector3 wheelCenter;

    TankWheel[] wheels;

    Vector2 trackVelocity;

    public TankWheel[] Wheels { get => wheels; }
    public Vector2 TrackVelocity { get => trackVelocity; }
    // Метод, що визивається на початку
    void Awake()
    {
        SetupWheels();
    }
    // Призначення колес
    void SetupWheels()
    {
        foreach (Transform child in wheelMeshContainer)
        {
            Transform wheelObj = Instantiate(wheelPrefab, wheelHandler,
false).transform;

            wheelObj.position = child.position;

            TankWheel.WheelType type = child.name.Contains(wheelLeftSubstring) ?
                TankWheel.WheelType.Left :
                TankWheel.WheelType.Right;

            TankWheel wheel = wheelObj.gameObject.AddComponent<TankWheel>();
            wheel.type = type;
            wheel.Mesh = child;

            wheelObj.GetComponent<WheelCollider>().center = wheelCenter;
        }

        wheels = GetComponentsInChildren<TankWheel>();
    }
    // Встановлення швидкості повертання колеса
    public void SetWheelSpeed(Vector2 leftRightSpeeds)
    {
```

```
        float l = leftRightSpeeds.x;
        float r = leftRightSpeeds.y;

        foreach (TankWheel wheel in wheels)
        {
            wheel.Speed = wheel.type == TankWheel.WheelType.Left ? l : r;
        }
    }

    Vector3 lppos; // left previous frame position
    Vector3 rppos; // right previous frame position
    // Оновлення швидкості повороту гусениці
    void UpdateTrackSpeed()
    {
        Vector3 lpos = leftTrack.position;
        float ldelta = Vector3.Dot(lpos - lppos, transform.forward);

        Vector3 rpos = rightTrack.position;
        float rdelta = Vector3.Dot(rpos - rppos, transform.forward);

        lppos = lpos;
        rppos = rpos;

        trackVelocity = new(ldelta, rdelta);
    }
    // Метод, що визивається кожен кадр фізики
    public void FixedUpdate()
    {
        UpdateTrackSpeed();
        SetWheelSpeed(trackVelocity / Time.fixedDeltaTime);
    }

    void OnDrawGizmosSelected()
    {
        if (!Application.isPlaying) return;
        Gizmos.color = Color.yellow;
        foreach (TankWheel wheel in wheels)
        {
            float spd = wheel.Speed;
            Vector3 offset = wheel.type == TankWheel.WheelType.Right ? Vector3.right :
Vector3.left;
            Vector3 pos = wheel.transform.position;
            Gizmos.DrawLine(pos, pos + wheel.transform.rotation * offset * spd / 10);
        }
    }

}


// TankWheel.cs
using UnityEngine;
// Клас колеса танка
public class TankWheel : MonoBehaviour
{
    public enum WheelType { Left, Right }

    WheelCollider wheel;
    Transform mesh;
```

```
    float wheelLength;
    float wheelAngularSpeed;

    public WheelType type = WheelType.Left;
    public WheelCollider Wheel { get => wheel; }
    public float Speed { get => GetWheelSpeed(); set => SetWheelSpeed(value); }
    public Transform Mesh { get => mesh; set => mesh = value; }
    // Метод, що визивається на початку
    void Start()
    {
        wheel = GetComponent<WheelCollider>();
        wheelLength = wheel.radius * 2 * Mathf.PI;
    }
    // Отримання швидкості повертання колеса
    float GetWheelSpeed()
    {
        float angularSpeed = wheel.rpm / 60;
        return wheelLength * angularSpeed;
    }
    // Встановлення швидкості повертання колеса
    void SetWheelSpeed(float speed)
    {
        wheelAngularSpeed = speed / wheelLength;
    }
    // Метод, що визивається кожен кадр
    void Update()
    {
        mesh.Rotate(wheelAngularSpeed, 0, 0);
    }

}

// VisionManager.cs
using UnityEngine;
// Клас віртуальної камери ШІ-гравця
public class VisionManager : MonoBehaviour
{
    [SerializeField] Transform visionPoint;
    [SerializeField] LayerMask layerMask;
      [SerializeField] float visionDistance = 100;

    [SerializeField] float visionAngle = 90;
    // Перевірка видимості об’єкта
    public bool DoISeeIt(Vector3 point, bool useVisionAngles = true)
    {
        Vector3 me = visionPoint.position;
        Vector3 delta = point - me;
        float deltaDist = delta.magnitude;

        return deltaDist < visionDistance
            && (!useVisionAngles || Vector3.Angle(visionPoint.forward, delta) <
visionAngle)
            && !Physics.Raycast(new(me, delta.normalized), deltaDist, layerMask);
    }
    // Перевірка погляду на об’єкт
    public bool DoILookAt(Vector3 point, float maxAngle)
    {
        Vector3 lookDirection = visionPoint.forward;
        Vector3 delta = point - visionPoint.position;
```

```
        float angle = Vector3.Angle(lookDirection, delta);

        return angle < maxAngle;
    }
}

// Teams.cs

using System;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
using System.Linq;
// Клас, що відповідає за команди сутностей
public class Teams : MonoBehaviour
{
    static Teams instance;

    readonly Dictionary<Team, List<Entity>> teams = new();
    readonly Dictionary<Team, List<EnemySpawner>> spawners = new();
    readonly UnityEvent<Entity> onRemove = new UnityEvent<Entity>();

    public static Teams Instance { get => instance; }
    public UnityEvent<Entity> OnRemove { get => onRemove; }


    public UnityEvent<Team> onWin = new();
    // Перелік команд
    public enum Team
    {
        Allies,
        Enemies,
        Hostile, // (hostile against anyone, a team for some projectiles)
    }

    void Awake()
    {
        instance = this;

        foreach (Team team in Enum.GetValues(typeof(Team)))
        {
            teams.Add(team, new List<Entity>());
            spawners.Add(team, new List<EnemySpawner>());
        }

        Entity[] entities = FindObjectsByType<Entity>(FindObjectsSortMode.None);

        OnRemove.AddListener((_) => {
            int allies = GetTeamMemberCount(Team.Allies);
            int enemies = GetTeamMemberCount(Team.Enemies);

            if (allies == 0 && enemies != 0)
            {
                onWin.Invoke(Team.Enemies);
                Debug.Log($"Team {Team.Enemies} won!");
            }
            else if (enemies == 0 && allies != 0)
            {
```

```
                onWin.Invoke(Team.Allies);
                Debug.Log($"Team {Team.Allies} won!");
            }
        });
    }
    // Кількість гравців, що можуть спавнитись
    int GetTeamSpawnableCount(Team team)
    {
        int count = 0;

        foreach (EnemySpawner spawner in spawners[team])
        {
            count += spawner.EnemiesToSpawn;
        }
        return count;
    }
    // Кількість гравців у команді
    public int GetTeamMemberCount(Team team, bool includeSpawners = false)
    {
        return teams[team].Count + (includeSpawners ? GetTeamSpawnableCount(team) :
0);
    }
    // Знайти найближчого гравця
    public Entity FindClosestToMe(Team team, Vector3 me)
    {
        var entities = teams[team].OrderBy((health) => (health.transform.position -
me).magnitude);
        return entities.Any() ? entities.First() : null;
    }
    // Знайти найближчих гравців
    public Entity[] FindClosestToMe(Team team, Vector3 me, int entitiesCount)
    {
        var entities = teams[team].OrderBy((entity) => (entity.transform.position -
me).magnitude);
        return entities.Take(entitiesCount).ToArray();
    }
    // Знайти гравця з найбільшим здоров’ям
    public Entity FindBiggest(Team team)
    {
        var entities = teams[team].OrderBy((entity) => entity.MaxHP);
        return entities.Any() ? entities.Last() : null;
    }
    // Додання учасника команди
    public void AddObject(Entity health)
    {
        teams[health.team].Add(health);
    }
    // Видалення учасника команди
    public void RemoveObject(Entity health)
    {
        teams[health.team].Remove(health);
        onRemove.Invoke(health);
    }
    // Додання спавнера сутностей
    public void AddSpawner(Team team, EnemySpawner spawner)
    {
        spawners[team].Add(spawner);
    }
}
```

```
// Damaging.cs

using UnityEngine;
// Клас об’єкта, що наносить шкоду
public abstract class Damaging : MonoBehaviour
{
    protected Entity owner;
    public Entity Owner { set => owner = value; }
}

// Damage.cs
// Клас об’єкта, що наносить шкоду безпосередньо
using UnityEngine;
using UnityEngine.Events;

public class Damage : Damaging
{
    [SerializeField] int damageAmount = 1;
    [SerializeField] float velocityThreshold = 5f;
    [SerializeField] bool damageOnCollision = true;
    [SerializeField] bool damageOnTrigger = false;
    [SerializeField] Teams.Team team = Teams.Team.Enemies;

    public UnityEvent<Transform> OnDamage;
    // Метод, що визивається при зіткненні об’єкта з колайдером
    void OnCollisionEnter(Collision collision)
    {
        if (!damageOnCollision) return;

        bool enoughVelocity = collision.relativeVelocity.magnitude >=
velocityThreshold;
        bool hasHealth = collision.gameObject.TryGetComponent(out Damageable health);

        if (hasHealth && enoughVelocity) health.ApplyDmg(damageAmount, team, owner);
        OnDamage.Invoke(collision.transform);
    }
    // Метод, що визивається при зіткненні об’єкта з тригером
    void OnTriggerEnter(Collider collision)
    {
        if (!damageOnTrigger) return;

        bool hasHealth = collision.gameObject.TryGetComponent(out Damageable health);

        if (hasHealth) health.ApplyDmg(damageAmount, team, owner);
        OnDamage.Invoke(collision.transform);
    }
}

// Damageable.cs
// Клас об’єкта, якому можна нанести шкоду
using UnityEngine;
using static Teams;

public class Damageable : MonoBehaviour
{
    Entity entity;
    // Метод, що визивається на початку
    void Start()
```

```
    {
        if(!TryGetComponent(out entity)) entity = GetComponentInParent<Entity>();
    }

    public void ApplyDmg(int damageAmount, Entity attacker) =>
entity.Damage(damageAmount, attacker);
    public void ApplyDmg(int damageAmount, Team team, Entity attacker) =>
entity.Damage(damageAmount, team, attacker);
}

// KillRecorder.cs

using UnityEngine;
using UnityEngine.Events;
// Клас, що слідкує за вбивствами
public class KillRecorder : MonoBehaviour
{
    public UnityEvent<Entity> OnDamage { get; set; }
    public UnityEvent<Entity> OnKill { get; set; }
    // Метод, що визивається на початку
    private void Awake()
    {
        OnDamage = new();
        OnKill = new();
        GetComponent<Entity>().KillRecorder = this;
    }
}

// Gun.cs

using UnityEngine;
using UnityEngine.Events;
// Клас зброї
public class Gun : MonoBehaviour
{
    // Види зброї
    public enum GunType
    {
        main,
        additive,
        grenade
    }
    // Режими стрільби
    public enum ShootingMode
    {
        single,
        turnSprayOn,
        turnSprayOff
    }
    // Стани зброї
    public enum State
    {
        ready,
        notReady,
        reloadingMags,
    }

    [SerializeField] GunType type = GunType.main;
```

```
[SerializeField] GameObject projectile;
[SerializeField] Transform projectileSpawnPoint;

Transform projectileHandler;
[SerializeField] UIManager uiManager;
[SerializeField] float raycastDistance = 100;

[SerializeField] protected Transform raycastPoint;

[SerializeField] float reloadingTime = 5;
[SerializeField] float magReloadingTime = 5;
[SerializeField] int ammoCount = 30;
[SerializeField] int ammoCountMax = 30;
[SerializeField] int mags = 0;
[SerializeField] int magsMax = 0;

[SerializeField] float angleError = 2;

[SerializeField] bool useAmmo = true;

[SerializeField] Controllable character;

float sprayAccuracyCoef = 1;
float movementAccuracyCoef = 1;

readonly UnityEvent onShot = new();

State state = State.ready;
bool spray = false;
LayerMask raycastMask;

Timers.CooldownTimer accuracyTimer;

Entity me;

public GunType Type { get => type; }
public Transform GunSpawnPoint { get => projectileSpawnPoint; }
public int AmmoCount { get => ammoCount; }
public int TotalAmmoCount { get => ammoCount + mags * ammoCountMax; }
public int Mags { get => mags; }
public UnityEvent OnShot { get => onShot; }
public State GunState { get => state; }

// Метод, що визивається на початку
private void Awake()
{
    if (uiManager != null) uiManager.UpdateAmmoText(type, $"{ammoCount}");
    if (raycastPoint == null) raycastPoint = projectileSpawnPoint;
    raycastMask = GameManager.Instance.RaycastLayerMask;
    projectileHandler = GameManager.Instance.ProjectileHandler;
    accuracyTimer = new Timers.CooldownTimer(0.3f);
    me = GetComponent<Entity>();
}
// Стрільба
public void Shoot(ShootingMode mode)
{
    spray = mode switch
    {
        ShootingMode.turnSprayOn => true,
```

```
            _ => false,
        };
        if (mode == ShootingMode.turnSprayOff) return;

        ShootAndReload();
    }
    // Стрільба з перезарядкою
    private void ShootAndReload()
    {
        if (gameObject == null || state != State.ready) return;

        bool accurate = accuracyTimer.Check();
        accuracyTimer.Reset();

        Quaternion error = Quaternion.Euler(0, 0, Random.value * 360)
                * Quaternion.Euler(angleError * Random.value * ((accurate ? 0 :
sprayAccuracyCoef) + movementAccuracyCoef), 0, 0);

        GameObject projectileObj = Instantiate(
            projectile,
            projectileSpawnPoint.position,
            Quaternion.LookRotation(projectileSpawnPoint.rotation * (error *
Vector3.forward)),
            projectileHandler
        );
        if (useAmmo) ammoCount--;
        state = State.notReady;

        projectileObj.GetComponent<Damaging>().Owner = me;

        Timers.Delay(reloadingTime, () => {
            if (ammoCount > 0) state = State.ready;
            else
            {
                if (mags < 1) return;
                mags--;
                state = State.reloadingMags;
                character.SetReloadAnim(GunType.main, true);
                Timers.Delay(magReloadingTime, () => {
                    ammoCount = ammoCountMax;
                    state = State.ready;
                    character.SetReloadAnim(GunType.main, false);
                    if (spray) ShootAndReload();
                });
                return;
            }
            if (spray) ShootAndReload();
        });

        onShot.Invoke();
    }
    // Проведення променя зі зброї
    public Vector3 RayCastGun()
    {
        Vector3 forward = raycastPoint.forward;
        Vector3 gunPos = raycastPoint.position;
        return Physics.Raycast(gunPos, forward, out RaycastHit hitInfo,
raycastDistance, raycastMask) ? hitInfo.point : gunPos + forward * raycastDistance;
    }
```

```
    public void SetSprayAccuracyCoef(float value)
    {
        sprayAccuracyCoef = value;
    }

    public void SetMovementAccuracyCoef(float value)
    {
        movementAccuracyCoef = value;
    }

    public void Refill()
    {
        mags = magsMax;
    }
}

// Turret.cs

using UnityEngine;
// Клас зброї, що повертається та цілиться
public class Turret : Gun
{
    [Tooltip("In degrees/sec")]
    [SerializeField] Vector2 rotationSpeed = Vector2.one*20;

    [SerializeField] Transform root;
    [SerializeField] Transform turret;
    [SerializeField] Transform gun;

    [SerializeField] Vector2 verticalAngleBounds = Vector2.up * 10;

    [Tooltip("Local rotation axis for turret")]
    [SerializeField] Vector3 turretRotationAxis = Vector3.up;

    [Tooltip("Local rotation axis for barrel")]
    [SerializeField] Vector3 gunRotationAxis = Vector3.right;

    Vector3 tmpLocalDirHor;

    Vector2 targetRotationAngles;

    Vector2 currentRotationAngles;

    bool ignoreLookPoint = false;

    public bool IgnoreLookPoint { set => ignoreLookPoint = value; }

    float To180sys(float val) => (val - 540) % 360 + 180;

    Vector2 To180sys(Vector2 angles)
    {
        angles.Set(
            To180sys(angles.x),
            To180sys(angles.y)
        );
        return angles;
    }
```

```
    Vector2 ClampVector(Vector2 vec, Vector2 max)
    {
        return new Vector2(
            Mathf.Clamp(vec.x, -max.x, max.x),
            Mathf.Clamp(vec.y, -max.y, max.y)
        );
    }
    // Оновлення орієнтації турелі
    void UpdateTurretRotation()
    {
        Vector2 delta = To180sys(targetRotationAngles - currentRotationAngles);

        currentRotationAngles += ClampVector(delta, rotationSpeed * Time.deltaTime);
        currentRotationAngles = To180sys(currentRotationAngles);

        turret.localRotation = Quaternion.Euler(turretRotationAxis *
currentRotationAngles.x);
        gun.localRotation = Quaternion.Euler(gunRotationAxis * -
currentRotationAngles.y);
    }
    // Метод, що визивається кожен кадр
    void Update()
    {
        UpdateTurretRotation();
    }

    void OnDrawGizmosSelected()
    {
        if (!Application.isPlaying) return;
        Gizmos.DrawLine(turret.position, turret.position + tmpLocalDirHor);
    }
    // Метод повороту турелі
    public void Rotate(Vector2 angles)
    {
        targetRotationAngles = angles;
    }
    // Метод повороту турелі на точку
    public void LookAt(Vector3 lookPoint)
    {
        if (ignoreLookPoint) return;
        Quaternion localRotation = Quaternion.Inverse(root.rotation);

        Vector3 localDirTurret = localRotation * (lookPoint -
turret.position).normalized;
        Vector3 localDirGun = localRotation * (lookPoint -
raycastPoint.position).normalized;

        Vector3 localDirHor = Vector3.ProjectOnPlane(localDirTurret,
Vector3.up).normalized;
        Vector3 localDirVert = Vector3.ProjectOnPlane(localDirGun, Quaternion.Euler(0,
targetRotationAngles.x, 0) * Vector3.right).normalized;

        targetRotationAngles.Set(
            Quaternion.LookRotation(localDirHor).eulerAngles.y,
            Mathf.Clamp(-
To180sys(Quaternion.LookRotation(localDirVert).eulerAngles.x), verticalAngleBounds.x,
verticalAngleBounds.y)
        );
    }
```

```
}

// AmmoManager.cs

using UnityEngine;
// Клас боєприпасів персонажа
public abstract class AmmoManager : MonoBehaviour
{
    public abstract bool IsReloading { get; }
    public abstract int Ammo { get; }
    public abstract int TotalAmmo { get; }
    public virtual int GrenadeAmmo => 0;
}

// GunManager.cs

using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using static Gun;
// Клас зброї персонажа
public class GunManager<T> : AmmoManager where T : Gun
{
    protected Dictionary<GunType, List<T>> guns = new();

    protected T[] allGuns;

    private T mainGun;
    private T grenades;

    public T MainGun { get => mainGun; }

    public T Grenades { get => grenades; }

    public override int Ammo { get => mainGun.AmmoCount;  }
    public override int TotalAmmo { get => mainGun.TotalAmmoCount; }
    public override int GrenadeAmmo { get => grenades.AmmoCount; }

    public override bool IsReloading { get => mainGun.GunState == State.reloadingMags;
}
    // Метод, що визивається на початку
    void Awake()
    {
        allGuns = GetComponents<T>();

        foreach (GunType t in Enum.GetValues(typeof(GunType))) guns[t] = new();

        foreach (var g in allGuns) guns[g.Type].Add(g);

        mainGun = guns[GunType.main].First();
        if (guns[GunType.grenade].Any()) grenades = guns[GunType.grenade].First();
    }

    public Vector3 RayCastGun() => mainGun.RayCastGun();
    public void Shoot(GunType type, ShootingMode mode = ShootingMode.single) =>
guns[type].ForEach((gun) => gun.Shoot(mode));
    public void SetSprayAccuracyCoef(float value) =>
mainGun.SetSprayAccuracyCoef(value);
```

```
    public void SetMovementAccuracyCoef(float value) =>
mainGun.SetMovementAccuracyCoef(value);
    // Перезарядка
    public void Reload()
    {
        foreach (var gun in allGuns) gun.Refill();
    }
}

public class GunManager : GunManager<Gun> { }

// TurretManager.cs

using UnityEngine;
// Клас турелей персонажа
public class TurretManager : GunManager<Turret>
{
    // Вимкнення керування туреллю
    public void IgnoreLookPoint(bool on)
    {
        foreach (var turret in allGuns) turret.IgnoreLookPoint = on;
    }
    // Направлення турелі на точку
    public void LookAt(Vector3 Direction)
    {
        foreach (Turret g in allGuns) g.LookAt(Direction);
    }
    // Поворот турелі
    public void Rotate(Vector2 angles, Gun.GunType type)
    {
        foreach (var turret in guns[type]) turret.Rotate(angles);
    }
}

// Projectile.cs
using UnityEngine;
// Клас кулі
public class Projectile : MonoBehaviour
{
    [SerializeField] float projectileSpeed = 120f;
    [SerializeField] Transform trail;

    Rigidbody rb;
    // Метод, що визивається на початку
    private void Start()
    {
        rb = GetComponent<Rigidbody>();
        rb.velocity += transform.forward * projectileSpeed;
    }
}

// CameraController.cs

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
// Клас управління камерою
public class CameraController : ViewController
{
```

```
[SerializeField] float rotationAmount = 2f;
[SerializeField] float raycastDistance = 100;
[SerializeField] float zoomSpeed = 0.05f;
[SerializeField] float smoothing = 1f;
[SerializeField] float shootShakeIntensity = 0.7f;
[SerializeField] float zoomOutShakeMultiplier = 0.5f;

[SerializeField] Transform cursorBig;
[SerializeField] Transform cursorSmall;

LayerMask raycastMask;
LayerMask occlusionRaycastMask;

Vector2 targetRotation;
Quaternion targetRotationQuaternion;
Transform follow;
Vector3 cameraOffset;
Vector3 cameraZoomOffset;
Vector3 followPoint;
Camera cam;

float zoomVal = 0;
float zoomValTarget = 0;
bool zoomingState = false;
float smoothingMultiplier = 1;

Vector3 cursorSmallGlobalPos;

List<Component> deactivateOnZoom = new();

enum Zoom
{
    Out = 0,
    In = 1,
}

public Controllable Follow {
    set {
        follow = value.CameraPivot;
        cameraOffset = value.CameraOffset;
        cameraZoomOffset = value.CameraZoomOffset;
    }
}

public Vector2 TargetRotation { get => targetRotation; }
public List<Component> DeactivateOnZoom { set => deactivateOnZoom = value; }
// Метод, що визивається на початку
void Start()
{
    cam = GetComponent<Camera>();
    raycastMask = GameManager.Instance.RaycastLayerMask;
    occlusionRaycastMask = GameManager.Instance.OcclusionRaycastLayerMask;
}
// Приховування текстур, що заважають камері при наближенні
public void EnableOccludingMeshes(bool on)
{
    foreach (SkinnedMeshRenderer mesh in deactivateOnZoom.Cast<SkinnedMeshRenderer>())
    {
```

```
            mesh.enabled = on;
        }
        zoomingState = on;
    }

    void LateUpdate()
    {
        if (follow == null) return;

        zoomVal = Mathf.Lerp(zoomVal, zoomValTarget, zoomSpeed);
        Vector3 offsetTarget = transform.rotation * Vector3.Lerp(cameraOffset,
cameraZoomOffset, zoomVal);

        bool zoomed = zoomVal > 0.9f;

        if (zoomingState && zoomed) EnableOccludingMeshes(false);
        else if (!zoomingState && !zoomed) EnableOccludingMeshes(true);

        smoothingMultiplier = zoomValTarget == 1 ? Mathf.Lerp(smoothingMultiplier, 1, 0.01f)
: 0;

        followPoint = Vector3.Lerp(followPoint, follow.position, Mathf.Max(smoothing *
Time.deltaTime, smoothingMultiplier));

        Vector3 offset = Physics.Raycast(followPoint, offsetTarget, out RaycastHit hitInfo,
offsetTarget.magnitude, occlusionRaycastMask) ? offsetTarget.normalized * hitInfo.distance :
offsetTarget;
        transform.position = followPoint + offset;

        bool cursorVisible = Vector3.Dot(cursorSmallGlobalPos - transform.position,
cam.transform.forward) > 0;
        GameObject cursor = cursorSmall.gameObject;
        if (cursor.activeSelf != cursorVisible) cursor.SetActive(cursorVisible);
    }

    void ClampView(ref Vector2 vector, float val) => vector.y = Mathf.Clamp(vector.y, -val,
val);
    // Поворот камери
    public void RotateLook(Vector2 input)
    {
        const float deadzone = 0.1f;

        targetRotation += input;
        ClampView(ref targetRotation, 90 - deadzone);

        targetRotationQuaternion = Quaternion.Euler(-targetRotation.y, targetRotation.x, 0);
        transform.rotation = Quaternion.Lerp(
            transform.rotation,
            targetRotationQuaternion,
            rotationAmount
        );

        Vector3 rotationVec = targetRotationQuaternion * Vector3.forward;
        cursorBig.position = cam.WorldToScreenPoint(transform.position + rotationVec);
    }
    // Вектор погляду
    public Vector3 ForwardView() => transform.position + transform.forward *
raycastDistance;
```

```
    // Проведення променя у напрямку курсора
    public Vector3 RaycastCursor()
    {
        Vector3 raycastDir = targetRotationQuaternion * Vector3.forward;

        bool hit = Physics.Raycast(transform.position, raycastDir, out RaycastHit hitInfo,
raycastDistance, raycastMask);

        return hit ? hitInfo.point : transform.position + raycastDir * raycastDistance;
    }
    // Проведення променя у напрямку курсора та отримання об’єкта, в який влучив
промінь
    public override bool RaycastItem(out GameObject obj)
    {
        Vector3 raycastDir = targetRotationQuaternion * Vector3.forward;

        bool hit = Physics.Raycast(transform.position, raycastDir, out RaycastHit hitInfo,
interactDistance, raycastMask);

        obj = hit ? hitInfo.collider.gameObject : null;

        return hit;
    }
    // Встановлення додаткового курсора
    public void SetAdditiveCursor(Vector3 globalPos)
    {
        cursorSmall.position = cam.WorldToScreenPoint(globalPos);
        cursorSmallGlobalPos = globalPos;
    }

    // Наближення камери
    public void SetZoom (bool on)
    {
        zoomValTarget = (float)(on ? Zoom.In : Zoom.Out);
    }
    // Імпульс камери від пострілу
    public void ShootShake()
    {
        float shake = zoomingState ? shootShakeIntensity * zoomOutShakeMultiplier :
shootShakeIntensity;
        transform.Rotate(new(Mathf.Sign(Random.Range(-1, 1)) * shake,
Mathf.Sign(Random.Range(-1, 1)) * shake));
    }
}


// Весь код було вирішено не надавати, бо це займе 100+ сторінок додатка. Проект має
понад 3.5 тисяч строк власного коду. Основні класи програми, які варті уваги, були
надані вище.
```