МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Завідувач кафедри інженерії

програмного забезпечення

_____Євген ДАВИДЕНКО

«___» _____2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ МАГІСТРА

Комп'ютерна 3D гра з використанням штучного інтелекту

Спеціальність 121 Інженерія програмного забезпечення

Освітня програма «Інженерія програмного забезпечення»

Здобувач _____Віталій ДИМІНСЬКИЙ

«___» _____2024 р.

Керівник канд. техн. наук, доцент _____Гліб ГОРБАНЬ

«___» _____2024 р.

Миколаїв – 2024

Чорноморський національний університет імені Петра Могили

(повне найменування закладу вищої освіти)

Факультет	Комп’ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Другий (магістерський)
Освітній ступень	Магістр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інженерії
програмного забезпечення
_____Євген ДАВИДЕНКО
«___» _____2024 р.

ЗАВДАННЯ

на кваліфікаційну магістерську роботу здобувача

Димінського Віталія Івановича

(прізвище, ім’я, по батькові здобувача)

1. Тема кваліфікаційної роботи
Комп’ютерна 3D гра з використанням штучного інтелекту

Затверджена наказом ЧНУ ім. Петра Могили від «_4_» _вересня_ 2024 р. №_220_

2. Строк представлення кваліфікаційної роботи «___»_____2024 р.

3. Очікуваний результат роботи та початкові дані якщо такі потрібні

Очікуваним результатом є програмне забезпечення у вигляді комп’ютерної 3D-гри із використанням штучного інтелекту у жанрі Real-Time Strategy

4. Перелік питань, що підлягають розробці:

Аналіз предметної області, порівняльний аналіз аналогів, визначення вимог та функціоналу системи, моделювання, проектування програмного застосунку, кодування та тестування застосунку.

5. Перелік графічних матеріалів

Презентація

6. Консультанти:

Консультант	Кафедра (організація)	Частина роботи

Дата видачі завдання «____» _____20___р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: «Комп'ютерна 3D гра з використанням штучного інтелекту»

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КМР	04.09.2024р.	05.09.2024р.	виконано
2.	Огляд літератури за темою роботи	06.09.2024р.	11.09.2024р.	виконано
3.	Складання календарного плану КМР	12.09.2024р.	12.09.2024р.	виконано
4.	Аналіз предметної області	13.09.2024р.	16.09.2024р.	виконано
5.	Розробка проєктних рішень	16.09.2024р.	20.09.2024р.	виконано
6.	Моделювання та конструювання ПЗ	23.09.2024р.	30.09.2024р.	виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування, розробка керівництва користувача	30.09.2024р.	20.11.2024р.	виконано
8.	Відгук керівника КМР	21.11.2024р.	23.11.2024р.	виконано
9.	Оформлення КМР та презентації	23.11.2024р.	27.11.2024р.	виконано
10.	Попередній захист	28.11.2024р.	28.11.2024р.	виконано
11.	Рецензування	12.12.2024р.	12.12.2024р.	виконано
12.	Завершення оформлення КМР та презентації	30.11.2024р.	12.12.2024р.	виконано
13.	Захист кваліфікаційної роботи	19.12.2024р.	20.12.2024р.	виконано

Здобувач _____ Віталій ДИМІНСЬКИЙ

«___» _____20___р.

Керівник роботи
канд. техн. наук,
доцент _____ Гліб ГОРБАНЬ

«___» _____20___р.

АНОТАЦІЯ

до кваліфікаційної магістерської роботи

«Комп’ютерна 3D гра з використанням штучного інтелекту»

Здобувач 608м гр.: Димінський Віталій Іванович

Керівник: канд. техн. наук, доцент Горбань Г. В.

Захищена: «_20_» грудня 2024 р.

Індустрія комп’ютерних ігор є одним із найбільш динамічних і перспективних напрямів у сфері розваг та програмного забезпечення. Жанр стратегій у реальному часі (RTS) залишається популярним завдяки своїй здатності залучати гравців у складний, динамічний і тактичний ігровий процес. Інтеграція штучного інтелекту (ШІ) у цей жанр дозволяє створювати більш реалістичні та адаптивні ігрові світи, де поведінка противників та NPC підлаштовується під дії гравця.

Робота присвячена розробці комп’ютерної 3D-гри у жанрі RTS на платформі Unity з використанням штучного інтелекту. Платформа Unity надає широкий набір інструментів для створення багатофункціональних ігрових проектів з високою графічною якістю та складними механіками. Використання ШІ забезпечує створення інтелектуальних алгоритмів для адаптації складності гри, взаємодії NPC із середовищем та гравцем, а також реалізації нелінійних сценаріїв.

У першому розділі аналізується актуальність розробки ігор у жанрі RTS, проводиться огляд аналогів, визначаються вимоги до майбутньої гри, а також формується технічне завдання для її створення.

Другий розділ присвячений вибору інструментів та фреймворків для розробки, мови програмування та програмного забезпечення для моделювання.

У третьому розділі моделюються ігрові процеси: створюються UML-діаграми, діаграми станів, переходів, компонентів, а також алгоритми реалізації ігрових механік.

Четвертий розділ зосереджений на програмній реалізації гри: розробці UI-елементів, механіки рівнів, інтеграції штучного інтелекту та тестуванні створеного прототипу.

Об'єкт роботи: розробка комп'ютерної 3D-ігри у жанрі стратегії в реальному часі з використанням штучного інтелекту.

Предмет роботи: аналіз технологій використання штучного інтелекту та застосування їх в процесі розробки гри.

Мета: створення 3D-ігри на платформі Unity, що демонструє можливості використання ШІ для поліпшення ігрового процесу та взаємодії користувача з грою.

Для досягнення мети необхідно виконати такі **завдання**:

1. Дослідити актуальність розробки 3D ігор із впровадженням ШІ.
2. Провести аналіз існуючих RTS-ігор для визначення ключових механік, очікувань гравців і недоліків.
3. Визначити вимоги до гри та сформулювати технічне завдання.
4. Розробити загальний алгоритм гри з інтеграцією ШІ.
5. Реалізувати прототип із основними функціями та протестувати його.

КМР викладена на 88 сторінок, вона містить 4 розділи, 80 ілюстрацій, 7 таблиць, 21 джерел в переліку посилань.

ABSTRACT

of the Master`s Thesis

" Computer 3D Game Using Artificial Intelligence"

Student of group 608м: Dyminskyi Vitalii Ivanovich

Supervisor: Ph. D., Associate Professor Horban H. V.

The computer gaming industry is one of the most dynamic and promising fields in the entertainment and software development sectors. The real-time strategy (RTS) genre remains popular due to its ability to engage players in complex, dynamic, and tactical gameplay. Integrating artificial intelligence (AI) into this genre allows for the creation of more realistic and adaptive game worlds where the behavior of opponents and NPCs adjusts to player actions.

This thesis is dedicated to developing a computer 3D game in the RTS genre on the Unity platform using artificial intelligence. Unity provides a wide range of tools for creating multifunctional game projects with high graphical quality and intricate mechanics. The use of AI enables the creation of intelligent algorithms for adapting game difficulty, facilitating interactions between NPCs, the environment, and players, as well as implementing nonlinear scenarios.

The first chapter analyzes the relevance of RTS game development, reviews existing analogs, defines requirements for the future game, and formulates the technical specifications for its creation.

The second chapter focuses on selecting tools and frameworks for development, programming languages, and modeling software.

The third chapter models gameplay processes: creating UML diagrams, state diagrams, transitions, components, and algorithms for implementing game mechanics.

The fourth chapter focuses on the software implementation of the game: developing UI elements, level mechanics, AI integration, and testing the created prototype.

Object of the thesis: Development of a computer 3D game in the RTS genre with artificial intelligence.

Subject of the thesis: Analysis of AI technologies and their application in the game development process.

Goal: To create a 3D game on the Unity platform that demonstrates the capabilities of AI for improving the gaming process and user interaction with the game.

To achieve the goal, the following **tasks** must be completed:

1. Investigate the relevance of developing 3D games with AI implementation.
2. Analyze existing RTS games to identify key mechanics, player expectations, and shortcomings.
3. Define game requirements and formulate technical specifications.
4. Develop a general algorithm for the game with AI integration.
5. Implement a prototype with core features and test it.

The master's thesis comprises 88 pages, includes 4 chapters, 80 illustrations, 7 tables, and 21 references in the bibliography.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ПЗ – програмне забезпечення

ОС – операційна система

ПК – персональний комп’ютер

ШІ – штучний інтелект

ЛКМ – Ліва кнопка миші

ПКМ – Права кнопка миші

RTS – Real-Time Strategy

NPC – Non-player character

ВСТУП

Актуальність теми полягає в тому, що в сучасному світі комп’ютерні 3D-ігри займають важливе місце в індустрії розваг і програмного забезпечення. З розвитком технологій та збільшенням потужності комп’ютерних систем, 3D-ігри стають дедалі більш реалістичними, пропонуючи гравцям захоплюючі віртуальні світи. Зокрема, жанр стратегій у реальному часі (RTS) користується великою популярністю серед гравців, адже він дозволяє управляти ресурсами, будувати бази та вести битви в режимі реального часу. Цей жанр вимагає від гравця стратегічного мислення та швидкого реагування, що робить його особливо привабливим для любителів інтелектуальних ігор.

Розробка 3D-ігор на платформі Unity стала однією з найперспективніших можливостей для розробників. Unity надає зручний інтерфейс та потужний набір інструментів для створення ігор, що дозволяє зосередитися на творчому процесі та розробці ігрової механіки. Зручність кросплатформності також робить Unity ідеальним вибором для розробки ігор, які можуть бути запущені на різних платформах, включаючи ПК, консолі та мобільні пристрої.

Однією з ключових інновацій у сучасній розробці ігор є використання ШІ. ШІ може забезпечити адаптивну поведінку противників, покращити управління ресурсами та оптимізувати ігровий процес, роблячи його більш реалістичним та захоплюючим. Важливість використання ШІ в ігровій індустрії зростає, оскільки гравці шукають нові виклики та більш глибокі ігрові досвіди.

Таким чином, розробка комп’ютерних 3D-ігор на Unity з використанням штучного інтелекту в жанрі RTS є надзвичайно актуальною. Це дозволяє не лише задовольнити попит на інноваційні та цікаві ігри, але й сприяє розвитку індустрії в цілому, залучаючи нові інвестиції та забезпечуючи нові можливості для творчого самовираження розробників.

Об’єкт роботи: розробка комп’ютерної 3D-ігри у жанрі стратегії в реальному часі з використанням штучного інтелекту.

Предмет роботи: аналіз технологій використання штучного інтелекту та застосування їх в процесі розробки гри.

Мета: створення 3D-ігри на платформі Unity, що демонструє можливості використання ШІ для поліпшення ігрового процесу та взаємодії користувача з грою.

Для досягнення поставленої цілі необхідно виконати наступні **завдання:**

- проведення аналізу сфери штучного інтелекту в ігрових технологіях;
- огляд існуючих ігор жанру RTS;
- визначення основних функцій гри та користувацьких очікувань;
- оформлення специфікацій вимог до ПЗ;
- створення загального алгоритму розробки гри.

Сфера застосування: комп’ютерні ігри жанру RTS із використанням штучного інтелекту на платформі Unity надають гравцям можливість зануритися у складний ігровий процес, де важливу роль відіграють як стратегічне мислення, так і динамічні реакції на дії противників. Важливим аспектом таких ігор є не лише створення якісного геймплею, але й можливість розвитку навичок планування, тактики та швидкої реакції у гравців. Крім того, впровадження ШІ в такі ігри дозволяє зробити ігровий процес більш захоплюючим та непередбачуваним.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Актуальність предметної сфери

Комп'ютерні 3D-ігри вже багато років утримують свої позиції серед найпопулярніших форм цифрових розваг. Це стало можливим завдяки високій продуктивності сучасних ПК, потужним графічним процесорам та розширеним інтерактивним можливостям. Щороку технології вдосконалюються, дозволяючи розробникам створювати все більш реалістичні та захоплюючі ігрові світи. Комп'ютерні 3D-ігри пропонують користувачам унікальний досвід завдяки складним сюжетам, динамічній графіці та можливості зануритися у віртуальні світи з високим рівнем деталізації, що робить цей вид розваг постійно затребуваним серед аудиторії. Сьогодні попит на якісні ігри, особливо 3D-проєкти, стрімко зростає завдяки ширшій доступності високоякісного обладнання та технологій.

Особливо виділяється платформа Unity, яка є одним з найпопулярніших інструментів для розробки 3D-ігор на ПК та інших пристроях. Unity дозволяє створювати як 2D, так і 3D-ігри, що робить її універсальним вибором для розробників з різним рівнем досвіду. Важливо зазначити, що платформа підтримує мультиплатформність, дозволяючи адаптувати ігри для різних систем, включаючи ПК, iOS, Android та інші платформи. Ця особливість робить Unity особливо актуальною в сучасній тенденції до створення ігор, доступних на різних платформах, що дозволяє значно розширити аудиторію. Розробка ігор на Unity дозволяє підтримувати та оновлювати проєкти більш ефективно, що є важливим аспектом у динамічній індустрії ігор [1].

Однією з ключових переваг Unity є її велика спільнота розробників, наявність численних ресурсів та підтримка передових технологій, таких як штучний інтелект. Створення 3D-ігор на Unity із застосуванням ШІ відкриває нові горизонти в індустрії, дозволяючи розробникам впроваджувати складні механіки, що адаптуються до дій гравця. Unity підтримує інтеграцію з бібліотеками машинного навчання, такими як TensorFlow та OpenAI, що

дозволяє створювати динамічні сценарії, інтелектуальних ворогів та NPC, здатних реагувати на дії гравця в реальному часі.

Комп'ютерні 3D-ігри на платформі Unity із застосуванням штучного інтелекту є важливою віхою у розвитку ігрової індустрії. ШІ надає можливість створювати більш складні ігрові системи, які можуть навчатися та адаптуватися до дій гравця. Це відкриває нові горизонти для розробників у створенні реалістичних та захоплюючих ігрових світів, де персонажі можуть взаємодіяти з гравцем так, ніби вони мають власну волю [4].

На платформі Unity ШІ може бути використаний у різних аспектах гри: від створення інтелектуальних ворогів, які адаптують свої дії до поведінки гравця, до розробки складних NPC, що виконують різні завдання та реагують на зміни в ігровому середовищі. Використання ШІ дозволяє розробникам створювати ігровий світ, який "живе" і реагує на дії користувача, що підвищує рівень занурення в гру.

Unity пропонує широкі можливості для інтеграції з популярними бібліотеками машинного навчання, такими як TensorFlow і OpenAI, що дозволяє розробникам використовувати передові алгоритми для створення реалістичних та адаптивних ігрових механік. Завдяки штучному інтелекту можна створювати ворогів, які вивчають стратегії гравця, щоб ефективніше протидіяти його діям, роблячи ігровий процес непередбачуваним і цікавим з постійними новими викликами.

Штучний інтелект у 3D-іграх на Unity дозволяє створювати складні ігрові механіки, які автоматично адаптуються до рівня майстерності гравця. Аналізуючи дії гравця, ШІ може змінювати складність рівнів для підтримання інтересу та стимулювання розвитку навичок, роблячи гру привабливою для широкої аудиторії, незалежно від їхнього досвіду [4].

Використання ШІ також дозволяє створювати нелінійні сценарії, де кожне рішення гравця впливає на розвиток сюжету. NPC з ШІ можуть реагувати на вибори гравця, пропонуючи різноманітні варіанти розвитку подій, що робить кожне проходження гри унікальним і тримає увагу користувача тривалий час.

У створенні мультиплеєрних ігор ШІ відіграє ключову роль. В Unity розробники можуть створювати розумних ботів, які допомагають підтримувати баланс у грі, коли недостатньо реальних гравців. Це дозволяє підтримувати активність у грі незалежно від змін кількості гравців.

Мобільні 3D-ігри з використанням ШІ мають великий потенціал завдяки підтримці Unity на мобільних платформах. ШІ адаптується до поведінки гравця, пропонуючи нові виклики та динамічні рівні, що підвищує інтерес до гри та збільшує тривалість її використання. Unity дозволяє легко інтегрувати ШІ у мобільні 3D-ігри, роблячи їх більш привабливими для сучасних користувачів.

Комп'ютерні 3D-ігри на платформі Unity є важливим інструментом для розвитку технологічної галузі. Створення таких ігор вимагає високих компетенцій у програмуванні, дизайні та використанні новітніх технологій, включаючи ШІ. Відомі розробники, такі як ті, що створили Subnautica, Hearthstone, або Escape from Tarkov, використовують Unity для успішних проектів, що стимулює розвиток галузі та появу нових можливостей для інновацій як великих студій, так і незалежних розробників.

Таким чином, комп'ютерні 3D-ігри на платформі Unity є перспективним напрямком у сфері технологій та розваг. Розробка таких ігор дозволяє створювати високоякісні проекти з реалістичною графікою, складними механіками та інноваційними підходами до взаємодії з користувачами. Використання Unity відкриває широкі можливості для розробки ігор, залучаючи велику аудиторію і забезпечуючи стабільний розвиток ринку комп'ютерних ігор.

1.2 Огляд аналогів

Cossacks 3 [5]

Cossacks 3 – це стратегічна гра в реальному часі, яка розгортається в Європі XVII-XVIII століть. Гравці керують економікою, будують армії та беруть участь у масштабних битвах з тисячами юнітів. Гра пропонує як одиночний режим, так і багатокористувацькі кампанії. Вона є ремейком класичної серії "Козаки", з поліпшеною графікою та новими ігровими можливостями.

Таблиця 1.1 – Опис «Cossacks 3»

Назва	Cossacks 3
Виробник	GSC Game World
Мова реалізації	C++
Функції	1. Масштабні битви з до 32 000 юнітів на полі бою. 2. Детальне управління економікою та ресурсами. 3. Багатокористувацький режим на до 8 гравців. 4. Історичні кампанії.
Переваги	1. Велика кількість юнітів на полі бою. 2. Історична достовірність. 3. Різноманітні нації та війська.
Недоліки	1. Баги при запуску та проблеми з оптимізацією. 2. Відсутність інновацій у порівнянні з оригінальними частинами серії.
Вебсайт	https://store.steampowered.com/app/333420/Cossacks_3/

Рисунок 1.1 – Обкладинка гри «Cossacks 3»

Age of Empires II [6]

Age of Empires II – це стратегія в реальному часі, яка охоплює період від раннього Середньовіччя до пізнього середньовіччя. Гравці керують цивілізацією, будуючи економіку, розвиваючи армію та борючись з іншими націями. Гра відзначається деталізованим геймплеєм та великим вибором націй, кожна з яких має унікальні війська та технології.

Таблиця 1.2 – Опис «Age of Empires II»

Назва	Age of Empires II
Виробник	Ensemble Studios
Мова реалізації	C++
Функції	1. 13 унікальних націй з власними технологіями. 2. Розвинена система економіки та розвитку цивілізацій. 3. Кампанії на основі історичних подій. 4. Штучний інтелект з кількома рівнями складності.
Переваги	1. Глибокий стратегічний геймплей. 2. Величезний вибір цивілізацій. 3. Активна спільнота модів та багатокористувацька підтримка.
Недоліки	1. Дещо застаріла графіка (в оригінальній версії). 2. Довга тривалість окремих ігор може втомлювати.
Вебсайт	https://store.steampowered.com/app/813780/

Рисунок 1.2 – Обкладинка гри «Age of Empires II»

Stronghold Crusader 2 [7]

Stronghold Crusader 2 – це гра в жанрі стратегій в реальному часі, дія якої відбувається в часи хрестових походів. Гравці будують замки, керують економікою та обороняються від ворогів, водночас атакуючи фортеці суперників. Гра зосереджена на веденні облоги та управлінні ресурсами в пустельних умовах Близького Сходу.

Таблиця 1.3 – Опис «Stronghold Crusader 2»

Назва	Stronghold Crusader 2
Виробник	Firefly Studios
Мова реалізації	C++
Функції	1. Будівництво замків та управління їх захистом. 2. Реалістична система облоги з використанням облогових машин. 3. Пустельна економіка з управлінням водними ресурсами. 4. Кооперативний режим гри на двох гравців.

Кінець таблиці 1.3

Переваги	1. Унікальний акцент на облоги замків. 2. Інтерактивне оточення та економічна система. 3. Можливість кооперативної гри.
Недоліки	1. Складність керування великою кількістю юнітів. 2. Деякі елементи геймплею можуть здатися застарілими.
Вебсайт	https://store.steampowered.com/app/232890/Stronghold_Crusader_2/

Рисунок 1.3 – Обкладинка гри «Stronghold Crusader 2»

Проведений аналіз аналогів ігор жанру дозволив зрозуміти, що цей жанр продовжує залишатися одним із найулюбленіших серед гравців завдяки своїй здатності поєднувати глибокий стратегічний геймплей із динамічними елементами. Жанр RTS залишається живим і актуальним завдяки своїй здатності поєднувати стратегію, тактику та динаміку. Кожна з проаналізованих ігор вносить свій внесок у розвиток жанру, пропонуючи різні аспекти геймплею та взаємодії. Незважаючи на еволюцію технологій та зміну уподобань гравців, основні елементи стратегічного мислення, управління ресурсами та бойових дій залишаються ключовими у всіх цих іграх. Це свідчить про те, що жанр RTS не тільки витримав випробування часом, але й продовжує розвиватися, пропонуючи нові можливості для стратегічного мислення та креативності.

1.3 Визначення технічного завдання для ігрового застосунку

Технічне завдання передбачає створення та впровадження оригінальної комп'ютерної 3D-ігри, яка забезпечить тривалий і захопливий ігровий процес для користувачів, використовуючи технології штучного інтелекту. Додаток повинен відповідати наступним вимогам [2]:

1. меню при запуску: Після запуску гри повинно з’являтися меню, що дозволяє користувачам готуватися до виконання ігрових завдань.

2. перегляд та запуск збережених етапів гри: Користувач має мати можливість переглядати інформацію про збережені сесії та видаляти їх.

3. розташування кнопок: Усі функціональні кнопки для взаємодії з інтерфейсом повинні бути розміщені так, щоб забезпечити зручність використання на комп’ютерах з різними розмірами екранів.

4. ігрове поле в перспективній проєкції: Основне ігрове поле повинно бути реалізовано у вигляді перспективної проєкції.

5. системні вимоги: Програма повинна працювати на ПК з операційною системою Windows версії не нижче 10.

Перш ніж почати розробку гри, необхідно вибрати платформу, яка відповідатиме наступним критеріям:

1. підтримка комп’ютерних платформ: платформа повинна підтримувати розробку ігор для комп’ютерних систем;

2. створення 3D-ігор: можливість розробки 3D ігор є обов’язковою;

3. зручність використання: інтерфейс розробника має бути зрозумілим та зручним;

4. якісна документація: наявність детальної документації є важливою для швидкого освоєння;

5. безкоштовне розповсюдження: платформа повинна пропонувати безкоштовну версію, що дозволяє створювати ігри без додаткових витрат.

Unity була обрана як одна з найпопулярніших платформ для розробки ігор, оскільки вона відповідає усім наведеним вимогам:

1. підтримка різноманітних операційних систем: Unity дозволяє розробляти ігри для багатьох ОС, включаючи Windows, macOS та Linux, забезпечуючи запуск ігор на різних ПК;

2. потужний інструмент для 3D розробки: Unity надає широкий набір інструментів для моделювання, анімації, освітлення та інших аспектів тривимірної розробки;

3. інтуїтивно зрозумілий інтерфейс: Простий у використанні інтерфейс Unity дозволяє розробникам концентруватися на творчому процесі, мінімізуючи витрати часу на освоєння складних інструментів;

4. розширена документація: Unity забезпечує багатий вибір ресурсів, включаючи офіційні посібники, навчальні курси та матеріали від спільноти, що дозволяє швидко знаходити рішення та вирішувати проблеми;

5. безкоштовне розповсюдження: Unity пропонує безкоштовну версію з усіма необхідними функціями для розробки більшості ігор, а також забезпечує простий експорт ігор на різні платформи без додаткових витрат.

Unity також має зручний інтерфейс, який складається з таких вікон, як Scene (Сцена) для перегляду ігрового поля, Inspector (Інспектор) для налаштування властивостей об'єктів та Project (Проєкт) для зберігання всіх елементів проєкту. Крім цього, Unity пропонує інтегровані сервіси для залучення, утримання та монетизації гравців.

Окрім зручного інтерфейсу, Unity включає редактор, який дозволяє створювати 3D-моделі та анімації. Для розробки ігор у Unity використовується мова програмування C#, яка є досить простою для освоєння та має безліч ресурсів для самостійного навчання. Unity також забезпечує можливість експорту ігор на різні платформи, такі як Windows, macOS та Linux. Більшість налаштувань можна здійснювати безпосередньо в редакторі Unity, що допомагає заощадити час на розробку та тестування [3].

1.4 Специфікація вимог

ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Доступність

Гра повинна бути доступною для всіх користувачів ПК, які відповідають встановленим системним вимогам.

Переносимість

Програмне забезпечення має корректно працювати на операційних системах Windows 10.

Продуктивність

Продуктивність гри залежатиме від технічних характеристик комп’ютера користувача.

Надійність

Дані користувачів є приватними і кожен має свій власний ігровий процес.

ПРИЗНАЧЕННЯ ЗАСТОСУНКУ

Призначення застосунку, для якої розробляється програмне забезпечення

Це ПЗ розроблене для забезпечення користувачам розважального досвіду в жанрі стратегії в реальному часі.

Погодження, що ухвалені в програмній документації

Для створення програмного забезпечення та забезпечення його стабільної роботи будуть використані сторонні ассети та бібліотеки Unity.

ЗАГАЛЬНИЙ ОПИС

Сфера застосування

Застосунок розроблено для використання користувачами з метою відпочинку та розваги.

Характеристики користувачів

Основні характеристики користувачів: наявність ПК який відповідає технічним характеристикам ПЗ.

ВИМОГИ ДО ІНФОРМАЦІЙНОГО ЗАБЕЗПЕЧЕННЯ

Джерела і зміст вхідної інформації (даних)

У цьому ПЗ основним джерелом вхідної інформації є користувач, який самостійно вводить дані, такі як досягнення та ім’я персонажу.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Вимоги до програмного забезпечення можна описати через кілька ключових характеристик, які відображають якість і надійність роботи програми.

Коректність

Ця характеристика відображає здатність програми реалізувати алгоритми без помилок. Коректність охоплює всі етапи розробки – від специфікацій та проєктування до реалізації алгоритмів та структури даних.

Стійкість

Ця характеристика визначає здатність програмного забезпечення здійснювати обробку інформації, зберігаючи при цьому відповідність визначеним специфікаціям.

Відновлюваність

Ця характеристика описує можливість програми адаптуватися до виявлених помилок та їх усунення.

Надійність

Надійність охоплює кілька аспектів, таких як цілісність програми, її живучість, завершеність та працездатність. Цілісність означає здатність ПЗ захищатися від відмов і підтримувати свою працездатність. Живучість відображає контроль програми за вхідними даними під час її роботи. Завершеність означає відсутність помилок у готовому програмному забезпеченні та високу якість тестування. Працездатність характеризує здатність програми відновлюватися після збоїв або інших непередбачених ситуацій.

ВИМОГИ ДО ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

1. Операційна система: Windows 10 64bit.
2. Процесор: 2.4 Ghz i5 or greater or AMD equivalent.
3. Оперативна пам’ять: 4 ГБ RAM.

4. Відеокарт: Graphic card with 2 GB VRAM or higher.
5. Вільне місце: 5 ГБ.
6. DirectX: версії 11.

1.5 Опис загального алгоритму виконання проєкту

Для успішної розробки гри важливо ретельно спланувати весь процес та встановити чіткі часові межі. Тривалість розробки може значно коливатися в залежності від складності проєкту. Створення гри – це багатогранний процес, що включає кілька етапів, кожен з яких реалізує певний аспект, інтегрований у кінцевий продукт [2].

Перший етап – проєктування меню гри. Меню є важливою складовою, адже воно слугує головним інтерфейсом для користувачів. На цьому етапі визначається дизайн інтерфейсу, обирається колірна гамма, а також планується функціональність, структура і навігація меню.

Другий етап – проєктування рівнів. Рівні гри визначають, чим саме займатимуться гравці. Тут розробляються загальні концепції рівнів, їх дизайн, складність і ключові завдання, які необхідно виконати.

Третій етап – моделювання. На цьому етапі створюються 3D-моделі персонажів, об’єктів, ландшафтів та інших елементів гри. Для цих цілей використовуються спеціалізовані програми, такі як Blender, Maya або 3DS Max.

Четвертий етап – розробка. Тут створюється програмне забезпечення гри, яке реалізує всі функції, відповідає за геймплей та інші аспекти. Розробники використовують потужні платформи, такі як Unity, Unreal Engine або GameMaker Studio.

Останній етап – тестування. У цей період гра проходить серію тестів, які перевіряють її стабільність, функціональність та ігровий процес. Цей етап надзвичайно важливий, оскільки дозволяє виявити й виправити помилки, баги та інші недоліки до релізу для широкої аудиторії.

На основі запитань, що виникають під час підготовки до виробництва, складається GDD (документ ігрового дизайну). Цей документ містить елементи,

такі як жанр, концепція, сюжет, персонажі, рівні та складність, а також деталі геймплею. Список може доповнюватись протягом розробки. Це допомагає зменшити кількість запитань і помилок під час створення проєкту. Крім того, це сприяє формуванню цілісного проєкту, що потребуватиме менше виправлень у кінці роботи. Водночас гра буде випущена у бета-версії, щоб мати можливість усунути недоліки, які могли бути пропущені під час розробки.

Висновки до розділу 1

Майбутнє інформаційних технологій безсумнівно пов’язане з розвитком комп’ютерних 3D ігор, які стають все більш популярними серед користувачів. Сучасні ПК мають величезну обчислювальну потужність, що дозволяє створювати складні й візуально вражаючі проєкти, здатні забезпечити гравцям захоплюючий досвід. Ігри в жанрі RTS (стратегії в реальному часі) мають безліч аналогів, тому важливо знайти рішення, яке дозволить виділити проєкт серед конкурентів. Однією з ключових переваг цієї гри є її можливість поєднувати глибокий стратегічний геймплей з якісною графікою та інтуїтивно зрозумілим управлінням. Проте, проєкт також стикається з деякими викликами, такими як оптимізація штучного інтелекту, локалізація для різних ринків, що має вирішити завдання підтримки інтересу та залучення цільової аудиторії. Успішне вирішення цих питань може суттєво підвищити конкурентоспроможність гри на ринку.

2 АНАЛІЗ ФРЕЙМВОРКІВ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

2.1 Вибір рушія для розробки ігрового застосунку

Unity

Unity – це багатофункціональна платформа для створення ігор, яка дозволяє розробляти проєкти для широкого спектра пристроїв, включаючи Windows, macOS, Linux, Android, iOS, Xbox, PlayStation та інші. Середовище розробки Unity вирізняється інтуїтивним інтерфейсом, що полегшує роботу як з візуальними елементами, так і зі скриптами, роблячи його універсальним інструментом для програмістів та дизайнерів. Unity надає можливість створювати графічні об’єкти, керувати їх взаємодією та застосовувати різні ефекти для покращення якості гри. Додатково, платформа пропонує численні інструменти та компоненти, що дозволяють створювати проєкти з мінімальним обсягом кодування, суттєво скорочуючи час розробки [8].

Рисунок 2.1 – Логотип «Unity»

Серед важливих переваг Unity – підтримка кількох мов програмування, таких як C# та JavaScript, що дає розробникам вибір мови, з якою їм зручно працювати. Також Unity активно підтримує розробку додатків у віртуальній та доповненій реальності (VR і AR), що підвищує її привабливість для сучасних розробників. Unity має велику спільноту користувачів, що забезпечує швидкий доступ до навчальних матеріалів, прикладів коду та готових рішень для створення ігор різної складності.

Також варто відзначити, що платформа забезпечує широкою підтримкою спільноти користувачів та наявністю великої кількості навчальних ресурсів, що значно полегшує вирішення технічних питань і пришвидшує процес розробки.

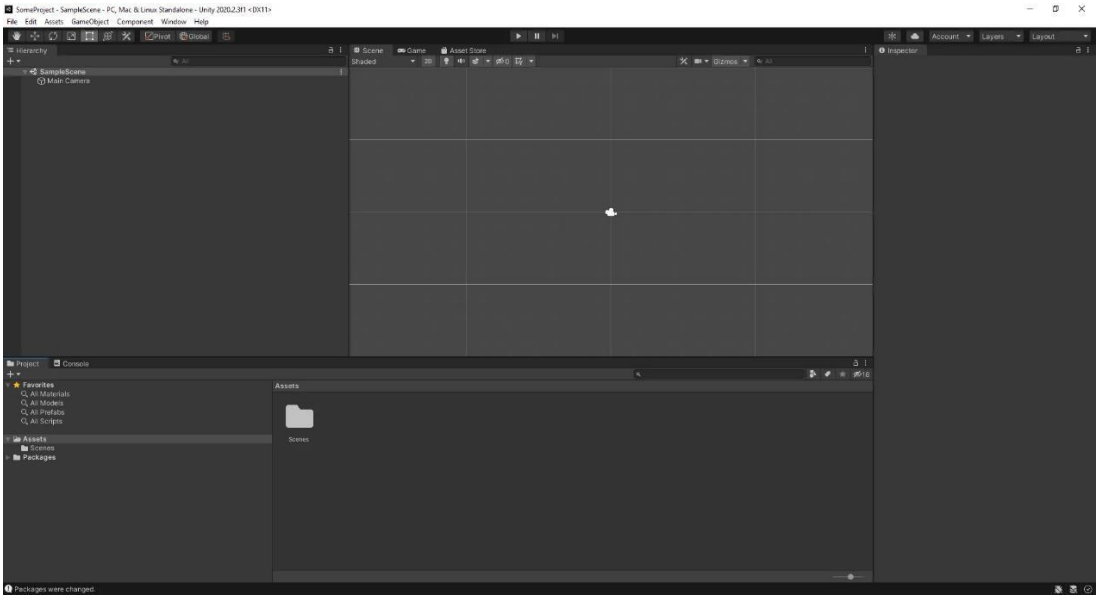

Рисунок 2.2 – Інтерфейс платформи «Unity»

Unreal Engine

Unreal Engine – це високопотужна платформа для створення ігор та візуальних проєктів, розроблена компанією Epic Games. Спочатку представлена у 1998 році як рушій для шутерів від першої особи, вона значно розширила свою функціональність і зараз широко використовується не тільки в ігровій індустрії, але й для архітектурної візуалізації, кінематографії та віртуальної реальності. Unreal Engine пропонує розробникам набір передових інструментів для роботи з фізикою, анімацією, освітленням та штучним інтелектом, що дозволяє створювати ігри з високим рівнем деталізації та реалістичними світами [11].

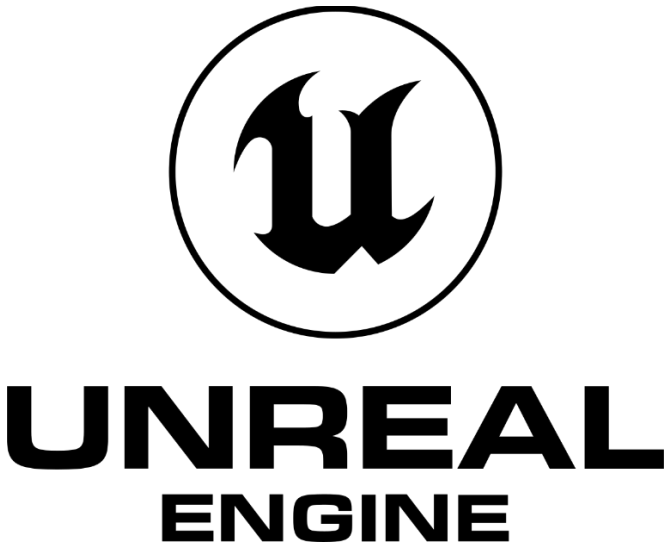

Рисунок 2.3 – Логотип «Unreal Engine»

Основною мовою програмування в Unreal Engine є C++, що забезпечує високий рівень контролю та оптимізації. Крім того, платформа підтримує використання скриптових мов, таких як Blueprint (візуальне програмування), Python та Lua, що робить її доступнішою для широкого кола розробників. Важливою особливістю Unreal Engine є підтримка рендерингу в режимі реального часу, що дозволяє досягати високоякісних візуальних ефектів.

Unreal Engine також має власний маркетплейс, де можна придбати готові моделі, текстури, звукові ефекти та інші ресурси для розробки ігор, що значно полегшує роботу з проєктами.

Орієнтація на графіку робить Unreal Engine більш складним та ресурсозатратним інструментом. Процес розробки вимагає значних технічних знань та потужних комп’ютерних ресурсів, що може уповільнити створення ігор середньої складності, таких як RTS. Для жанру стратегій, де основна увага приділяється управлінню ресурсами та ШІ, а не реалістичній графіці, такі можливості можуть бути зайвими. Крім того, розробка на Unreal Engine може бути менш зручною для індивідуальних розробників або малих команд через складність процесу програмування та високу криву навчання.

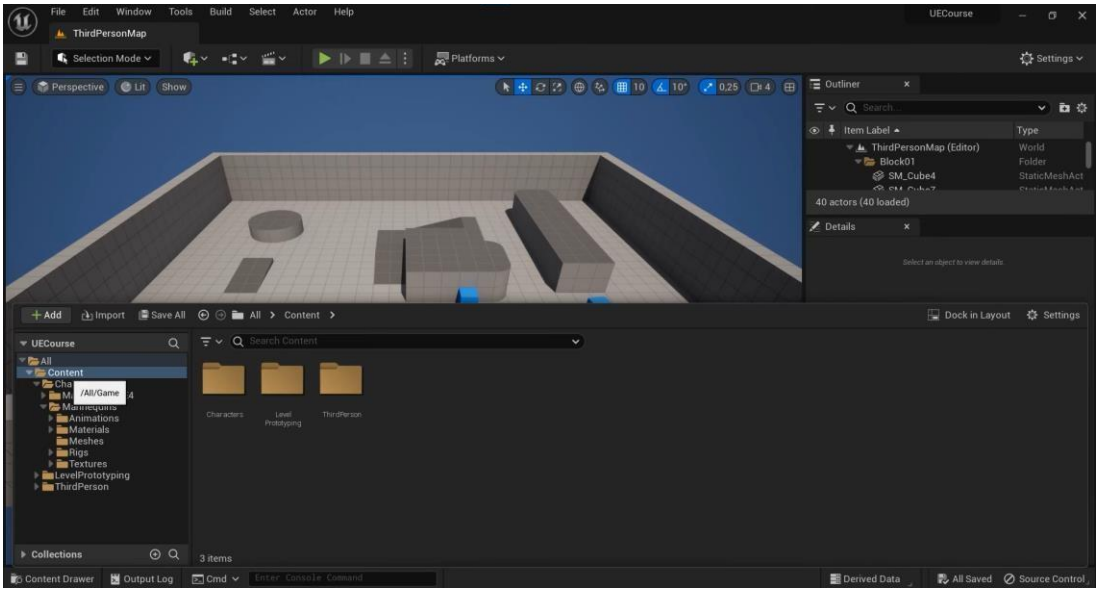

Рисунок 2.4 – Інтерфейс платформи «Unreal Engine»

CryEngine

CryEngine – це високоякісний ігровий рушій, розроблений компанією Crytek, який відзначається своїми можливостями у фотореалістичній графіці та фізичних симуляціях. CryEngine широко застосовується не тільки для розробки ігор, але й для архітектурної та промислової візуалізації. Основні його функції включають потужну систему рендерингу, реалістичну симуляцію фізики та підтримку технологій віртуальної реальності. Цей рушій дозволяє створювати деталізовані світи з високою точністю відображення освітлення, тіней та інших графічних ефектів [12].

Рисунок 2.5 – Логотип «CryEngine»

CryEngine підтримує розробку ігор для різних платформ, включаючи ПК, консолі та мобільні пристрої. Проте, у порівнянні з іншими рушіями, він може бути складнішим у використанні, що робить його менш доступним для початківців. Крім того, CryEngine вимагає високих апаратних ресурсів як від розробників, так і від кінцевих користувачів, що обмежує його використання у випадках, коли важливою є оптимізація під менш потужні пристрої.

Ще одним важливим аспектом є менша активність спільноти розробників навколо CryEngine в порівнянні з Unity або Unreal Engine, що може створювати певні труднощі при пошуку допомоги чи готових рішень.

Як і Unreal Engine, CryEngine має високу криву навчання та вимагає значних технічних знань для повного використання його можливостей. Для розробки RTS-ігор, де основними елементами є геймплей та інтелектуальна поведінка ворогів, CryEngine може бути надмірним через свою орієнтованість на фотореалістичність. Менша популярність CryEngine серед розробників порівняно з Unity або Unreal також означає обмежену доступність ресурсів для навчання та підтримки.

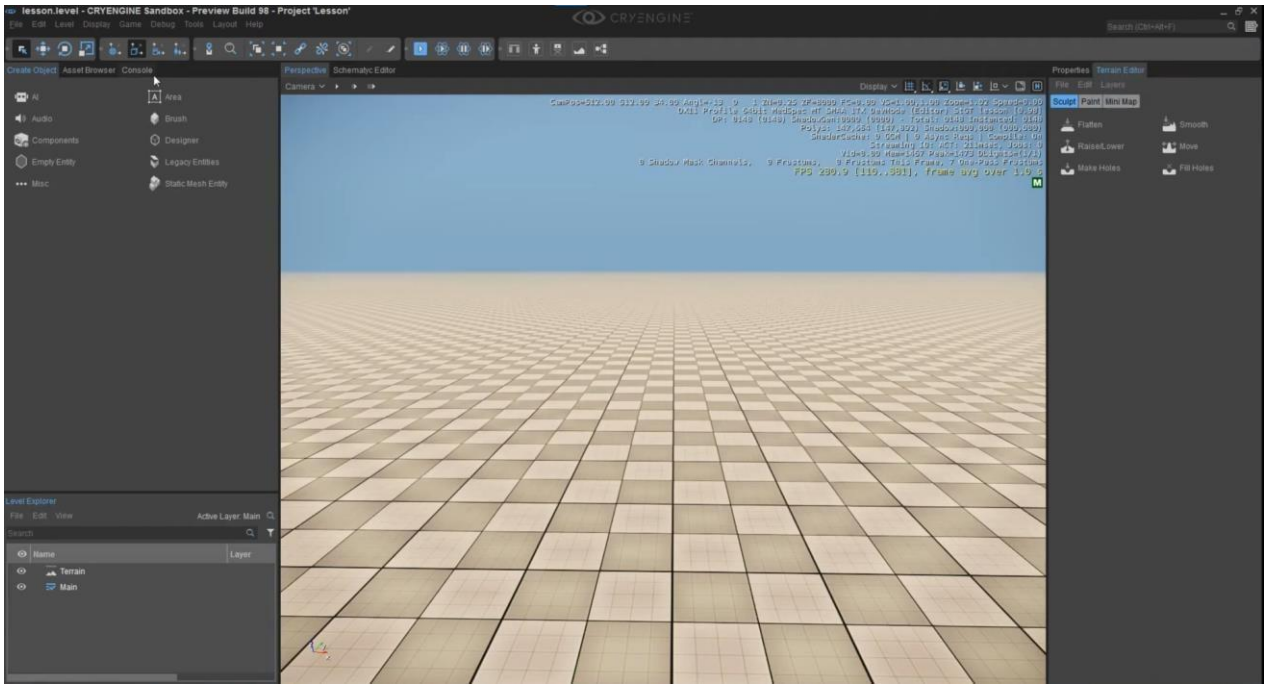

Рисунок 2.6 – Інтерфейс платформи «CryEngine»

Таблиця переваг та недоліків для порівняння платформ розробки (табл. 2.1-2.3): Unity, Unreal Engine, CryEngine.

Таблиця 2.1 – Переваги та недоліки Unity [8].

№	**Unity**	
	Переваги	*Недоліки*
1	Простий та інтуїтивний інтерфейс	Обмежені можливості для розробки графіки, залежно від версії
2	Велика спільнота користувачів та безкоштовні ресурси, що допомагають в навчанні та розробці проєктів	Обмежена можливість контролювання рендеринга та обробки даних
3	Широке застосування і підтримка різних платформ	Іноді можуть виникати проблеми з оптимізацією та продуктивністю
4	Безкоштовний	

Таблиця 2.2 – Переваги та недоліки Unreal Engine [11].

№	Unreal Engine	
	Переваги	*Недоліки*
1	Висока якість графіки та візуальних ефектів	Висока вартість підписки на платформу
2	Широкі можливості для розробки ігор, включаючи VR та AR проєкти	Складніше вивчення та розробка порівняно з Unity

Кінець таблиці 2.2

3	Є можливість використовувати скриптові мови, такі як C++, Python, а також власні мови програмування	Через свої потужні особливості до графіки, має великі вимоги до обладнання
		Потребує великого досвіду у розумінні C++

Таблиця 2.3 – Переваги та недоліки CryEngine [12].

№	CryEngine	
	Переваги	*Недоліки*
1	CryEngine має досить зручний та потужний інтерфейс для розробників, що дозволяє швидко створювати та редагувати ігрові об’єкти	CryEngine вимагає від розробника високого рівня знань та навичок в програмуванні та редагуванні ігрових об’єктів
2	CryEngine надає готові рішення для створення ігрових механік, таких як фотореалізм та розумне освітлення	Для роботи з CryEngine потрібна потужна комп’ютерна система, що може бути проблемою для деяких користувачів
3	CryEngine має вражаючу графіку, яка дозволяє створювати дуже деталізовані ігрові світи	У порівнянні з Unity та Unreal Engine, спільнота CryEngine є досить маленькою, що може ускладнити пошук допомоги

Кінець таблиці 2.3

4	Може працювати з дуже великими ігровими світами, що дозволяє створювати масштабні проєкти	CryEngine підтримує лише деякі платформи, такі як Windows та Xbox, що може бути обмеженням для деяких розробників

Unity обрана як найбільш підходяща платформа для розробки RTS-ігри завдяки своїй простоті, гнучкості, можливостям для реалізації штучного інтелекту та легкій інтеграції на різні платформи. Unreal Engine та CryEngine, попри свої потужні інструменти для графіки та фізики, є більш складними у використанні та вимогливими до ресурсів, що робить їх менш придатними для розробки проєкту у жанрі RTS.

2.2 Вибір мови програмування для розробки

У сфері розробки ігор на платформі Unity, особливо з елементами штучного інтелекту (ШІ), найбільш поширеними мовами програмування є C#, C++ та Java. Кожна з цих мов має свої унікальні особливості та переваги, і вибір конкретної мови залежить від вимог проєкту, технічних характеристик гри та навичок розробника. Наприклад, C++ зазвичай використовується для створення високопродуктивних ігор з великою кількістю графічних об’єктів та інтенсивними обчисленнями, особливо коли проєкт вимагає максимальної ефективності. Java часто застосовується завдяки своїй мультиплатформенній підтримці, але для розробки на Unity зазвичай використовують C# [9].

Зважаючи на тісну інтеграцію Unity з мовою програмування C#, для реалізації штучного інтелекту в 3D грі була обрана саме ця мова. C# є однією з найпопулярніших мов серед Unity-розробників завдяки її простоті, високому рівню абстракції та зручному синтаксису, що дозволяє швидко та ефективно створювати складні ігрові механіки та елементи ШІ.

Однією з ключових переваг C# є її зручність у створенні графічних інтерфейсів та інтеграції з візуальними ефектами та анімацією. Це дозволяє

створювати насичені 3D світи та реалістичних персонажів зі складною поведінкою, що є важливим для ігор у жанрі RTS з ШІ. Крім того, C# є частиною платформи .NET, що відкриває доступ до багатих бібліотек та фреймворків для роботи з різними аспектами розробки програмного забезпечення, що значно прискорює розробку ігор.

Одна з переваг мови C# полягає в підтримці багатопотоковості, що дає змогу виконувати складні обчислення паралельно, підвищуючи продуктивність гри. Це особливо важливо для ігор, у яких одночасно взаємодіє багато ігрових об’єктів, таких як юніти, які управляються штучним інтелектом у реальному часі. ШІ може обробляти дії кожного персонажа одночасно, не знижуючи загальної швидкості гри. Завдяки цьому C# стає чудовим вибором для розробки ігор, де важлива складна взаємодія та оптимізація продуктивності.

Ще одна значуща перевага C# – високий рівень безпеки та стійкості до помилок завдяки статичному типізуванню та механізмам управління пам’яттю. Це дає змогу виявляти потенційні помилки ще на етапі компіляції та знижує ймовірність виникнення критичних багів під час виконання програми. Це особливо важливо для складних ігор із великою кількістю компонентів, особливо якщо йдеться про інтеграцію ШІ. Крім того, розробка на C# допомагає створювати більш безпечні програми, які менш схильні до атак або помилок, пов'язаних з неправильним управлінням пам’яттю.

Для розробки гри було обрано середовище програмування Visual Studio, яке надає потужний набір інструментів для написання коду, налагодження та тестування програм. Однією з головних переваг є інтеграція з Unity, що дозволяє легко переносити зміни між проєктом у Unity та кодом у Visual Studio. Завдяки цьому середовищу, розробник може швидко тестувати та виправляти помилки у коді, що сприяє більш плавному та швидкому процесу розробки [10].

Visual Studio підтримує синхронізацію з Unity, забезпечуючи автоматичне оновлення змін у коді без необхідності ручної інтеграції. Це значно спрощує роботу з великими проєктами та дозволяє швидко створювати нові компоненти гри, такі як шейдери, механіки для штучного інтелекту або інші елементи, які

безпосередньо впливають на ігровий процес. Крім того, Visual Studio надає інструменти для ефективного управління проєктом, включно з тестуванням і налагодженням, що допомагає розробникам оперативно виявляти й виправляти помилки.

Рисунок 2.7 – Логотип «Visual Studio»

Таким чином, використання мови програмування C# у поєднанні з середовищем Visual Studio є оптимальним вибором для розробки 3D-ігор на Unity із використанням штучного інтелекту. Це рішення забезпечує зручність у роботі, високу продуктивність, безпеку та можливість кросплатформної розробки, що робить його популярним серед розробників сучасних ігор.

2.3 Вибір застосунку для моделювання

Візуальні елементи відіграють ключову роль у створенні комп'ютерних 3D-ігор, особливо на платформі Unity. Все, що бачить гравець – від персонажів до оточення, – є плодом креативної роботи розробників та дизайнерів. Кожен аспект візуалізації має значення, оскільки він визначає атмосферу гри та впливає на її сприйняття користувачами. Вибір програмного забезпечення для 3D-моделювання є однією з найбільш критично важливих рішень у процесі розробки, адже це визначає якість візуальних об’єктів та взаємодію з ігровим середовищем.

Для порівняння програм, що використовуються у 3D-моделюванні для створення ігор на Unity, розглянемо дві популярні платформи: Cinema 4D та Blender.

Cinema 4D

Cinema 4D, розроблене компанією Maxon, широко використовується для створення 3D-моделей, анімації та візуалізації. Це програмне забезпечення знайшло застосування не лише в анімаційних фільмах і рекламних роликах, але й у комп'ютерних іграх, зокрема у 3D-проєктах на Unity. Cinema 4D дозволяє розробникам створювати високоякісні візуальні ефекти та моделі, які легко інтегруються в Unity для реалізації ігрових сцен і персонажів [13].

Рисунок 2.8 – Логотип «Cinema 4D»

Однією з найбільших переваг Cinema 4D є її зручний і зрозумілий інтерфейс, який робить програму доступною як для новачків, так і для досвідчених розробників. Cinema 4D підтримує численні формати файлів, що дозволяє легко передавати створені моделі до інших програм для подальшої обробки або інтеграції в Unity. Це забезпечує високу гнучкість у розробці комп'ютерних 3D-ігор, оскільки дозволяє ефективно використовувати наявні ресурси та моделі. Додатково, програма включає вбудовану бібліотеку готових 3D-моделей і матеріалів, що значно пришвидшує процес створення ігрового середовища.

Cinema 4D пропонує широкі можливості для створення анімацій. Ця програма включає інструменти, такі як ієрархічна структура об'єктів та скелетна анімація, що дозволяють детально налаштовувати рухи персонажів та інших об'єктів у грі. Такі функції можна використовувати для розробки анімації ігрових персонажів у Unity, враховуючи складну динаміку, взаємодії та рухи, які можуть

керуватися скриптами або штучним інтелектом. Програма також підтримує створення тривимірних текстових елементів та заголовків, що може бути корисно при розробці ігрового інтерфейсу.

Інша значуща особливість Cinema 4D – її потужний рендеринговий механізм, здатний створювати дуже реалістичні сцени та зображення. Це особливо актуально при розробці 3D-ігор на Unity, де якість візуального контенту є критично важливою для створення захоплюючої атмосфери. Cinema 4D підтримує як власний рендеринговий двигун, так і зовнішні рендерери, такі як Arnold, Octane та Redshift, що дозволяє користувачам обирати найбільш підходящу технологію для своїх проєктів. Під час рендерингу ігрових сцен можна використовувати передові методи освітлення та текстурування, щоб досягти максимальної реалістичності.

Серед ключових особливостей Cinema 4D, які відрізняють її від інших програм, є система MoGraph. Це інструмент для створення складних анімаційних графіків та динамічних візуальних ефектів, які можуть бути використані для візуалізації взаємодій у грі або динамічної поведінки об'єктів. Наприклад, розробники можуть створювати складні ефекти, що змінюються у відповідь на дії гравця або події в грі, значно підвищуючи рівень інтерактивності ігрового середовища.

Cinema 4D також інтегрується з іншими програмами, такими як Adobe After Effects та Adobe Illustrator, що відкриває додаткові можливості для створення комплексних проєктів. Ця інтеграція дозволяє легко переносити створені моделі та анімації до інших програм для додаткової обробки або використання у відео та рекламних роликах для просування гри. Крім того, підтримка широкого спектру форматів файлів дозволяє імпортувати та експортувати активи між Cinema 4D та іншими популярними 3D-редакторами, що є важливою перевагою для розробників ігор на Unity.

Отже, Cinema 4D є потужним інструментом для розробки 3D-ігор на Unity, зокрема завдяки своїм можливостям у сфері моделювання, анімації та інтеграції з іншими програмами.

Blender

Blender – це потужна безкоштовна програма для 3D-моделювання, анімації та візуалізації, яка розвивається завдяки зусиллям відкритої спільноти ентузіастів і професіоналів. Регулярні оновлення програми гарантують підтримку новітніх технологій та інструментів, що є важливою перевагою для розробників 3D-ігор на платформі Unity. Завдяки цьому, Blender стає невід’ємною частиною інструментарію для створення високоякісних візуальних елементів у ігрових проєктах [14].

Рисунок 2.9 – Логотип «Blender»

Однією з найбільших переваг Blender є зручний та ергономічний інтерфейс, що спрощує процес навчання та використання програми. Це особливо важливо для розробників ігор на Unity, яким потрібна швидкість і гнучкість при створенні візуального контенту. Крім того, Blender підтримує розширення на основі Python, що дозволяє створювати власні скрипти та інструменти для автоматизації певних процесів, таких як генерація процедурних об’єктів або інтеграція ШІ для автоматичного налаштування сцен. Це робить Blender дуже зручним для створення складних проєктів з використанням Unity.

Blender має великий набір інструментів для моделювання, включаючи можливість створення базових форм (примітивів) та роботи з мешами. Це дозволяє розробникам створювати високодеталізовані моделі для ігрових середовищ і персонажів. Крім того, в Blender доступні інструменти для анімації з використанням ключових кадрів і кривих руху, що дає можливість створювати

динамічні сцени, які можна інтегрувати в Unity. Наприклад, такі функції будуть особливо корисні для створення персонажів, які керуються штучним інтелектом, або для анімації взаємодій між об’єктами у грі. Blender також має інструменти для візуалізації даних, що дозволяють легко створювати графіки та діаграми для внутрішніх потреб.

Blender пропонує широкий набір інструментів для текстурування та налаштування освітлення, що дозволяє створювати реалістичні сцени та об'єкти. Особливо варто відзначити можливості фізичних симуляцій, таких як моделювання рідин, твердих і м'яких тіл, що допомагає створювати природні рухи в грі та взаємодію з оточенням у реальному часі на платформі Unity. Це робить Blender ідеальним для проєктів, де важлива симуляція фізичних явищ і реалістична поведінка об'єктів.

Blender також підтримує створення контенту для віртуальної реальності (VR), що робить його цінним інструментом для проєктів Unity, орієнтованих на VR-ігри. Це дозволяє розробникам створювати інтерактивні середовища та персонажів, оптимізованих для VR-технологій. Крім того, Blender підтримує різні формати файлів, що полегшує інтеграцію з іншими програмами для 3D-моделювання та візуалізації, що дозволяє безперешкодно переносити моделі та анімації між різними програмами чи інтегрувати їх у Unity.

На основі порівняння Blender і Cinema 4D можна зробити висновок, що Blender є привабливим вибором для розробників комп'ютерних 3D-ігор на Unity, які шукають безкоштовний і функціонально потужний редактор з широкими можливостями. Blender відзначається своєю універсальністю: підтримка різних платформ і форматів файлів дозволяє легко експортувати створені моделі та анімації до Unity. Важливо, що Blender має відкритий вихідний код, що дозволяє розробникам розширювати його функціональність, створювати власні плагіни та інструменти, оптимізуючи процес розробки. Крім того, тісна співпраця Unity з розробниками Blender може вказувати на майбутню глибшу інтеграцію між цими платформами. Тому для створення комп'ютерних 3D-ігор на Unity Blender є логічним і ефективним вибором.

Висновки до розділу 2

У процесі роботи над другим розділом було опрацьовано інформацію про різні види ігрових рушіїв, проведено аналіз основних відмінностей між сучасними еталонами інструментів для розробки ігор, а також детально вивчено програми для створення 3D-простору та моделей для проєкту.

В якості основного ігрового рушія для розробки 3D-гри було обрано Unity. Основні переваги Unity полягають у зручності використання, великій кількості доступних готових компонентів та ресурсів, а також підтримці багатьох платформ що робить його універсальним вибором для розробки ігор.

Мовою програмування для проєкту було обрано C#, яка є однією з найпопулярніших мов для розробки ігор. Завдяки її широким можливостям для створення графічних інтерфейсів та інтеграції з Unity, вона дозволяє швидко створювати ігри з високоякісною графікою, візуальними ефектами та плавною анімацією.

Що стосується вибору програмного забезпечення для 3D-моделювання, то перевага була надана Blender. Ця програма є відмінним варіантом для студій та індивідуальних розробників, оскільки пропонує широкий функціонал, є безкоштовною та активно підтримується великою спільнотою користувачів, що сприяє її постійному розвитку та удосконаленню.

3 МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ ІГРОВОГО ЗАСТОСУНКУ

У проєктуванні використано мову графічного моделювання UML (Unified Modeling Language), яка є універсальним інструментом для візуалізації та створення моделей об’єктів програмного забезпечення. UML-діаграми мають важливе значення під час створення проєктів програмного забезпечення, оскільки дозволяють детально описати логіку роботи системи, виявляти можливі помилки на ранніх етапах розробки та підвищувати ефективність процесу.

Завдяки своїй універсальності UML широко використовується фахівцями в IT-галузі: програмістами, аналітиками, менеджерами проєктів і навіть власниками компаній, які прагнуть зрозуміти структуру майбутніх рішень. Хоча UML не є мовою програмування, її моделі можуть слугувати основою для автоматичної генерації коду. Для побудови UML-діаграм у цьому проєкті застосовано програмне забезпечення StarUML, яке забезпечує інтуїтивно зрозумілий інтерфейс і потужні засоби моделювання.

3.1 Написання usecase

Use Case — це текстовий опис різних сценаріїв взаємодії користувача із системою, які можуть закінчитися як успішно, так і невдало, залежно від виконання певних цілей. Сценарій являє собою послідовність дій, які виконує користувач для реалізації конкретних операцій у системі [15].

Існують три основні форми написання:

− Коротка форма – це стислий опис одного зі сценаріїв, зазвичай успішного, у вигляді одного абзацу. Використовується на етапі початкового аналізу вимог до системи.

− Поверхнева форма – це детальніший опис всіх сценаріїв у вільній формі (основного та альтернативних) одного з варіантів використання. Використовується на етапі первинного аналізу вимог до системи.

− Повна форма – це детальний опис всіх кроків та дій, включаючи попередні та постумови виконання use case. Використовується на етапі вибору з

повного списку варіантів використання важливих (критично для роботи системи).

Короткий use case

Користувач має будувати власну економіку та збройні сили, щоб атакувати базу противника й досягти перемоги. Важливим елементом гри є ефективне управління ресурсами, що видобуваються робітниками. Користувач отримує контроль над єдиним доступним рівнем і має виконувати всі дії, спираючись на механіку реального часу. Успіх залежить від вміння грамотно розподіляти ресурси та ухвалювати стратегічні рішення.

Поверхневий use case

Користувач керує робітниками для видобутку ресурсів, будує й покращує споруди та військові підрозділи, а також розробляє стратегії для атаки бази противника. Всі дії відбуваються в межах одного рівня, і користувач змагається з штучним інтелектом. Гра має офлайн-режим, що дозволяє насолоджуватися нею без необхідності підключення до інтернету. Для досягнення перемоги користувач повинен знищити базу суперника, ефективно використовуючи доступні ресурси.

Альтернативний сценарій:

1) Користувач не зміг захистити свою базу й програв матч.
2) Користувач збудував недостатньо сильну армію, що не дозволило успішно атакувати базу противника.
3) Користувач не використав ресурси ефективно, що призвело до сповільнення темпів розвитку та поразки.
4) Користувач припустився помилки у стратегії, через що програв.

Таблиця 3.1 – Повний use case [15]

Primary Actor	Гравець
Scope	Комп’ютерна RTS-гра на базі Unity з використанням ШІ
Level	Стратегічна перемога над противником
Preconditions	Гравець запустив гру
Stakeholders and interests	1. програміст: зацікавлений у виконанні своєї задачі та у стабільності роботи своєї розробки; 2. тестувальник: зацікавлений у знаходженні помилок з подальшим їх усуненням програмістом для бездоганного ігрового процесу; 3. гравець: зацікавлений у досягненні перемоги, удосконалюючи стратегії та майстерність.
Main Success Scenario: 1. гравець встановлює гру; 2. гравець запускає гру; 3. гравець аналізує ситуацію на карті та обирає оптимальну стратегію розвитку; 4. гравець керує робітниками для збору ресурсів; 5. покращує структури й війська, враховуючи наявність ресурсів і дій ШІ; 6. гравець атакує ворожу базу, використовуючи обрані стратегії; 7. гравець досягає перемоги шляхом повного знищення ворожої бази.	

Продовження таблиці 3.1

Result	Гравець отримує задоволення від перемоги й покращує власні стратегічні навички.
Extensions:	
1.	1. Гравець не задоволений ігровим процесом: 1.1. гравець може звернутися до розробників через зворотний зв’язок для подання пропозицій або скарг; 1.2. негативний досвід може бути пов’язаний із: 1.2a. низькою складністю гри; 1.2b. надмірною складністю штучного інтелекту; 1.2c. недостатньо інтуїтивним інтерфейсом або технічними обмеженнями; 1.2d. некоректна адаптація складності до прогресу гравця; 1.2e. відсутність певного контенту або очікуваних функцій.
2.	2. Тестувльник знаходить помилку у грі: 2.1 a. помилка не глобальна; 2.1 b. програміст її швидко виправляє; 2.2 a. помилка глобальна; 2.2b. потрібно знаходити та вирішувати помилку та зупиняти гру на технічне обслуговування.

Кінець таблиці 3.1

3.	3. Штучний інтелект поводиться неочікувано: 3.1. наприклад, ШІ виявляє надмірну агресивність або пасивність у певних ситуаціях; 3.2. гравець може повідомити про такі ситуації, що допоможе тестувальникам вдосконалити алгоритми; 3.3. розробник налаштовує баланс гри через оновлення.
Special Requirements	1) Система підтримки багатомовного інтерфейсу для полегшення адаптації гри до різних аудиторій. 2) Адаптивний інтерфейс, зручний для різних розмірів екранів та технічних параметрів. 3) Унікальні алгоритми штучного інтелекту, які забезпечують реалістичну поведінку противника.
Frequency of Occurrence	Усі функціональні процеси реалізуються без необхідності постійного підключення до інтернету. Гра розрахована на автономний режим роботи, що робить її доступною для широкого кола користувачів.

Після аналізу основних та альтернативних сценаріїв використання гри, наведена таблиця дозволяє чітко визначити всі можливі варіанти взаємодії користувача з ігровою системою. Це допомагає зрозуміти і врахувати всі аспекти поведінки користувача, що можуть вплинути на ігровий досвід.

Документування повного use case також сприяє виявленню потенційних проблем і вузьких місць на ранніх етапах розробки. Це забезпечує вищу якість кінцевого продукту і задоволення потреб користувачів. Визначення спеціальних

вимог, таких як багатомовна підтримка та адаптивний інтерфейс, гарантує, що гра буде доступною та привабливою для широкого кола гравців.

3.2 Створення діаграми використання

Діаграми варіантів використання (Use Case diagrams) є незамінним інструментом для опису взаємовідносин та залежностей між різними групами варіантів використання та дійових осіб, що беруть участь у процесі. Ці діаграми допомагають чітко візуалізувати, як кінцеві користувачі взаємодіють із системою, а також визначають основні функції та завдання системи з точки зору користувача. Важливо відзначити, що діаграми варіантів використання не розкривають внутрішньої архітектури системи, а зосереджуються на її зовнішній поведінці. Вони слугують для полегшення комунікації з користувачами та клієнтами, а також для визначення вимог до системи. Діаграми варіантів використання особливо корисні на етапах проектування та розробки, оскільки вони дозволяють уникнути непорозумінь і сприяють узгодженню вимог між усіма зацікавленими сторонами [16].

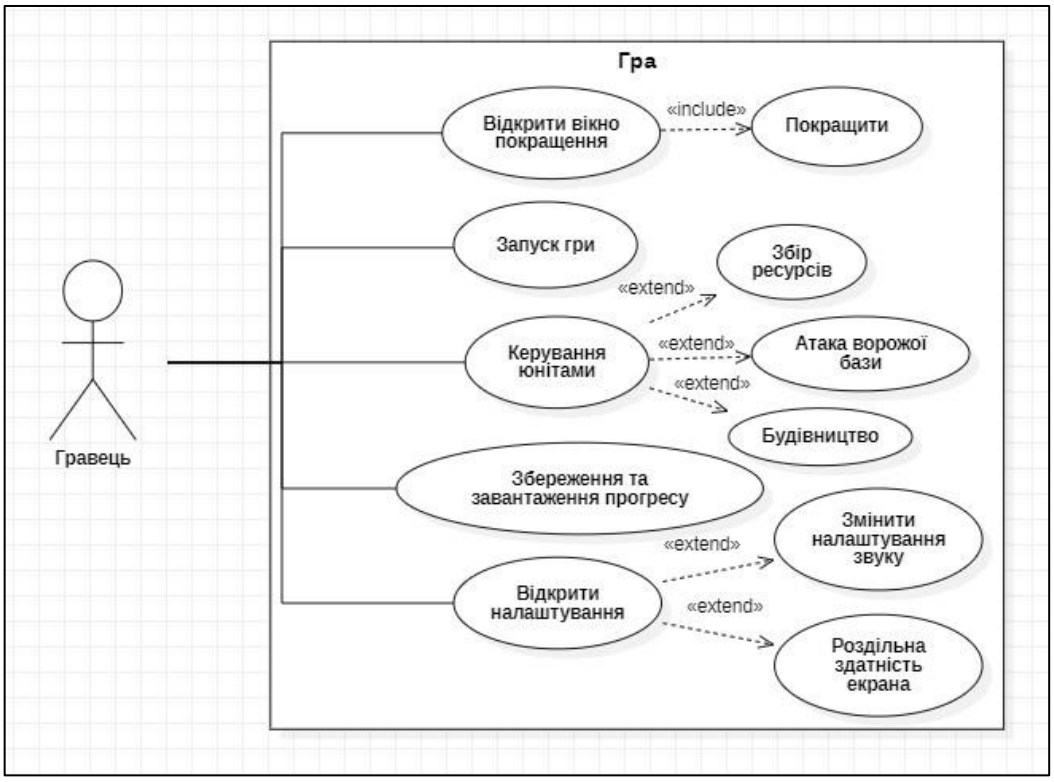

Рисунок 3.1 – Діаграма варіантів використання

Ця діаграма показує варіанти використання гри та взаємодію гравця з різними функціями системи. У центрі діаграми стоїть елемент "Гра", до якого пов’язані різні випадки використання, такі як "Відкрити вікно покращення", "Запустити гру", "Керувати юнітами", "Зберегти та завантажити прогрес", тощо. Діаграма демонструє, як різні функції взаємопов’язані і які дії може виконати гравець, обираючи різні опції. Додаткові зв’язки, позначені як «include» та «extend», вказують на обов’язкові та необов’язкові функції, що підкреслює гнучкість унавпаки з основним сценарієм. Ця діаграма є важливим інструментом, оскільки вона дає змогу чітко бачити, які функції має реалізувати програма для забезпечення інтерактивності та повноцінного користувацького досвіду.

Компоненти діаграми варіантів використання [17]

Діаграми варіантів використання включають чотири основні елементи: актор, варіант використання (прецедент), система та зв’язок.

Актор. Це об’єкт або особа, яка взаємодіє із системою, але не є її частиною (знаходиться поза її межами).

Варіант використання (прецедент). Відображає передбачувану поведінку системи та відповідає на запитання, що саме виконує система. Представлений у вигляді еліпса з дієсловом, що позначає дію. Прецедент фокусується на тому, що має відбутися, але не пояснює, як саме це станеться.

Система. Об’єкт моделювання, наприклад, вебсайт, мобільний додаток чи програмний модуль. Зображується прямокутником із назвою системи у верхній частині.

Зв’язок. Актори завжди повинні бути з’єднані з варіантами використання, хоча самі варіанти не обов’язково пов’язані з акторами.

У діаграмах розрізняють чотири типи зв’язків:

1. **Асоціація (Association).** Основний тип зв’язку між актором і варіантом використання, зображений суцільною лінією без напису.
2. **Розширення (Extend).** Вказує на додаткові функції або необов’язкові варіанти поведінки системи. Базовий варіант залишається самостійним і не

залежить від розширення. Позначається пунктирною стрілкою з написом <<extend>>, що вказує на основний варіант, і активується лише за певних умов.

3. Включення (Include). Демонструє, що певна поведінка одного варіанта є частиною іншого. Використовується для уникнення дублювання та забезпечення обов’язкової функціональності. Позначається пунктирною стрілкою з написом <<include>>.

4. Генералізація (Generalization). Відображає ієрархічні зв’язки між елементами, наприклад, між актором-нащадком і актором-предком. Нащадок успадковує всі властивості предка, але може мати унікальні особливості. Зображується суцільною лінією зі стрілкою у вигляді порожнього трикутника.

3.3 Побудова діаграм взаємодії (послідовності та кооперації)

Діаграми взаємодії є важливим інструментом у процесі моделювання програмних систем за допомогою UML (Unified Modeling Language). Вони дозволяють візуалізувати та деталізувати сценарії взаємодії між об'єктами, забезпечуючи розуміння динамічної поведінки системи. Два основні типи таких діаграм — діаграми послідовності та діаграми кооперації — використовуються для різних аспектів аналізу взаємодії [16].

- Діаграма кооперації показує структурні взаємозв'язки між об'єктами.
- Діаграма послідовності ілюструє порядок взаємодій у часі.

Коли потрібно деталізувати зв'язки між об'єктами, доцільно використовувати діаграми кооперації. Для аналізу порядку виконання дій найкраще підходять діаграми послідовності.

1) Діаграми послідовності (Sequence Diagrams):

Діаграми послідовності дозволяють відобразити динамічні аспекти системи, зокрема часовий порядок обміну повідомленнями між об'єктами. Вони ілюструють сценарії, у яких об'єкти взаємодіють, акцентуючи увагу на послідовності дій.

На відміну від діаграм кооперації, діаграми послідовності забезпечують точне відображення хронологічного порядку взаємодій. Наприклад, у грі RTS

можна продемонструвати, як гравець створює військову одиницю через інтерфейс, система обробляє цей запит, а створений NPC починає виконувати завдання.

Особливості діаграми послідовності:

- Відображає часовий порядок передачі повідомлень.
- Підкреслює взаємодію об'єктів під час виконання сценарію.

Ключові елементи [17]:

- об’єкти, що беруть участь у сценарії, представлені у вигляді горизонтальних ліній життя;
- повідомлення, передані між об’єктами, представлені стрілками;
- фокус управління, що позначає період активності об’єкта, коли він виконує певну дію.

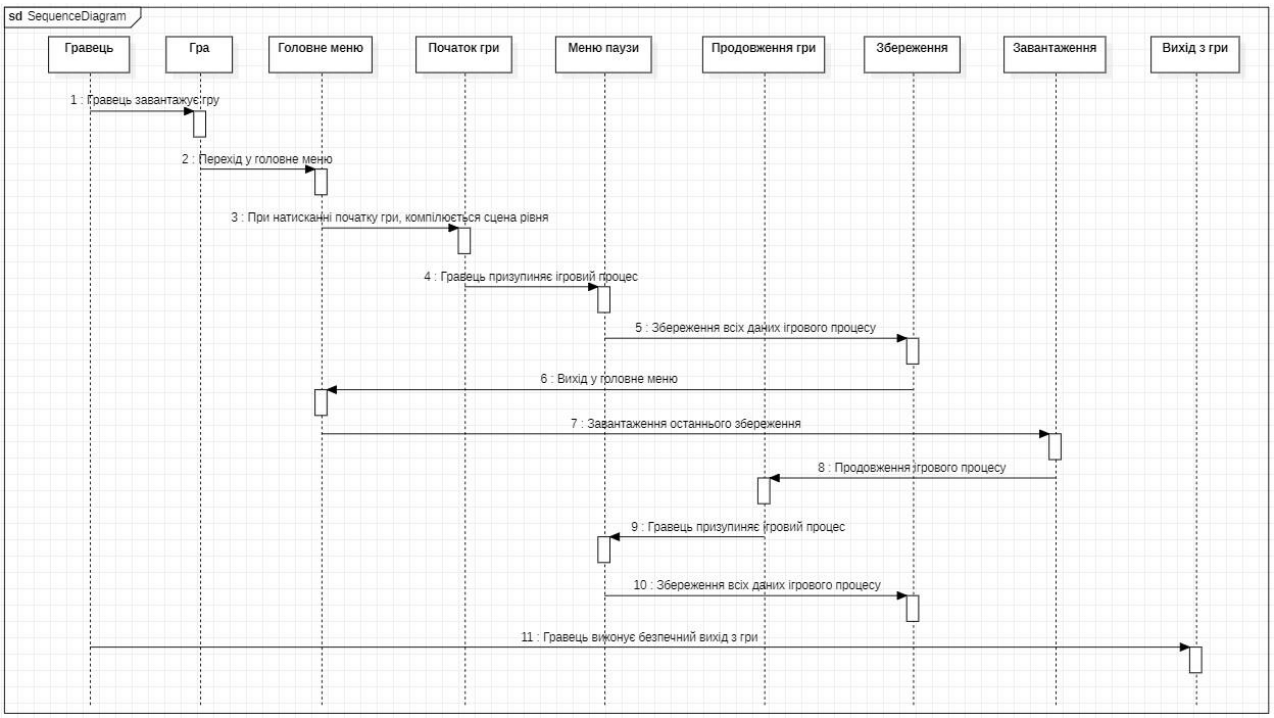

Рисунок 3.2 – Діаграма послідовності

Ця діаграма показує послідовність взаємодій між різними об’єктами та компонентами системи в процесі виконання конкретного сценарію. Діаграма ілюструє, як гравець взаємодіє з елементами інтерфейсу, такими як головне меню, рівні гри, збереження та вихід. У ній зображено послідовність повідомлень, яка передається між об’єктами, починаючи з завантаження гри та

переходу до головного меню. Кожен крок відображає важливу дію: наприклад, обробка натискання кнопки для початку гри або переходу до меню паузи. Важливо зазначити, що діаграма показує кілька умовних переходів, які ведуть до різних сценаріїв, підкреслюючи важливість кожної взаємодії в контексті загального процесу.

2) Діаграми кооперації (Collaboration Diagrams):

Діаграми кооперації демонструють структурні взаємозв’язки між об'єктами системи, які беруть участь у виконанні спільного завдання. Вони показують, як об'єкти співпрацюють, обмінюючись повідомленнями для досягнення певної мети.

Цей тип діаграм дозволяє описати часовий "зріз" взаємодій між об’єктами для досягнення бізнес-цілі системи. Наприклад, у контексті комп’ютерної 3D-гри RTS на Unity, діаграма кооперації може відображати процес взаємодії між гравцем, NPC і системою управління ресурсами, коли гравець віддає наказ збирати ресурси або будувати структури.

Ключові елементи діаграми кооперації [17]:

– екземпляри об’єктів, що беруть участь у сценарії (акторів і класів);
– зв’язки між об’єктами, що визначають їхню асоціацію;
– повідомлення, якими обмінюються об’єкти для виконання завдання.

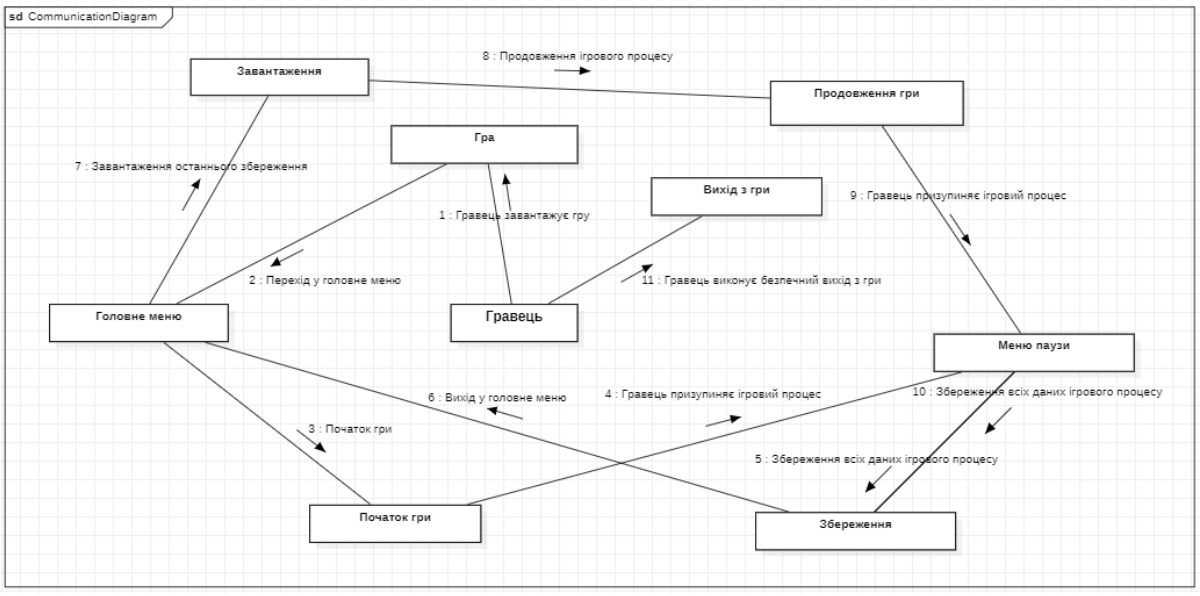

Рисунок 3.3 – Діаграма кооперації

Ця діаграма зосереджується на взаємозв’язках між об’єктами системи, зокрема в контексті кооперативних дій у грі. Вона демонструє, як різні компоненти, такі як головне меню, гра, збереження й вихід, співпрацюють для виконання основних функцій. Кожен об’єкт представлений у вигляді блоку, а стрілки між ними показують, як передаються повідомлення і здійснюється взаємодія. Це дозволяє візуалізувати загальну архітектуру програми, демонструючи, які елементи взаємодіють один з одним та на якому етапі відбуваються певні дії. Діаграма підкреслює ключові рольові функції окремих елементів системи, вказуючи на їх обов’язки, що допомагає зрозуміти загальну структуру і логіку роботи гри.

3.4 Діаграми станів та переходів

Діаграми станів (state diagrams) є графічним засобом UML, який використовується для моделювання поведінки об’єктів або систем протягом їхнього життєвого циклу. Ці діаграми ілюструють послідовність станів, у яких може перебувати об’єкт, а також події, що спричиняють переходи між ними. Вони є особливо корисними для моделювання реактивних об’єктів, стан яких змінюється залежно від зовнішніх чи внутрішніх впливів [16].

Діаграми станів дозволяють розробникам вивчити динамічні аспекти системи, зрозуміти, як її компоненти реагують на певні події, і створити чіткий опис поведінкової моделі. Вони використовуються для аналізу поведінки об’єктів, сценаріїв використання, операцій або функціональних елементів системи.

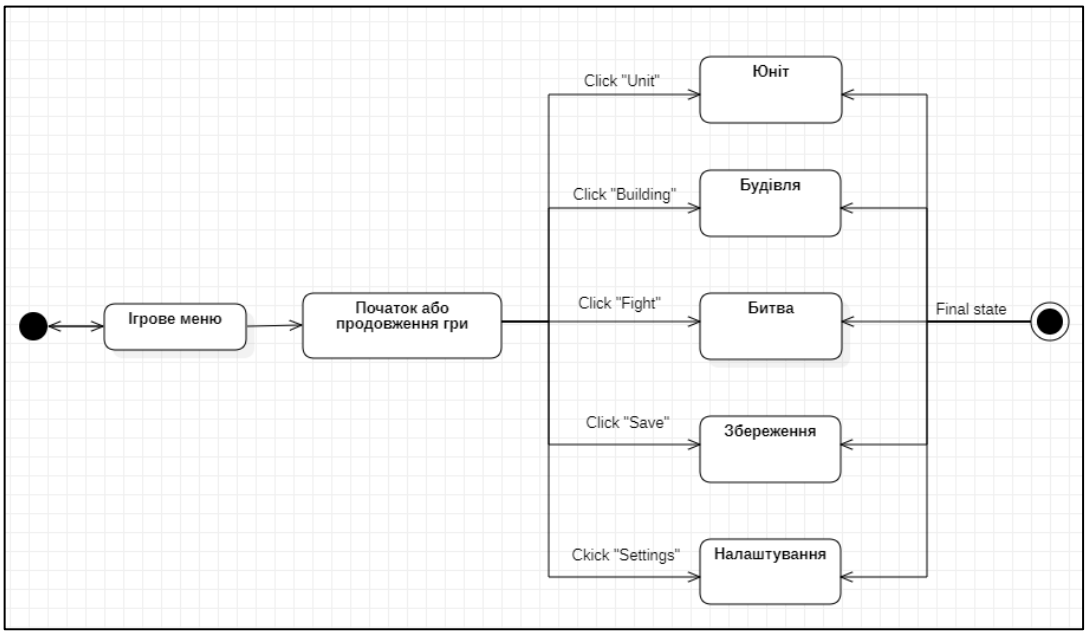

Рисунок 3.4 – Діаграма станів для всієї гри

Ця діаграма демонструє різні стани, в яких може перебувати гра, а також переходи між цими станами. Вона забезпечує чітке розуміння того, що відбувається у грі на різних етапах, таких як початковий екран, активна гра, пауза, закінчення гри тощо. Кожен стан представлений у вигляді блоку, а стрілки вказують на можливі переходи, викликані діями користувача або внутрішніми умовами гри.

Основні компоненти діаграм станів та переходів [17]

1. **Стан (State):** Стан відображає певну ситуацію, у якій перебуває об’єкт упродовж деякого часу. Він визначається діями, які виконує об’єкт, або очікуванням на подію.

2. **Подія (Event):** Подія є описом важливої зміни чи дії, що ініціює перехід об’єкта з одного стану до іншого. Це може бути взаємодія користувача, зміна параметрів середовища або сигнал від іншого компонента.

3. **Перехід (Transition):** Перехід визначає зв’язок між станами. Він показує, як об’єкт переходить від початкового до цільового стану внаслідок певної події, якщо виконуються задані умови.

4. Дія (Action): Дія представляє невелике обчислення чи операцію, яка виконується під час переходу. Наприклад, це може бути зміна атрибутів об’єкта або передача повідомлення.

5. Діяльність (Activity): Діяльність описує довготривалу операцію, яка відбувається, доки об’єкт залишається в певному стані. Це можуть бути обчислювальні процеси або взаємодії з іншими компонентами.

6. Початковий та кінцевий стани (Initial and Final States): Початковий стан визначає точку, з якої стартує життєвий цикл об’єкта, тоді як кінцевий стан позначає його завершення.

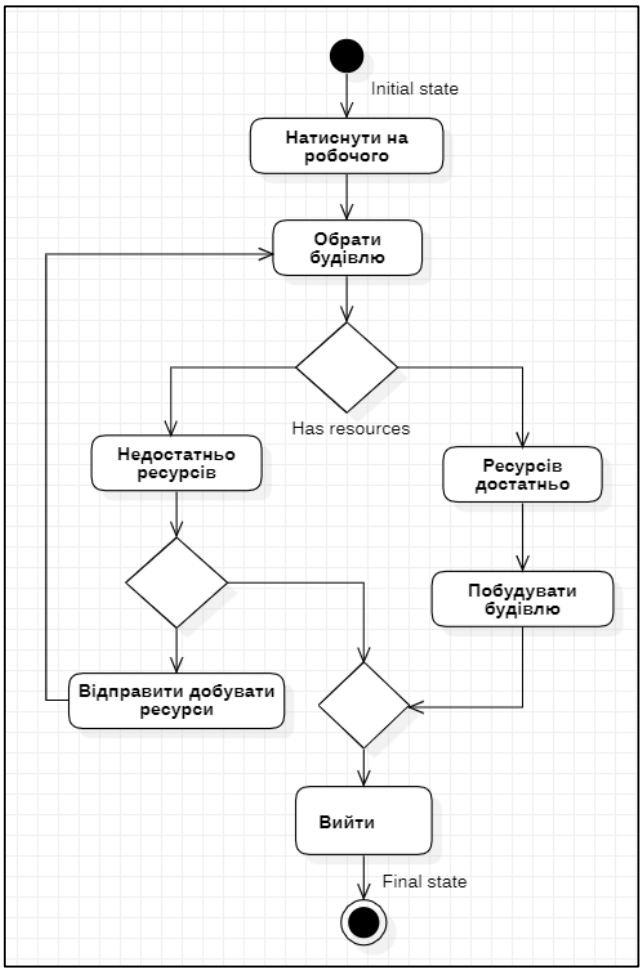

Рисунок 3.5 – Діаграма станів для будівництва

У цій діаграмі детально описано стани та переходи, пов’язані з процесом будівництва в грі. Діаграма показує, які дії повинен виконати гравець, щоб здійснити будівництво, починаючи від вибору будівлі до завершення її зведення. Вона демонструє логіку, за якою гравець може спочатку обрати рольового

персонажа, а потім ініціювати процес будівництва, після чого система перевіряє наявність необхідних ресурсів. Якщо ресурсів не вистачає, гра дозволяє гравцеві повернутися до ресурсоотримуючих дій або скасувати будівництво. Ця діаграма є важливим елементом, оскільки вона допомагає точно визначити, які умови та дії мають місце у рамках будівельного процесу.

3.5 Діаграма діяльності

Діаграма діяльності (Activity Diagram) — це інструмент моделювання в UML, який дозволяє відобразити алгоритмічну поведінку системи. Вона показує послідовність виконання дій та їхні взаємозв’язки. На вигляд діаграма схожа на блок-схеми, однак її функціональність ширша — вона дає змогу моделювати як послідовні, так і паралельні процеси, включно з їхньою синхронізацією. Діаграма описує потік виконання дій, які можуть бути послідовними, альтернативними чи паралельними, і демонструє, як система чи її компоненти змінюють стан під впливом подій [16].

Цей тип діаграм ефективний для моделювання бізнес-процесів, поведінки програмних компонентів і складних операцій у програмних системах. Вони деталізують алгоритми, ілюструючи, як дані переміщуються між етапами, які дії виконуються, і як здійснюються переходи. Завдяки універсальності, діаграми діяльності застосовуються у проєктуванні процесів — від бізнес-моделей до складних програмних систем.

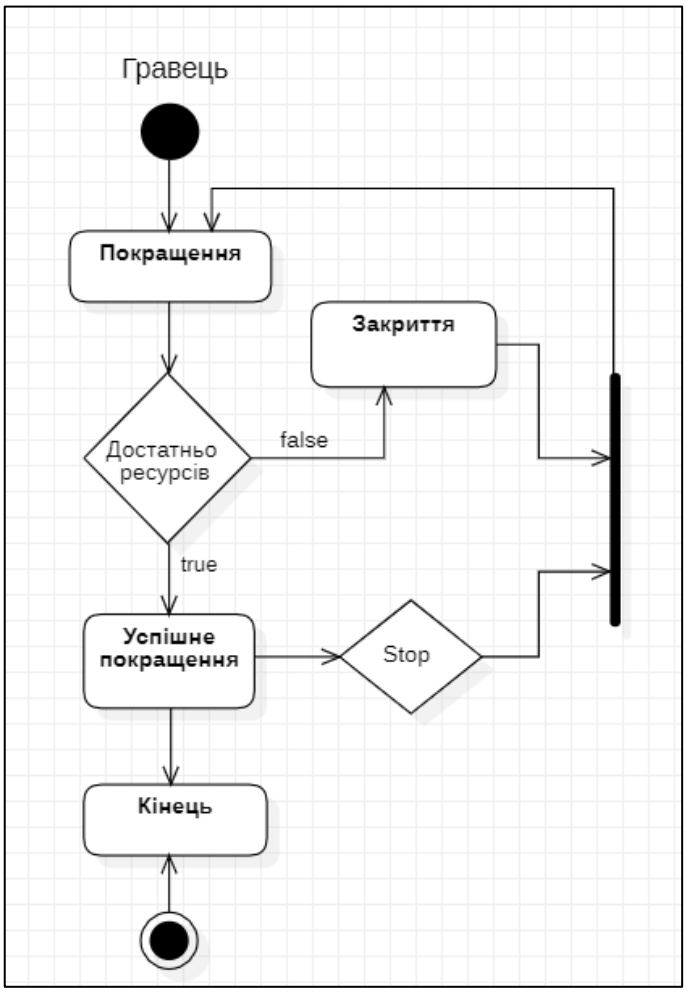

Рисунок 3.6 – Діаграма діяльності покращення

Ця діаграма описує процес покращення в контексті гри, зосереджуючи увагу на діях гравця та рішеннях, які приймаються на кожному етапі. Діаграма починається зі старту дії "Покращення", за яким слідує перевірка наявності достатніх ресурсів. Якщо ресурси наявні, відбувається успішне покращення, і гра переходить до основного етапу завершення цієї дії. У випадку, якщо ресурсів недостатньо, гравець може закрити вікно покращення, що призводить до завершення дій. Ця діаграма добре ілюструє логіку умов, циклів та взаємодій у процесі покращення, відображаючи, як рішення гравця впливають на хід гри і результативність.

Основні компоненти діаграми діяльності [17]

1. Дія (Action): Дія є базовою операцією, яку виконує система чи її компоненти. Вона починається після отримання необхідних даних або події та

завершується, коли результат передається на наступний етап. На діаграмі дія позначається прямокутником із закругленими кутами.

2. **Перехід (Transition):** Перехід відображає зв’язок між діями чи іншими елементами діаграми. Він позначає момент, коли виконання переходить від однієї дії до іншої, і зображується у вигляді стрілки.

3. **Початковий і кінцевий стани (Initial and Final States):** Початковий стан є точкою, з якої починається процес, тоді як кінцевий стан позначає його завершення. Початковий стан зображується заповненим кружком, а кінцевий — подвійним кільцем із заповненим центром.

4. **Розгалуження (Decision):** Розгалуження моделює вибір між кількома шляхами залежно від певних умов. Воно позначається ромбом, із якого виходять кілька стрілок із зазначеними умовами.

5. **Паралельне виконання (Fork and Join):** Для моделювання одночасного виконання кількох дій та їхньої синхронізації використовуються горизонтальні смуги. Fork — це розгалуження, з якого виходить кілька стрілок, а Join — об’єднання, в яке вони сходяться.

6. **Доріжки (Swimlanes):** Цей елемент використовується для поділу дій між суб’єктами, такими як користувачі, системи або модулі. Доріжки позначаються вертикальними лініями, які створюють структуру, схожу на плавальні доріжки.

Використання діаграми діяльності в бізнес-процесах

Діаграми діяльності часто застосовуються для моделювання бізнес-процесів, допомагаючи описати послідовність дій, що виконуються різними суб’єктами чи відділами організації. Вони наочно відображають, як завдання пов’язані між собою, які умови впливають на кожен етап і які ресурси чи ролі залучені до виконання.

У бізнес-контексті особливо корисними є елементи, такі як доріжки, що дозволяють чітко визначити відповідальність за виконання певних дій. Це робить діаграми діяльності ефективним інструментом для аналізу, оптимізації та

автоматизації бізнес-процесів, а також для поліпшення координації між різними підрозділами організації.

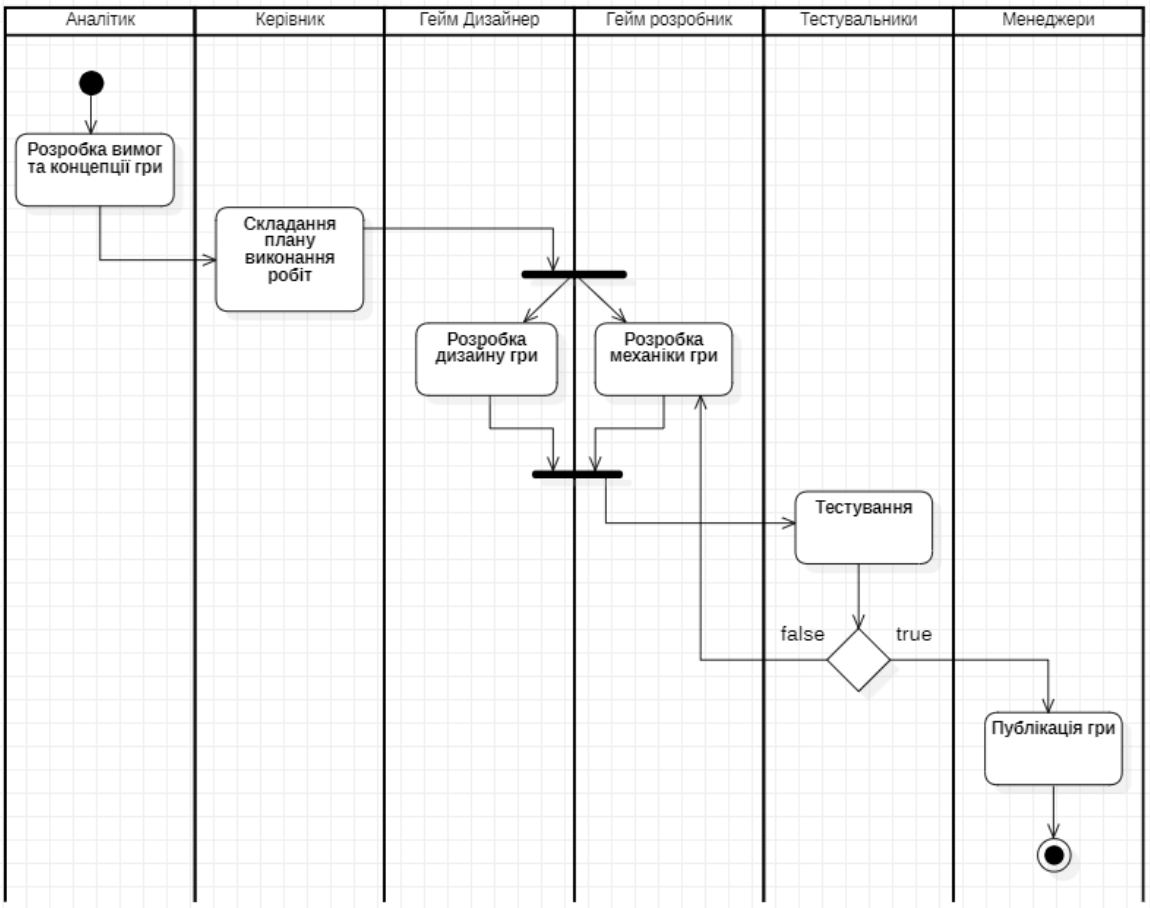

Рисунок 3.7 – Діаграма діяльності для відображення бізнес-процесів

Ця діаграма ілюструє бізнес-процеси в рамках розробки гри, описуючи етапи, через які проходять основні активності та рішення, що впливають на проект. Вона акцентує на ланцюгу дій, що беруть участь у розробці, починаючи від аналізу вимог до публікації гри. Кожен етап представлений у вигляді блоку, а стрілки відображають порядок виконання дій. Наприклад, діаграма може показувати, як на етапі розробки гри формується концепція, яка потім впливає на подальші дії, такі як тестування та публікація. Ця діаграма є цінним інструментом, оскільки вона дозволяє ясно побачити всі етапи розробки, зводячи їх у зрозумілу ієрархію.

3.6 Розробка діаграм компонентів та розгортання

Діаграми компонентів (Component Diagram) і діаграми розгортання (Deployment Diagram) є ключовими інструментами UML, які використовуються для відображення фізичної структури програмної системи. Вони дозволяють наочно представити архітектуру, відобразити компоненти та їхні зв’язки, а також показати фізичне середовище, у якому функціонує система. Ці діаграми сприяють кращому розумінню організації програмного забезпечення, його взаємодії з іншими системами і середовищем виконання [16].

Діаграма компонентів

Цей тип діаграм слугує для моделювання статичної структури програмного забезпечення, представляючи основні програмні модулі системи. Вона показує залежності між компонентами, бібліотеками, інтерфейсами та іншими елементами, які беруть участь у розробці та виконанні системи.

Діаграма компонентів корисна для проєктування модульної архітектури, оскільки вона дає змогу виявляти можливості повторного використання компонентів, оптимізувати структуру коду й аналізувати ключові зв’язки між модулями.

Основні елементи діаграми компонентів [17]:

1. **Компонент (Component):** Це логічна частина системи, наприклад, модуль або бібліотека. Зображується прямокутником із назвою.
2. **Інтерфейс (Interface):** Інтерфейс визначає набір послуг, які компонент може надавати або отримувати. Зображується у вигляді кола, підключеного до компонента.
3. **Зв’язок (Dependency):** Зв’язки відображають залежності між компонентами або між компонентами та інтерфейсами, показуючи, які модулі використовують послуги інших.

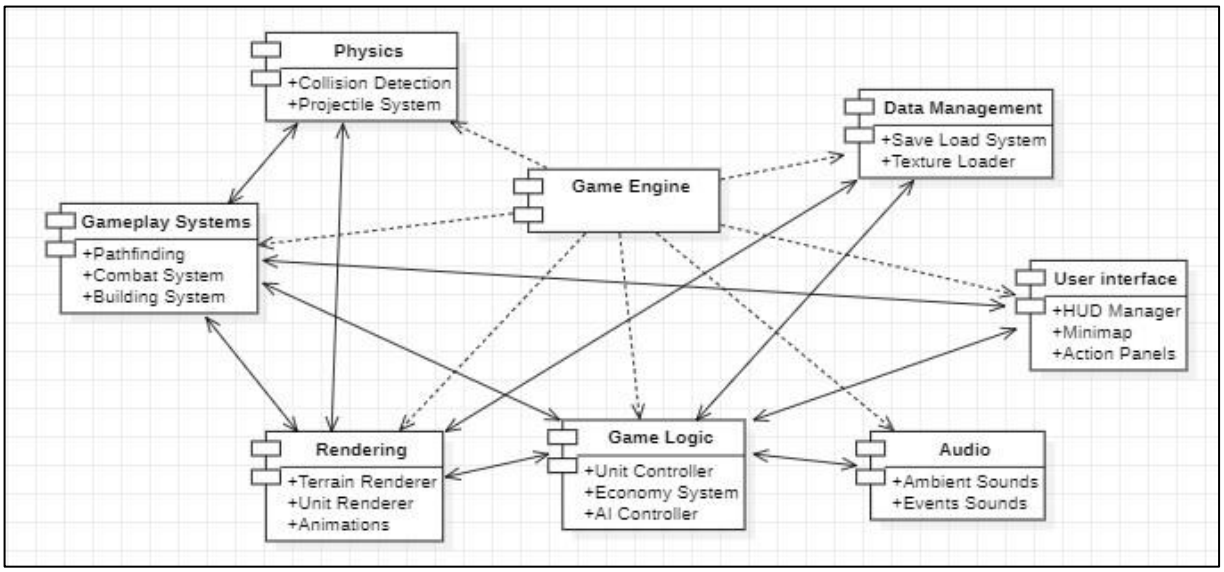

Рисунок 3.8 – Діаграма компонентів

У цій діаграмі представлено основні компоненти системи гри і їх взаємозв’язки. Вона деталізує архітектуру програми, зокрема ключові підсистеми, такі як графіка, логіка гри, система управління даними та інтерфейс користувача. Кожен блок описує функції відповідного компонента, наприклад, система рендерингу відповідає за відображення графічних елементів, в той час як система аудіо управляє звуковими ефектами. Стрілки між компонентами показують, як інформація переміщується тривимірним світом і як окремі модулі взаємодіють між собою, що допомагає виявити зв’язки та залежності. Ця діаграма є важливим елементом для визначення архітектурних рішень, які дозволяють забезпечити ефективність використання ресурсів і зручність подальшої розробки.

Діаграма розгортання

Цей тип діаграм використовується для моделювання фізичного середовища виконання системи. Вона показує вузли (апаратні або мережеві ресурси), на яких розміщуються компоненти, і зв’язки між цими вузлами.

Діаграма розгортання дозволяє зрозуміти, як програмні модулі розподіляються між апаратними елементами й мережевими компонентами.

Основні елементи діаграми розгортання [17]:

1. Вузол (Node): Це фізичний пристрій або апаратний компонент, наприклад, сервер або клієнтський комп’ютер, на якому розміщуються елементи системи. Зображується у вигляді тривимірного прямокутника.

2. Артефакт (Artifact): Фізичний файл або ресурс, наприклад, виконуваний файл чи бібліотека, який розташовується на вузлі. Позначається прямокутником із назвою всередині.

3. Зв’язок (Association): Лінії, які з’єднують вузли, відображають обмін даними або взаємодію між ними.

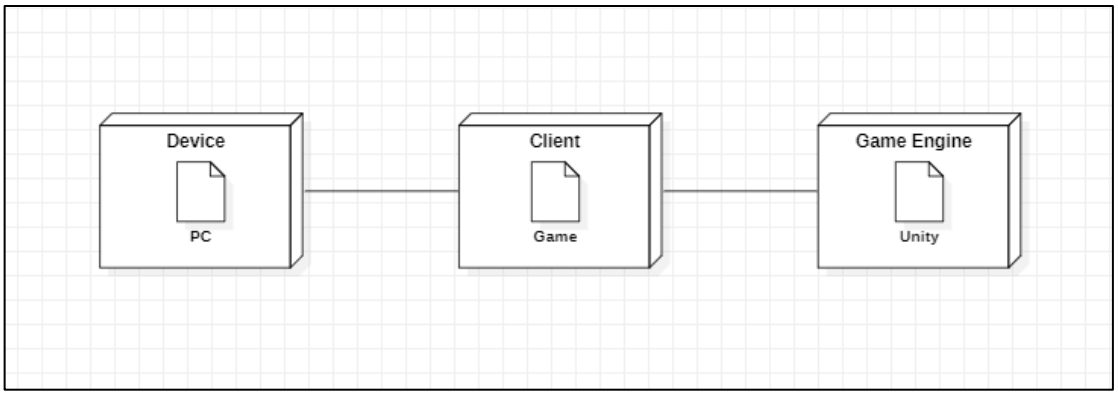

Рисунок 3.9 – Діаграма розгортання

Ця діаграма показує архітектуру гри на рівні розгортання, відображаючи компоненти системи та їх взаємозв’язки. Вона включає три основні елементи: Device (Пристрій), Client (Клієнт), і Game Engine (Ігровий движок). На діаграмі видно, що пристрій (наприклад, ПК) підключається до клієнта, який відповідає за саму гру. Клієнт, у свою чергу, взаємодіє з ігровим движком Unity, забезпечуючи відтворення графіки, фізики та логіки гри. Ця діаграма важлива, оскільки вона ілюструє, як компоненти системи розподілені на різних рівнях архітектури.

Висновки до розділу 3

Третій розділ роботи присвячений моделюванню та проєктуванню ігрового застосунку. У межах цього розділу було виконано декілька ключових етапів. Написання usecase, що включало в себе детальне описання сценаріїв

використання, що дозволяє краще зрозуміти функціональні можливості та взаємодію користувача з грою.

Наступним етапом було створення UML-діаграм. Створення діаграми варіантів використання допомогло візуалізувати основні взаємодії між користувачами та системою, ідентифікуючи ключові функції гри. Побудова діаграм взаємодії (послідовності та кооперації) демонструє порядок взаємодії між різними елементами системи під час виконання певних функцій, що дозволяє чітко зрозуміти процеси, які відбуваються у грі. Діаграми станів та переходів показують можливі стани гри та переходи між ними, що важливо для забезпечення логічної послідовності ігрових подій. Діаграма діяльності описує процеси, які відбуваються в межах гри, з акцентом на дії користувача та системи, забезпечуючи деталізацію кожного кроку. Розробка діаграм компонентів та розгортання показує структурні компоненти системи та їх взаємодію, а також порядок розгортання системи.

Загалом, третій розділ надає цілісний огляд процесів моделювання та проєктування ігрового застосунку. Завдяки виконаним діаграмам та їх аналізу, вдалося створити базову структуру гри, що включає основні функції, взаємодії та архітектурні компоненти. Це створює міцний фундамент для подальшої розробки гри, забезпечуючи її функціональність та відповідність вимогам користувачів.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ

4.1 Створення дизайну ігрового застосунку

Користувацький інтерфейс (UI) у грі має вирішальне значення, оскільки саме він визначає спосіб, у який гравці взаємодіють з грою. Добре продуманий дизайн може суттєво покращити досвід гри, забезпечуючи зручність та інтуїтивність.

В ігровому застосунку є ключові елементи інтерфейсу, такі як головне меню, меню налаштування, меню паузи, меню покупок, меню апгрейдів та інші. Для створення цих елементів були використані стандартні інструменти Unity, зокрема компоненти Canvas, що включають кнопки, текстові поля, панелі та інші елементи.

Дизайн користувацького інтерфейсу відіграє важливу роль не тільки у забезпеченні взаємодії гравця з грою, але й у створенні позитивних емоцій від геймплею. Грамотно спроєктований інтерфейс формує гармонійне та привабливе середовище, де гравці можуть легко орієнтуватися і насолоджуватися процесом гри. Це також сприяє підвищенню залученості гравців та їх задоволеності [18].

Рисунок 4.1 – Головне меню

На рис. 4.1, показано головне меню, яке включає чотири основні кнопки: Start Game, Continue Game, Settings та Exit. По натисненню на кнопку Start Game гравець розпочне нову гру, відкриється сцена з картою гри де і відбуваються основні дії гри.

Рисунок 4.2 – Інтерфейс у грі

По натисненню на Continue Game відкривається меню збережень, де відображаються збереженні сесії ігор, якщо гравець робив збереження, і може продовжити обрану сесію. Також є кнопка Settings, по натисканню на яку, відкривається меню налаштувань, де гравець може налаштувати такі параметри, як гучність музики та звуків, змінити роздільну здатність екрану та ввімкнути або вимкнути повноекранний режим. Та остання кнопка це повний вихід із гри. Мета гравця знищити всі побудови та юнітів противника, якщо це вдається то гравець переміг, але є ще сценарій поразки коли противник знищує всі побудови юнітів гравця, після чого гравець може розпочати нову гру або завантажити збереження, якщо такі існують.

Рисунок 4.3 – Меню збережень

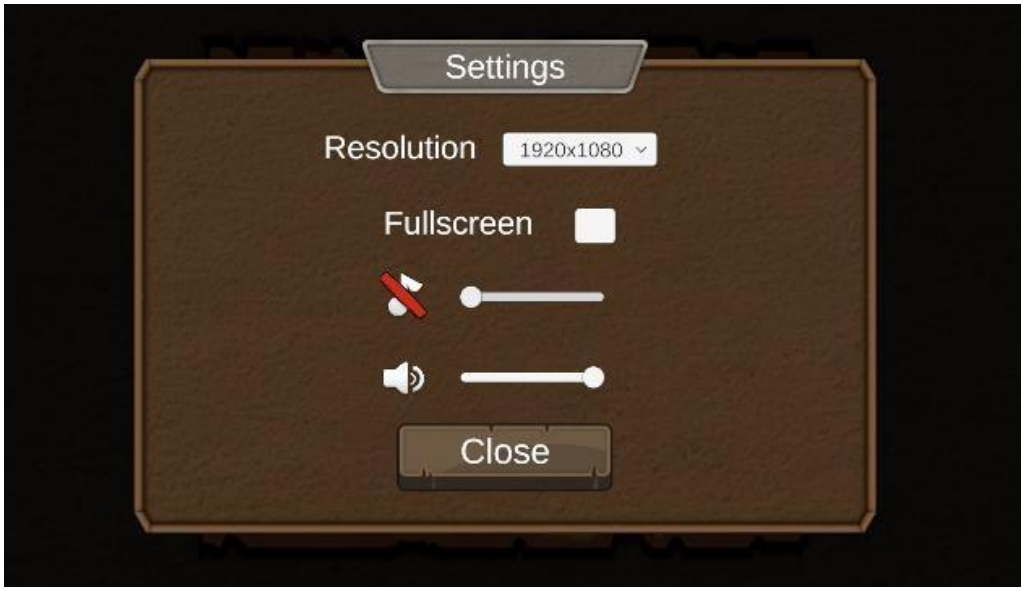

Рисунок 4.4 – Меню налаштувань

Як вже зазначено, розробка зручного та привабливого користувацького інтерфейсу (UI) є критично важливою для забезпечення доступності й зручності використання. Щоб утримати інтерес гравців, необхідно інтегрувати елементи, що покращують якість їхнього досвіду, полегшують доступ до контенту та сприяють зручнішій взаємодії з грою [18].

В результаті маємо такий сценарій: гравець починає гру з мінімальною кількістю ресурсів, за яких можна купити працівників і почати добувати ресурси,

ці ресурси гравець може використати у відповідності до своєї стратегії, це може бути укріплення бази, покупку чи покращення працівників чи одразу створювати воїнів для того щоб швидко знести базу противника поки вона не укріплена.

4.2 Програмна реалізація UI-елементів до застосунку

Програмна реалізація UI-елементів до застосунку полягає в створенні логіки, що керує взаємодією між користувачем та грою через інтерфейс. Це охоплює різні аспекти, такі як обробка введення користувача, відображення графічних компонентів і реакція на події.

Зазвичай програмна реалізація UI-елементів включає створення об’єктів, що представляють графічні елементи, такі як кнопки, текстові поля, панелі тощо. Ці об’єкти налаштовуються з відповідними параметрами, такими як розмір, положення, колір, шрифт і т. д. Додатково, до цих елементів можуть бути додані різні компоненти та функції, що визначають їх поведінку, такі як реакція на кліки, анімації, перехід до інших інтерфейсів тощо. Всі UI-елементи на сцені у Unity розташовуються на на Canvas (полотні). Canvas – це спеціальний об’єкт, який слугує як батьківський для всіх графічних елементів інтерфейсу користувача в сцені Unity [18].

Рисунок 4.5 – Компонент Canvas для UI-елементів

Для програмної реалізації ігрового меню було розроблено такі скрипти: SettingsManager – для функціонування налаштуваннями у головному меню та у самій грі, також слугує менеджером майже всіх кнопок в інтерфейсі, тобто ввімкнення та вимкнення потрібної панелі, початок або завантаження гри, меню налаштувань та збереження даних при їх зміні.

Рисунок 4.6 – Скрипт SettingsManager

Всі кнопки у Unity працюють за допомогою компонента Button, який забезпечує можливість реагувати на різні події, такі як натискання, утримання або відпускання кнопки. Щоб налаштувати поведінку кнопок, потрібно додати відповідні скрипти та обробники подій у редакторі Unity, що дозволяє створити бажану функціональність.

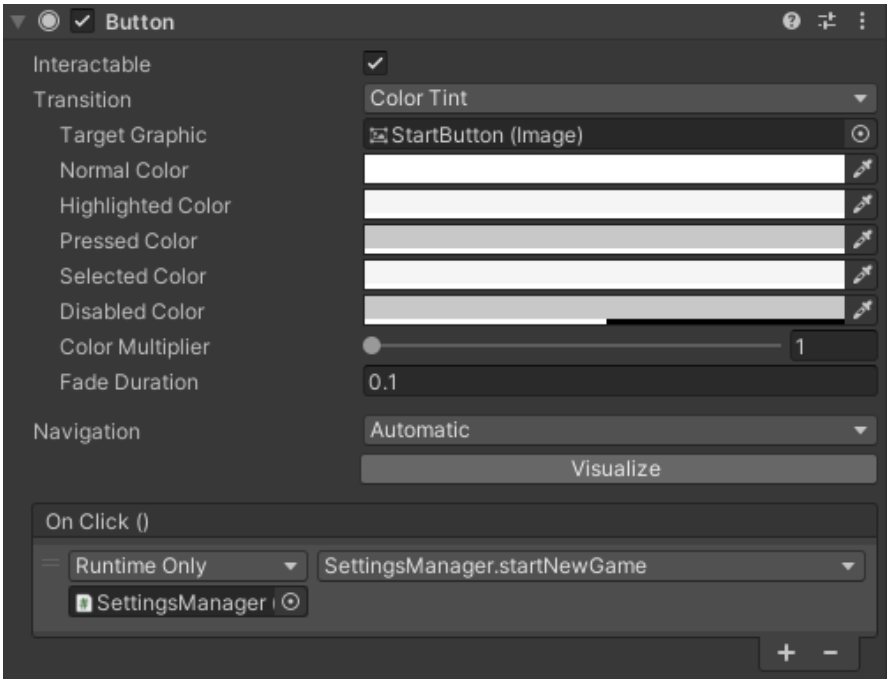

Рисунок 4.7 – Налаштування компоненту на сцені

Далі було розроблено скрипти: ResourceManager та UIManager. Перший відповідає за наявність ресурсів, їх кількість, збільшення або зменшення. А другий відповідає за відображення UI-елементів саме під час гри, по-перше: відображення кількості ресурсів у реальному часі.

```
public class ResourceManager : MonoBehaviour
{
    public static ResourceManager Instance;
    public float wood = 200;
    public float stone = 100;
    public float gold = 50;
    public float food = 50;
    Сообщение Unity | Ссылок: 0
    void Awake()...
    Ссылок: 12
    public bool HasEnoughResources(float requiredWood, float requiredStone, float requiredGold, float requiredFood)
    {
        return wood >= requiredWood && stone >= requiredStone && gold >= requiredGold && food >= requiredFood;
    }
    Ссылок: 7
    public void UseResources(float usedWood, float usedStone, float usedGold, float usedFood)
    {
        wood -= usedWood;
        stone -= usedStone;
        gold -= usedGold;
        food -= usedFood;

        UIManager.Instance.UpdateCurrentPanel();
    }
    Ссылок: 4
    public void AddResources(float wood, float stone, float gold, float food)
    {
        this.wood += wood;
        this.stone += stone;
        this.gold += gold;
        this.food += food;

        UIManager.Instance.UpdateCurrentPanel();
    }
}
```

Рисунок 4.8 – Головний скрипт управління ресурсами

```
public class UIManager : MonoBehaviour
{
    public static UIManager Instance;

    public ResourceManager resourceManager;
    public TextMeshProUGUI goldText;
    public TextMeshProUGUI stoneText;
    public TextMeshProUGUI woodText;
    public TextMeshProUGUI foodText;

    Сообщение Unity | Ссылок: 0
    private void Awake()...
    Сообщение Unity | Ссылок: 0
    private void Update()
    {
        goldText.text = Mathf.FloorToInt(resourceManager.gold).ToString();
        stoneText.text = Mathf.FloorToInt(resourceManager.stone).ToString();
        woodText.text = Mathf.FloorToInt(resourceManager.wood).ToString();
        foodText.text = Mathf.FloorToInt(resourceManager.food).ToString();
    }
```

Рисунок 4.9 – Відображення кількості ресурсів у реальному часі

Також в скрипті UIManager реалізовано налаштування та відображення всіх панелей для управління об’єктами, наприклад коли натискаєш на різні будівлі повинно відображати панель будівлі або юніта. Також повинно налаштовувати панель відповідно до об’єкту що обрав гравець.

```
public void ShowBuildingPanel(Building building)
{
    if (currentPanel != null) { currentPanel.SetActive(false); }
    if (building.isBuilding == true) { return; }
    int panelIndex = (int)building.type;
    currentPanel = buildingPanels[panelIndex];
    currentPanel.SetActive(true);
    selectedBuilding = building.gameObject;

    switch (building.type)
    {
        case Building.BuildingType.Main:
            SetupMainBuildingPanel(currentPanel, building);
            break;
        case Building.BuildingType.Barracks:
            SetupBuildingPanelBarrack(currentPanel, building);
            break;
        case Building.BuildingType.Tower:
            SetupBuildingPanelTower(currentPanel, building);
            break;
    }
}
```

Рисунок 4.10 – Відображення панелі побудови відповідно до його типу

```
Ссылок: 1
public void SetupMainBuildingPanel(GameObject panel, Building building)
{
    TextMeshProUGUI levelText = panel.transform.Find("LevelText").GetComponent<TextMeshProUGUI>();
    Button upgradeButton = panel.transform.Find("UpgradeButton").GetComponent<Button>();
    Image upgradeIcon = upgradeButton.GetComponent<Image>();
    Button buyWorker = panel.transform.Find("BuyWorkerButton").GetComponent<Button>(); ;
    Button upgradeWorkerButton = panel.transform.Find("UpgradeWorkerButton").GetComponent<Button>(); ;
    Image upgradeWorkerIcon = upgradeWorkerButton.GetComponent<Image>();

    levelText.text = $"Level: {building.level}";

    if (building.level < building.maxLevel && ResourceManager.Instance.HasEnoughResources(building.woodCost, building.stoneCost, 0, 0))
    {
        upgradeButton.interactable = true;
        upgradeIcon.sprite = building.upgradeIcons[building.level-1];
        upgradeButton.onClick.RemoveAllListeners();
        upgradeButton.onClick.AddListener(() => building.Upgrade());
    }
    else
    {
        upgradeIcon.sprite = building.upgradeIcons[building.level-1];
        upgradeButton.interactable = false;
    }
```

Рисунок 4.11 – Приклад налаштування кнопок на панелі побудови

Також було реалізовано додавання делегатів для обробки зміни подій, для того щоб при наведенні курсором на кнопку відображати кількість ресурсів яка необхідна для покращення чи придбання.

```
Ссылок: 9
private void AddButtonHoverEffect(Button button, System.Action onHoverEnter, System.Action onHoverExit)
{
    EventTrigger trigger = button.gameObject.GetComponent<EventTrigger>();
    if (trigger == null)
    {
        trigger = button.gameObject.AddComponent<EventTrigger>();
    }
    EventTrigger.Entry entryEnter = new EventTrigger.Entry
    {
        eventID = EventTriggerType.PointerEnter
    };
    entryEnter.callback.AddListener((data) => { onHoverEnter?.Invoke(); });
    trigger.triggers.Add(entryEnter);
    EventTrigger.Entry entryExit = new EventTrigger.Entry
    {
        eventID = EventTriggerType.PointerExit
    };
    entryExit.callback.AddListener((data) => { onHoverExit?.Invoke(); });
    trigger.triggers.Add(entryExit);
}
```

Рисунок 4.12 – Метод додавання кнопкам обробника подій

```
TextMeshProUGUI costText = panel.transform.Find("CostTextList").GetComponent<TextMeshProUGUI>();
AddButtonHoverEffect(upgradeButton,
        () => { costText.text = $"Wood: {building.woodCost}\nStone: {building.stoneCost}"; },
        () => { costText.text = string.Empty; });

AddButtonHoverEffect(buyWorker,
    () => { costText.text = "Gold: 50\nFood: 100"; },
    () => { costText.text = string.Empty; });
```

Рисунок 4.13 – Приклад застосування методу обробника подій

Рисунок 4.14 – Приклад результату роботи скрипту UIManager

В результаті маємо такий сценарій роботи UIManager: Коли натискаєш на будівлю або працівників, з’являється меню знизу екрана на якому знаходяться кнопки покращення будівлі або юнітів та придбання самих юнітів та інформація о вартості того чи іншого в залежності що обираєш. Також якщо обрати працівника відкриється панель із спорудами для їх будування.

4.3 Програмна реалізація механік застосунку

Спочатку було створено керування камерою, а саме скрипт CameraController. Керування камерою буде відбуватись за допомогою кнопок «WASD» на клавіатурі, разом із колесиком миші яке дозволяє приближуватись чи віддалятись. Також було реалізовано гнучкі параметри налаштування керування камерою, а саме: швидкість пересування камери, швидкість та максимальні та мінімальні параметри зуму, та створення меж досяжності для камери для кожної сторони окремо, щоб не вийти за ігрові межі.

```
Скрипт Unity (ссылок на ресурсы: 2) | Ссылок: 3
public class CameraController : MonoBehaviour
{
    public static CameraController Instance;

    public float moveSpeed = 10f;
    public float zoomSpeed = 5f;
    public Vector2 panLimit = new Vector2(50, 50);
    public Vector2 panLimitM = new Vector2(50, 50);
    public float minZoom = 5f;
    public float maxZoom = 20f;

    public GameObject target;
    public bool isBuilding = false;
    Сообщение Unity | Ссылок: 0
    private void Awake()...
    Сообщение Unity | Ссылок: 0
    void Update()
    {
        Vector3 pos = transform.position;

        if (Input.GetKey("w")) pos.z += moveSpeed * Time.deltaTime;
        if (Input.GetKey("s")) pos.z -= moveSpeed * Time.deltaTime;
        if (Input.GetKey("d")) pos.x += moveSpeed * Time.deltaTime;
        if (Input.GetKey("a")) pos.x -= moveSpeed * Time.deltaTime;

        pos.x = Mathf.Clamp(pos.x, -panLimitM.x, panLimit.x);
        pos.z = Mathf.Clamp(pos.z, -panLimitM.y, panLimit.y);
        if (!isBuilding)
        {
            float scroll = Input.GetAxis("Mouse ScrollWheel");
            Camera.main.orthographicSize -= scroll * zoomSpeed;
            Camera.main.orthographicSize = Mathf.Clamp(Camera.main.orthographicSize, minZoom, maxZoom);
        }
        transform.position = pos;
    }
}
```

Рисунок 4.15 – Скрипт CameraController

Рисунок 4.16 – Налаштування параметрів керування камерою

Далі було створено міні-карту, одне із дуже необхідних речей для таких ігор. Для цього було створено ще одну камеру але перпендикулярно направлену на землю зі збільшеною дистанцією, щоб охватити всю карту. Потім створено текстуру для Target Texture на компоненті камери. Потім на Canvas було створено об’єкт RawImage, який буде слугувати місцем відображення картинки з цієї камери та розташовано у правому нижньому кутку для зручності.

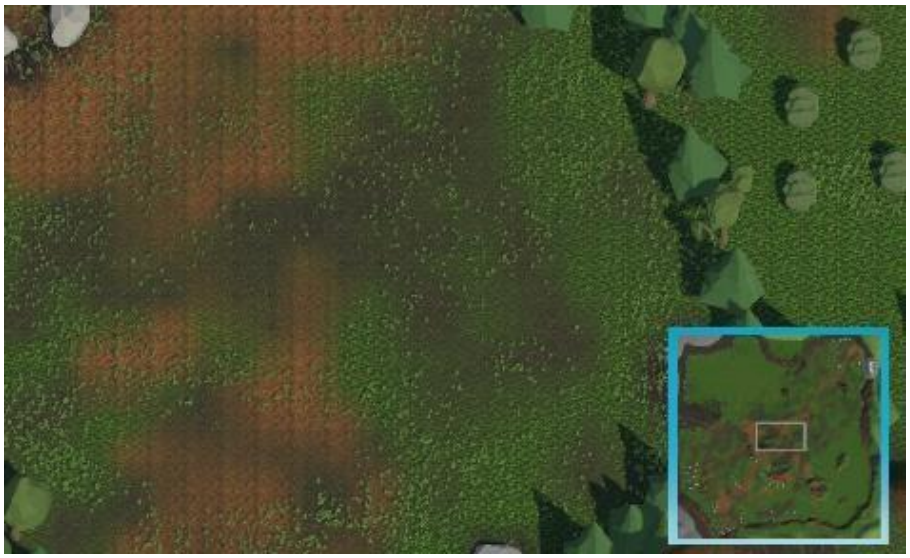

Рисунок 4.17 – Міні-карта

Далі було створено скрипт MinimapCameraController, в якому буде реалізовано відображення де знаходиться головна камера на карті у реальному часі та пересування за допомогою міні-карти через натискання лівої кнопки миші по будь-якій точці на карті.

```
Ссылок: 1
private void UpdateCameraViewRect()
{
    Vector3 viewportPosition = minimapCamera.WorldToViewportPoint(mainCamera.transform.position);
    if (mainCameraViewRect != null)
    {
        mainCameraViewRect.anchorMin = viewportPosition - new Vector3(panLimit.x, panLimit.y);
        mainCameraViewRect.anchorMax = viewportPosition + new Vector3(panLimitM.x, panLimitM.y);
    }
    minimapCamera.orthographicSize = mainCamera.orthographicSize * 6;
}
Ссылок: 1
private void ClampMainCameraIcon()
{
    Vector3 viewportPosition = minimapCamera.WorldToViewportPoint(mainCamera.transform.position);
    viewportPosition.x = Mathf.Clamp(viewportPosition.x, 0f + paddingTLR.x / minimapUI.rect.width, 1f - paddingTLR.y / minimapUI.rect.width);
    viewportPosition.y = Mathf.Clamp(viewportPosition.y, 0f + paddingBLR.x / minimapUI.rect.height, 1f - paddingBLR.y / minimapUI.rect.height);

    if (mainCameraViewRect != null)
    {
        mainCameraViewRect.anchorMin = viewportPosition - new Vector3(panLimit.x, panLimit.y);
        mainCameraViewRect.anchorMax = viewportPosition + new Vector3(panLimitM.x, panLimitM.y);
    }
}
```

Рисунок 4.18 – Відображення місця знаходження камери на міні-карті

```
private void HandleMinimapClick()
{
    if (Input.GetMouseButtonDown(0))
    {
        Vector2 mousePosition = Input.mousePosition;
        if (!RectTransformUtility.RectangleContainsScreenPoint(minimapUI, mousePosition, null))
        {
            return;
        }
        if (RectTransformUtility.ScreenPointToLocalPointInRectangle(minimapUI, mousePosition, null, out Vector2 localPoint))
        {
            Vector2 normalizedPoint = new Vector2(
                (localPoint.x / minimapUI.rect.width) + 0.5f,
                (localPoint.y / minimapUI.rect.height) + 0.5f
            );

            Ray ray = minimapCamera.ViewportPointToRay(normalizedPoint);
            if (Physics.Raycast(ray, out RaycastHit hit))
            {
                Vector3 targetPosition = hit.point + mainCameraOffset;
                targetPosition.y = cameraHeight; // Устанавливаем высоту
                targetPosition.z -= 50;
                mainCamera.transform.position = targetPosition;
            }
        }
    }
}
```

Рисунок 4.19 – Обробка натиску ЛКМ на міні-карті для пересування

Далі, було створено основний скрипт управління юнітами та будівлями – UnitController. В ньому було реалізовано вибір та пересування юнітів, вибір будівель та надання цілей юнітам. Для зручності управління юнітами було створено скрипт Worker, який буде знаходитись на об’єкті працівника. Спочатку було створено вибір юнітів та будівель на ЛКМ та надання команди йти на точку юніту за допомогою ПКМ.

```
private void HandleUnitSelection()
{
    if (Input.GetMouseButtonDown(0))
    {
        if (EventSystem.current.IsPointerOverGameObject()) { return; }
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit))
        {
            GameObject clickedObject = hit.collider.gameObject;
            if (clickedObject.CompareTag("Unit"))
            {
                ClearSelection();
                selectedUnits.Clear();
                SelectUnit(clickedObject);
                return;
            }
            if (clickedObject.CompareTag("Building"))
            {
                ClearSelection();
                selectedUnits.Clear();
                SelectBuilding(clickedObject);
                return;
            }
        }
    }
}
```

Рисунок 4.20 – Вибір юніта або будівлі на ЛКМ

Рисунок 4.21 – Вибір юніта на сцені

```
private void HandleUnitCommand()
{
    if (Input.GetMouseButtonDown(1))
    {
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if (Physics.Raycast(ray, out hit))
        {
            for (int i = 0; i < selectedUnits.Count; i++)
            {
                GameObject unit = selectedUnits[i];
                NavMeshAgent agent = unit.GetComponent<NavMeshAgent>();
                NavMeshPath path = new NavMeshPath();
                if (agent.CalculatePath(hit.point, path))
                {
                    agent.SetPath(path);
                }
            }
        }
    }
}
```

Рисунок 4.22 – Відправка юніта на точку за допомогою ПКМ

Для переміщення персонажів було використано компонент NavMeshAgent, який відповідає за автоматичне планування маршрутів та уникнення перешкод. Цей компонент дозволяє персонажам переміщатися по навігаційній сітці (NavMesh), яка генерується на основі поверхонь сцени. Задаючи кінцеву точку призначення, NavMeshAgent автоматично обирає оптимальний шлях [19].

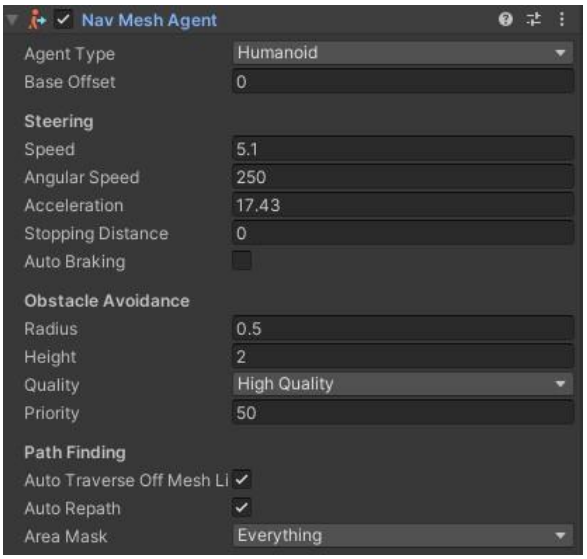

Рисунок 4.23 – Компонент Nav Mesh Agent

Створення навігаційної сітки створюється за допомогою компонента NavMeshSurface на основній поверхні землі. А параметри, такі як кут нахилу підйому або висота підйому на перешкоди налаштовуються у вікні Window→AI→Navigation [19].

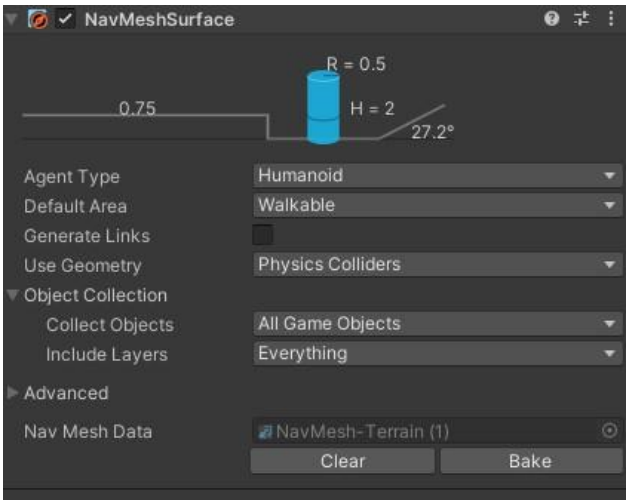

Рисунок 4.24 – Компонент Nav Mesh Surface

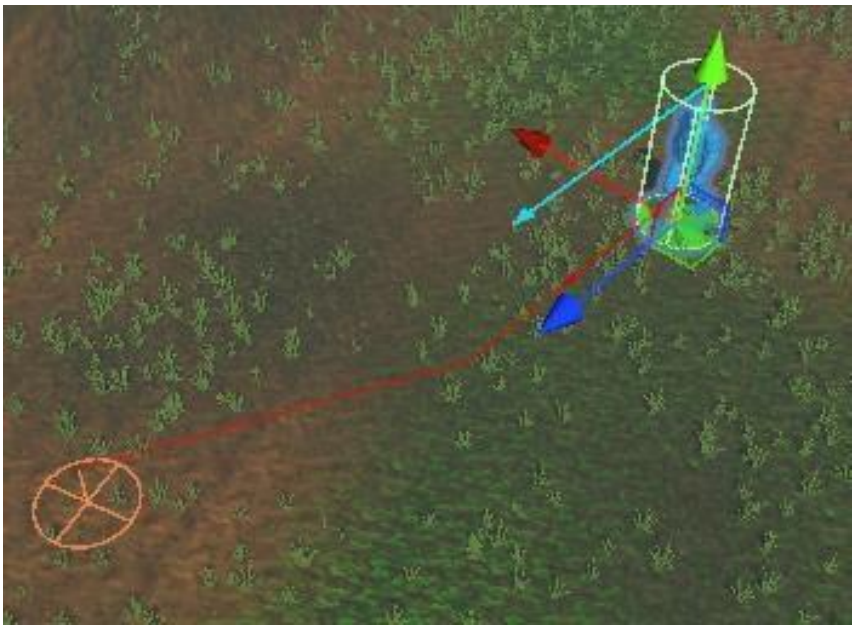

Рисунок 4.25 – Пересування та пошук оптимального шляху [19]

Також потрібно реалізувати можливість виділяти декілька юнітів одразу та реалізувати організоване пересування двох і більше юнітів у шеренгу, щоб уникнути нескінченного пересування юнітів один в одного на одному місці.

```
Ссылок: 0
private void HandleUnitSelection()
{
    if (Input.GetMouseButtonDown(0))
    {
        if (EventSystem.current.IsPointerOverGameObject())...
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;
        if (Physics.Raycast(ray, out hit))...
        isSelecting = true;
        startMousePosition = Input.mousePosition;
    }
    if (isSelecting)
    {
        endMousePosition = Input.mousePosition;
        if (Input.GetMouseButtonUp(0))
        {
            ClearSelection();
            selectedUnits.Clear();
            SelectUnitsInBox();
            isSelecting = false;
        }
    }
}
```

Рисунок 4.26 – Вибір одразу декількох юнітів

Рисунок 4.27 – Вибір одразу декількох юнітів на сцені

```
Ссылок: 1
public void MoveUnitsInFormation(Vector3 destination)
{
    if (selectedUnits.Count == 0) return;
    float formationSpacing = 2.0f;
    int unitsPerRow = Mathf.CeilToInt(Mathf.Sqrt(selectedUnits.Count));

    for (int i = 0; i < selectedUnits.Count; i++)
    {
        GameObject unit = selectedUnits[i];
        Worker worker = unit.GetComponent<Worker>();
        NavMeshAgent agent = unit.GetComponent<NavMeshAgent>();
        if (agent != null)
        {
            int row = i / unitsPerRow;
            int column = i % unitsPerRow;
            Vector3 offset = new Vector3(column * formationSpacing, 0, row * formationSpacing);
            agent.stoppingDistance = 0.5f;
            NavMeshPath path = new NavMeshPath();
            if (agent.CalculatePath(destination + offset, path))
            {
                agent.SetPath(path);
            }
        }
    }
}
```

Рисунок 4.28 – Метод організованого пересування

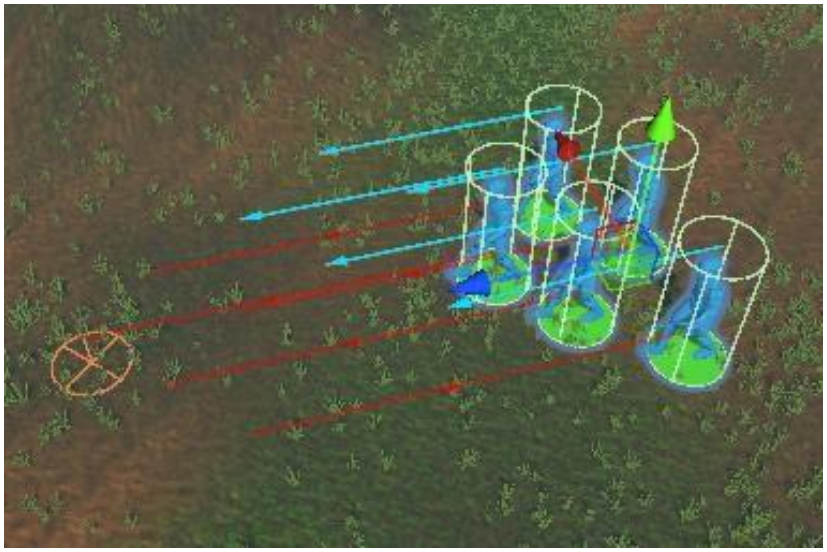

Рисунок 4.29 – Організоване пересування на сцені [19]

Далі було розроблено систему добування ресурсів, таких як камінь, дерево, золото та їжу. Для цього було створено префаби цих ресурсів для подальшого використання на сцені, кожен із типів ресурсів має свій відповідний тег, тому було додано до управління юнітами частину, де визначається чи є об’єкт, на який відправили юніта, ресурсом.

```
if (Input.GetMouseButtonDown(1))
{
    Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit))
    {
        if (hit.collider.CompareTag("Wood") || hit.collider.CompareTag("Stone") || hit.collider.CompareTag("Gold") || hit.collider.CompareTag("Food"))
        {
            foreach (GameObject unit in selectedUnits)
            {
                Worker worker = unit.GetComponent<Worker>();
                if (worker != null)
                {
                    worker.MoveToResource(hit.collider.gameObject);
                }
            }
```

Рисунок 4.30 – Перевірка на наявність ресурсу

Також було створено скрипт ResourceHealth який буде знаходитись на об’єктах ресурсів. В цьому скрипті також реалізується створення так званих точок добування ресурса, це створено для того щоб був візуально привабливий процес добування ресурса навколо нього та щоб працівники не заважали один одному. Кількість точок обмежено.

```
Ссылок: 1
private void GenerateGatherPoints()
{
    for (int i = 0; i < maxGatherPoints; i++)
    {
        float angle = i * Mathf.PI * 2 / maxGatherPoints;
        Vector3 pointPosition = transform.position + new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle)) * gatherRadius;
        GatherPoint point = new GatherPoint
        {
            Position = pointPosition,
            IsOccupied = false
        };
        gatherPoints.Add(point);
    }
}
```

Рисунок 4.31 – Генерація точок добування

```
Ссылок: 1
public bool TryGetNearestFreeSpot(Vector3 workerPosition, out Vector3 spot)
{
    spot = Vector3.zero;
    float shortestDistance = float.MaxValue;
    GatherPoint nearestPoint = null;
    foreach (var point in gatherPoints)
    {
        if (!point.IsOccupied)
        {
            float distance = Vector3.Distance(workerPosition, point.Position);
            if (distance < shortestDistance)
            {
                shortestDistance = distance;
                nearestPoint = point;
            }
        }
    }
    if (nearestPoint != null)
    {
        spot = nearestPoint.Position;
        nearestPoint.IsOccupied = true;
        return true;
    }
    return false;
}
```

Рисунок 4.32 – Метод пошуку вільного місця на ресурсі

Далі було додано до скрипту Worker, який знаходиться на працівниках, відповідні частини коду для відправки на ресурс та подальшого його добування.

```
Ссылок: 2
public void MoveToResource(GameObject resource)
{
    StopGathering();
    targetResource = resource;
    isGathering = false;
    currentTask = ResourceType.None;

    ResourceHealth resourceHealth = resource.GetComponent<ResourceHealth>();
    if (resourceHealth != null && resourceHealth.TryGetNearestFreeSpot(transform.position, out gatherSpot))
    {
        isSpotReserved = true;
        agent.stoppingDistance = 0.5f;
        NavMeshPath path = new NavMeshPath();
        if (agent.CalculatePath(gatherSpot, path))
        {
            agent.SetPath(path);
        }
    }
    else
    {
        Debug.Log($"{name}: No points available for the resource {resource.name}.");
    }
}
```

Рисунок 4.33 – Метод відправки на ресурс

Метод MoveToResource визначає ціль та відправляє обраного працівника на ресурс, і коли той досягає цілі то починається добування.

```
private void Update()
{
    if (isGathering)
    {
        float distance = Vector3.Distance(transform.position, gatherSpot);

        if (distance > gatherRange)
        {
            isGathering = false;
            agent.isStopped = false;
            agent.SetDestination(gatherSpot);
        }
        gatherTimer -= Time.deltaTime;
        if (gatherTimer <= 0f)
        {
            GatherResource();
            gatherTimer = gatherCooldown;
        }
    }
}
Ссылок: 1
private void GatherResource()
{
    if (targetResource == null || !isGathering) return;

    ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
    if (resourceHealth != null)
    {
        resourceHealth.TakeDamage(gatherAmount);
    }
}
```

Рисунок 4.34 – Процес добування

Рисунок 4.35 – Процес добування ресурсу на сцені

Також якщо потрібно відмінити процес добування або змінити ціль, то викликаються окремі методи зупинки добування та очищення занятої точки ресурсу для подальшої можливості зайняти її іншим працівником.

```
Ссылок: 5
public void StopGathering()
{
    if (isGathering || isSpotReserved)
    {
        ReleaseSpot();
        ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
        resourceHealth?.UpdateFreeSpots();
    }
    currentTask = ResourceType.None;
    isGathering = false;
    agent.isStopped = false;
    targetResource = null;
}

Ссылок: 1
private void ReleaseSpot()
{
    if (isSpotReserved && targetResource != null)
    {
        ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
        if (resourceHealth != null)
        {
            resourceHealth.ReleaseSpot(gatherSpot);
        }
        isSpotReserved = false;
    }
}
```

Рисунок 4.36 – Методи зупинки добування ресурсу

Для того щоб отримувати ресурси від добування у ресурсів є кількість здоров’я, і під час добування працівники наносять ударів ресурсу і це автоматично додає цей ресурс до загальної кількості. І якщо кількість здоров’я доходить до нуля то додається ще відповідна кількість ресурсу та ресурс знищується.

```
Ссылок: 1
public void TakeDamage(float amount)
{
    currentHealth -= amount;
    AddResourceToManager(resourceType, amount);

    if (currentHealth <= 0)
    {
        AddResourceToManager(resourceType, resourceAmount);
        currentHealth = 0;

        UpdateFreeSpots();
        DestroyResource();
    }
}
Ссылок: 2
private void AddResourceToManager(ResourceType type, float amount)
{
    switch (type)
    {
        case ResourceType.Wood:
            ResourceManager.Instance.AddResources(amount, 0, 0, 0);
            break;
        case ResourceType.Stone:
            ResourceManager.Instance.AddResources(0, amount, 0, 0);
            break;
        case ResourceType.Gold:
            ResourceManager.Instance.AddResources(0, 0, amount, 0);
            break;
        case ResourceType.Food:
            ResourceManager.Instance.AddResources(0, 0, 0, amount);
            break;
    }
}
```

Рисунок 4.37 – Нанесення урону та отримання ресурсів

Також було реалізоване автоматичне знаходження сусідніх ресурсів такого самого типу для продовження добування цього типу ресурсу якщо той був добутий. А якщо таких не знайшлося то працівники просто зупиняють роботу та стоять в очікуванні нових запитів.

```
Ссылок: 1
private void FindNewResource()
{
    Collider[] colliders = Physics.OverlapSphere(transform.position, gatherRadius * 5);
    GameObject nearestResource = null;
    float shortestDistance = float.MaxValue;

    foreach (var col in colliders)
    {
        ResourceHealth resource = col.GetComponent<ResourceHealth>();
        if (resource != null && resource != this && resource.resourceType == resourceType)
        {
            float distance = Vector3.Distance(transform.position, resource.transform.position);
            if (distance < shortestDistance)
            {
                shortestDistance = distance;
                nearestResource = resource.gameObject;
            }
        }
    }
    if (nearestResource != null)
    {
        Collider[] workersColliders = Physics.OverlapSphere(transform.position, gatherRadius * 1.5f);
        foreach (var col in workersColliders)
        {
            Worker worker = col.GetComponent<Worker>();
            if (worker != null && worker.targetResource == gameObject)
            {
                worker.MoveToResource(nearestResource);
            }
        }
    }
    else NotifyWorkersToStop();
}
```

Рисунок 4.38 – Пошук сусідніх ресурсів такого же типу

Потім було створено систему будівництва споруд. Для цього було створено скрипт BuildingSystem який також знаходиться на працівниках. Коли обираєш працівника то з’являється панель будівництва із двома спорудами, це башня, яка атакує всіх ворогів у радіусі, а друга – казарма з допомогою якої можна купувати воїнів. Коли обираєш один із цих споруд, авжеш якщо достатньо ресурсів, то починається режим будівництва.

```
Ссылок: 1
public void StartBuildingMode(Worker.BuildingType type)
{
    if (isBuildingMode) return;
    isBuildingMode = true;
    selectedBuildingType = type;
    GameObject prefab = type == Worker.BuildingType.Tower ? towerPrefab : barracksPrefab;
    previewObject = Instantiate(prefab);
    SetPreviewMode(previewObject, true);
    worker.enabled = false;
    UnitController.Instance.enabled = false;
}
```

Рисунок 4.39 – Початок режиму будівництва

Після активації режиму будівництва на місці курсору з’являється прев’ю споруди яку було обрано, тобто силует цієї споруди з’являється на місці курсору і цей силует прив’язаний до курсора. Режим будівництва можна також відмінити натиснув ПКМ. Працює метод HandleBuildingMode який в реальному часі перевіряє чи можна на цьому місці збудувати цю будівлю.

Рисунок 4.40 – Прев’ю споруди для будівництва

```
Ссылок: 2
private bool CanPlaceBuilding()
{
    if (previewObject == null) return false;

    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    if (!Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity) || !hit.collider.CompareTag("Ground"))
    {
        return false;
    }
    float checkInterval = 0.5f;
    List<Vector3> boundaryPoints = previewObject.GetComponent<Building>().GenerateBoundaryPoints(checkInterval);

    foreach (var point in boundaryPoints)
    {
        Collider[] colliders = Physics.OverlapSphere(point, 0.5f);
        foreach (var detected in colliders)
        {
            if (detected.gameObject != previewObject &&
                (detected.CompareTag("Building") || detected.CompareTag("Unit") || detected.gameObject.layer == LayerMask.NameToLayer("Resource")))
            {
                return false;
            }
        }
    }
    return true;
}
```

Рисунок 4.41 – Перевірка дозволу на будівництво

Метод CanPlaceBuilding перевіряє чи є в районі силуету споруди яку обрано для будівництва коллайдери інших об’єктів з тегами будівлі або юнітів

та коллайдери ресурсів, і якщо є то будувати тут не можна. Також прев'ю споруди підсвічується червоним щоб було інтуїтивно зрозуміло де будувати не можна.

Рисунок 4.42 – Відмова у будівництві

Перевірка дозволу будівництва відбувається за допомогою створення точок меж будівлі, і потім перевіряється наявність на цих точках інших об'єктів.

```
Ссылок: 1
public List<Vector3> GenerateBoundaryPoints(float interval)
{
    List<Vector3> boundaryPoints = new List<Vector3>();

    BoxCollider collider = GetComponent<BoxCollider>();
    if (collider == null)
    {
        Debug.LogError("Collider not found on the building!");
        return boundaryPoints;
    }
    Vector3 size = collider.size * transform.localScale.x;
    Vector3 center = transform.position;

    for (float x = -size.x / 2; x <= size.x / 2; x += interval)
    {
        boundaryPoints.Add(center + new Vector3(x, 0, -size.z / 2));
        boundaryPoints.Add(center + new Vector3(x, 0, size.z / 2));
    }
    for (float z = -size.z / 2; z <= size.z / 2; z += interval)
    {
        boundaryPoints.Add(center + new Vector3(-size.x / 2, 0, z));
        boundaryPoints.Add(center + new Vector3(size.x / 2, 0, z));
    }
    return boundaryPoints;
}
```

Рисунок 4.43 – Генерація точок меж будівлі

Після підтвердження місця для будівництва, на цьому місці з'являється ця будівля і працівник йде до неї для того щоб побудувати її.

```
Ссылок: 2
public void SendWorkerToConstruction(GameObject workerObject, GameObject building)
{
    Worker worker = workerObject.GetComponent<Worker>();
    if (worker == null || building == null) return;

    Building buildingComponent = building.GetComponent<Building>();
    if (worker.isConstructionSpotReserved)
    {
        StopWorkerConstruction(worker, worker.targetBuilding.GetComponent<Building>());
    }
    if (buildingComponent.isBuilding && buildingComponent.TryGetNearestFreeSpot(worker.transform.position, out Vector3 constructionSpot))
    {
        worker.targetBuilding = building;
        worker.isConstructing = false;
        worker.isConstructionSpotReserved = true;
        worker.agent.stoppingDistance = 0.5f;
        NavMeshPath path = new NavMeshPath();
        if (worker.agent.CalculatePath(constructionSpot, path))
        {
            worker.agent.SetPath(path);
        }
        worker.StartCoroutine(StartConstructionWhenArrived(worker, buildingComponent, constructionSpot));
    }
    else
    {
        Debug.Log("There are no available points for construction.");
    }
}
```

Рисунок 4.44 – Відправка працівника будувати споруду

Коли працівник доходить до цієї споруди запускається корутина, яка керує процесом будівництва цієї споруди.

Рисунок 4.45 – Побудова споруди

Також було створено систему атаки башти по всім противникам у радіусі атаки. Якщо башта побудована то в її радіусі завжди проводиться пошук інших колайдерів, і якщо цей об’єкт противник за запускається снаряд, у цьому випадку стрілу, у противника.

```
private void TryAttack()
{
    Collider[] enemies = null;
    if (this.gameObject.layer == LayerMask.NameToLayer("Blue"))
    {
        enemies = Physics.OverlapSphere(transform.position, attackRadius, LayerMask.GetMask("Red"));
    }else if (this.gameObject.layer == LayerMask.NameToLayer("Red"))
    {
        enemies = Physics.OverlapSphere(transform.position, attackRadius, LayerMask.GetMask("Blue"));
    }
    if (enemies.Length > 0)
    {
        Collider target = GetNearestEnemy(enemies);
        if (target != null)
        {
            ShootAtEnemy(target.transform);
        }
    }
}
Ссылок: 1
private Collider GetNearestEnemy(Collider[] enemies)...
Ссылок: 1
private void ShootAtEnemy(Transform target)
{
    GameObject projectile = Instantiate(projectilePrefab, spawnPoint.position, Quaternion.identity);
    Projectile projectileScript = projectile.GetComponent<Projectile>();
    if (projectileScript != null)
    {
        projectileScript.SetTarget(target);
    }
}
```

Рисунок 4.46 – Система атаки башти

Далі було розроблено систему атаки і для воїнів і для працівників, але у других показники менше. Для цього в управлінні юнітами на ПКМ було додано обробник, якщо ціль юніт противника то йти атакувати його.

```
}else if (hit.collider.CompareTag("Unit"))
{
    foreach (GameObject unit in selectedUnits)
    {
        Warrior warrior = unit.GetComponent<Warrior>();
        Worker worker = unit.GetComponent<Worker>();
        if(unit.gameObject.layer == LayerMask.NameToLayer("Blue"))
        {
            if (hit.collider.gameObject.layer == LayerMask.NameToLayer("Red"))
            {
                if (warrior != null) warrior.SetTarget(hit.collider.gameObject);
                if (worker != null) worker.SetTarget(hit.collider.gameObject);
            }
        }
        if (unit.gameObject.layer == LayerMask.NameToLayer("Red"))
        {
            if (hit.collider.gameObject.layer == LayerMask.NameToLayer("Blue"))
            {
                if (warrior != null) warrior.SetTarget(hit.collider.gameObject);
                if (worker != null) worker.SetTarget(hit.collider.gameObject);
            }
        }

    }
}
```

Рисунок 4.47 – Призначення атаки на ПКМ

Після призначення цілі, юніт буде переслідувати ціль поки не дійде до неї або якщо не змінити ціль.

```
Ссылок: 1
private void AttackTarget()
{
    animator.SetBool("IsAttacking", true);
    agent.transform.LookAt(currentTarget.transform.position);
    if (attackTimer <= 0.0f && currentTarget != null)
    {
        attackTimer = attackCooldown;
        Worker worker = currentTarget.GetComponent<Worker>();
        Warrior warrior = currentTarget.GetComponent<Warrior>();

        if (worker != null)
        {
            worker.TakeDamage(attackDamage);
            worker.IsAttacked(this.gameObject);
        }
        if (warrior != null)
        {
            warrior.TakeDamage(attackDamage);
            warrior.IsAttacked(this.gameObject);
        }
    }
}
Ссылок: 3
public void TakeDamage(float damage)
{
    health -= damage;
    healthBar.SetHealth(health, maxHealth);
    if (health <= 0)
    {
        FindNewTarget();
        Die();
    }
}
```

Рисунок 4.48 – Система атаки юнітів

Так само як і для ресурсів, було реалізовано пошук нових цілей поблизу після знищення цілі.

```
private void FindNewTarget()
{
    Collider[] colliders = Physics.OverlapSphere(transform.position, attackRange * 5);
    GameObject nearestEnemy = null;
    float shortestDistance = float.MaxValue;
    foreach (var col in colliders)
    {
        Warrior warrior = col.GetComponent<Warrior>();
        if (warrior != null && warrior != this && warrior.gameObject.layer == this.gameObject.layer)
        {
            float distance = Vector3.Distance(transform.position, warrior.transform.position);
            if (distance < shortestDistance) { shortestDistance = distance; nearestEnemy = warrior.gameObject;}
        }
        Worker worker = col.GetComponent<Worker>();
        if (warrior != null && warrior != this && warrior.gameObject.layer == this.gameObject.layer)
        {
            float distance = Vector3.Distance(transform.position, warrior.transform.position);
            if (distance < shortestDistance) { shortestDistance = distance; nearestEnemy = warrior.gameObject;}
        }
    }
    if (nearestEnemy != null)
    {
        Collider[] unitColliders = Physics.OverlapSphere(transform.position, attackRange * 1.5f);
        foreach (var col in unitColliders)
        {
            Worker worker = col.GetComponent<Worker>();
            if (worker != null && worker.currentTarget == gameObject) {worker.currentTarget = nearestEnemy;}
            Warrior warrior = col.GetComponent<Warrior>();
            if (warrior != null && warrior.currentTarget == gameObject) {warrior.currentTarget = nearestEnemy;}
        }
    }
    else NotifyAttackersToStop();
}
```

Рисунок 4.49 – Пошук нової цілі поблизу

Також було створено систему коли юніти можуть самі атакувати противників. Коли їх починають бити то вони захищаються і починають бити у відповідь, також прораховуючи потім чи є поблизу ще противники, якщо є то продовжує їх бити поки не залишиться поблизу ще противників.

```
Ссылок: 2
public void IsAttacked(GameObject attacker)
{
    if (currentTarget == null)
    {
        bool isMoving = agent.velocity.magnitude > 0.1f;
        if (isMoving) { return; }
        currentTarget = attacker;
    } else{return;}

    Collider[] nearbyColliders = Physics.OverlapSphere(transform.position, attackRange * 4f);
    foreach (var col in nearbyColliders)
    {
        Warrior nearbyWarrior = col.GetComponent<Warrior>();
        Worker nearbyWorker = col.GetComponent<Worker>();

        if (nearbyWarrior != null && nearbyWarrior != this && nearbyWarrior.gameObject.layer == gameObject.layer)
        {
            if (nearbyWarrior.currentTarget == null) { AssignNearestEnemy(attacker, nearbyWarrior); }
        }
        if (nearbyWorker != null && nearbyWorker.currentTarget == null && nearbyWorker.gameObject.layer == gameObject.layer)
        {
            AssignNearestEnemy(attacker, nearbyWorker);
        }
    }
}
```

Рисунок 4.50 – Початок атаки у відповідь

Рисунок 4.51 – Приклад бійки на сцені

Для зручності було реалізовано візуальне інформування кількості здоров’я юнітів щоб розуміти можливості того чи іншого юніта. Для цього було створено відповідну шкалу над всіма юнітами. На префабі юнітів було додано об’єкт Canvas, для того щоб шкала була UI-елементом, тільки знаходячись безпосередньо у 3Д просторі на сцені.

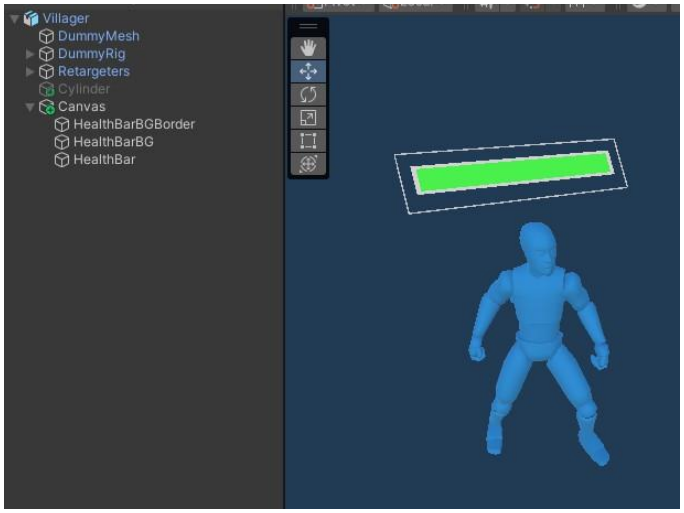

Рисунок 4.52 – Шкала здоров’я у префабі юнітів

Далі було створено скрипт HealthBar який буде знаходитись на цій шкалі та синхронізувати кількість здоров’я юніта із шкалою у реальному часі.

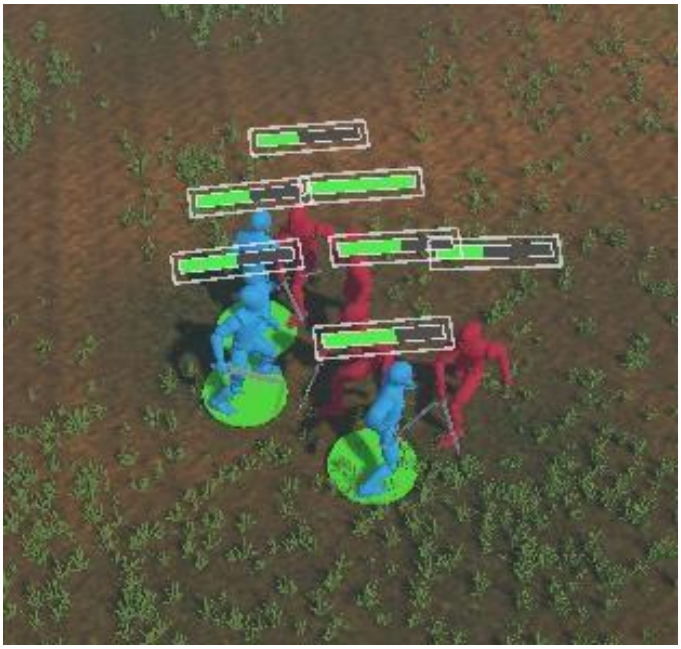

Рисунок 4.53 – Приклад відображення шкали при бійці на сцені

Також для створення більш реалістичного і захоплюючого ігрового процесу було реалізовано систему анімацій для юнітів. Вони забезпечують візуальний відгук на дії гравця, роблять юнітів в грі живими та динамічними. Для цього на об’єкт юнітів додається компонент Animator який потрібен для присвоєння анімації об’єктам на сцені [20].

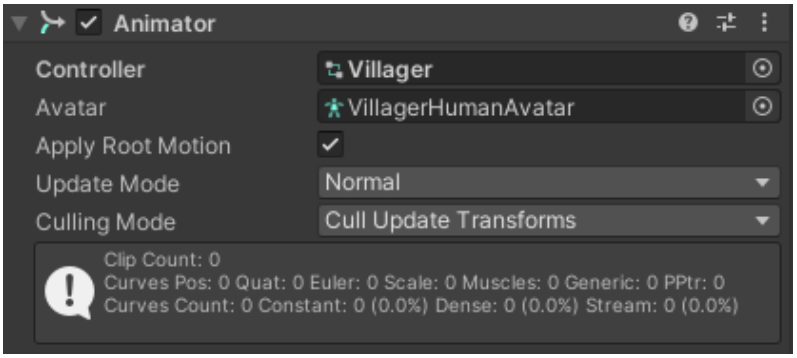

Рисунок 4.54 – Компонент Animator на юнітах

Далі було створено AnimatorController в якому налаштовуються стани та переходи між анімаціями. Налаштовано стани та переходи між ними для працівників та воїнів окремо. Для переходів створюються параметри із своїми назвами для кожної дії для подальшого налаштування переходів у коді [20].

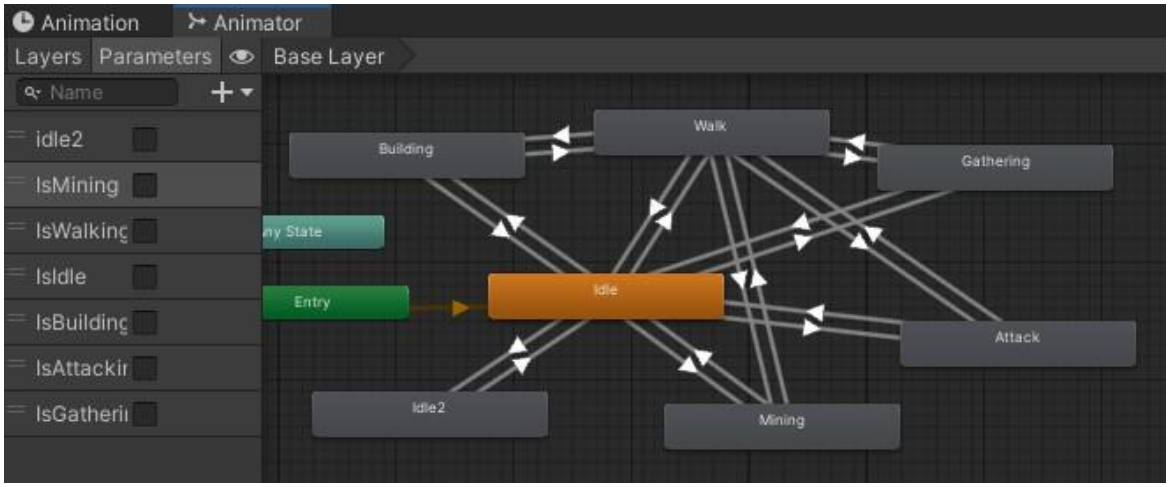

Рисунок 4.55 – Налаштування AnimatorController для працівників

Параметри створюються із змінною bool із двома станами true та false, що дає змогу гнучко налаштовувати переходи у коді. В якості контролера було створено скрипт AnimationController в якому було реалізовано головний метод для зміни анімації у відповідності до стану поточного завдання юніта. Даний метод буде застосовуватись коли відбувається зміна дій юніта [20].

```
public void CheckAnimation()
{
    switch (worker.currentTask)
    {
        case ResourceType.Wood:
            animator.SetBool("IsAttacking", true);
            break;
        case ResourceType.Stone:
            animator.SetBool("IsMining", true);
            break;
        case ResourceType.Gold:
            animator.SetBool("IsMining", true);
            break;
        case ResourceType.Food:
            animator.SetBool("IsIdle", false);
            animator.SetBool("IsWalking", false);
            animator.SetBool("IsGathering", true);
            break;
        case ResourceType.Build:
            animator.SetBool("IsBuilding", true);
            break;
        case ResourceType.Attack:
            animator.SetBool("IsAttacking", true);
            break;
        case ResourceType.None:
            animator.SetBool("IsMining", false);
            animator.SetBool("IsAttacking", false);
            animator.SetBool("IsBuilding", false);
            animator.SetBool("IsGathering", false);
            break;
    }
}
```

Рисунок 4.56 – Головний метод зміни анімацій для юнітів

Також було додано звуки у грі за допомогою компоненту Audio Source у інспекторі. Було додано фонове супроводження для гри для атмосферності та більш привабливого процесу гри. Також можна змінювати гучність музики у налаштуваннях якщо та набридла або занадто гучна.

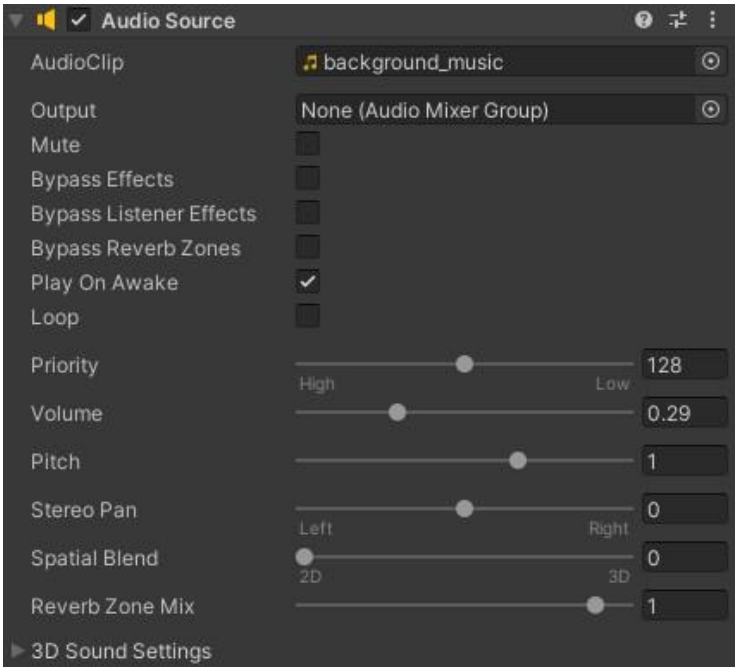

Рисунок 4.57 – Компонент Audio Source

На кінець було створено систему ШІ, тобто штучного інтелекту в якості противника. Для цього було створено скрипт AIManager і там реалізовно методи для ШІ, а саме прийняття рішень у тій чи іншій дії, це може бути покупка працівників, відправка їх на ресурс, будування, зміна добування ресурсів у відповідності до необхідних, покупка воїнів та саме атака гравця [21].

```
Ссылок: 1
void MakeDecision()
{
    UpdateResourcePriority();
    switch (currentState)
    {
        case AIState.Expanding:...
        case AIState.Attacking:...
    }
}
Ссылок: 1
void UpdateResourcePriority()
{
    if (workers.Count >= 10 && !hasBuiltBarracks)
    {
        prioritizedResource = "Wood";
        prioritizedResource2 = "Stone";
    }
    else if (playerUnits.Count >= lastArmyCountForTower + 1)
    {
        prioritizedResource = "Wood";
        prioritizedResource2 = "Stone";
    }
    else...
}
Ссылок: 1
void HandleExpansion()
{
    if (NeedsMoreWorkers())...
    if (workers.Count >= 10 && !hasBuiltBarracks)
    {
        if (HasResourcesForBuilding(100, 50, 0, 0))...
    }
    if (playerUnits.Count >= lastArmyCountForTower + 5)
    {
        if (HasResourcesForBuilding(100, 50, 0, 0))...
    }
    UpgradeMainBuildingIfNeeded();
    UpgradeWorkersIfNeeded();
    if (ShouldTrainArmy())...
    if (armyUnits.Count >= minimumArmySize)...
}
Ссылок: 1
void HandleAttack()
{
    foreach (var unit in armyUnits)
    {
        Warrior warriorComponent = unit.GetComponent<Warrior>();
        if (warriorComponent != null)
        {
            EngageClosestEnemyOrTarget(warriorComponent);
        }
    }
}
```

Рисунок 4.58 – Методи прийняття рішень ШІ

Рисунок 4.59 – Приклад роботи ШІ

Для виконання дій до ШІ було додано окремі методи дій саме для нього, які наслідують основні методи добування, будування і т.д. Також було додано до вже існуючих окремо ресурсів для ШІ та до методів використання перевірку на управління для ШІ. Всі методи ШІ прописані у додатку [Додаток Е].

Висновки до розділу 4

У четвертому розділі було описано процес розробки ігрового дизайну та ключових елементів користувацького інтерфейсу для стратегії в реальному часі (RTS). Основну увагу приділено створенню головного меню, панелі управління юнітами та будівлями, механіки збору ресурсів, будівництва споруд і бойових дій.

Також приділено увагу деталям візуалізації, таким як шкали здоров’я юнітів, систему анімацій, а також інтеграція звукових ефектів і музичного супроводу для створення захоплюючого ігрового досвіду.

Інтерфейс було створено за допомогою інструментів Unity UI, із використанням компонентів які забезпечують інтуїтивно зрозумілу взаємодію гравця з грою, гнучкість у налаштуваннях та комфортний ігровий процес.

ВИСНОВКИ

Під час виконання кваліфікаційної магістерської роботи проведено аналіз актуальності сфери комп’ютерних 3D-ігор, зокрема жанру стратегій у реальному часі (RTS). Були проаналізовані їх функціональні можливості, переваги та недоліки. Також було виконано системний аналіз для визначення цілей гри, користувачів та сценаріїв використання.

Крім того, було проаналізовано технології розробки 3D-ігор на платформі Unity, зокрема її можливості, переваги та особливості використання. Проведено огляд програм для моделювання 3D-об’єктів, порівняно доступні інструменти і визначено оптимальний варіант для реалізації проєкту.

У рамках роботи було реалізовано поставлене технічне завдання щодо створення ігрового застосунку. Для досягнення цієї мети виконано такі завдання:

– Проведено аналіз технологій штучного інтелекту у сфері ігрової індустрії.

– Здійснено огляд існуючих ігор жанру RTS.

– Визначено основні функції гри та оформлено специфікацію вимог до програмного забезпечення.

– Розроблено загальний алгоритм створення гри.

– Створено UML-діаграми сценаріїв використання, взаємодії, станів, діяльності, компонентів та розгортання.

– Реалізовано ігровий застосунок з використанням мови програмування C# та рушія Unity, впроваджено елементи штучного інтелекту для адаптивного ігрового процесу.

Результатом кваліфікаційної роботи є готовий прототип комп’ютерної 3D-ігри у жанрі RTS, що відповідає поставленим вимогам і демонструє використання штучного інтелекту. Цей застосунок може бути використаний як основа для подальшого розвитку та вдосконалення гри з перспективою комерційного запуску.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Unity wiki. URL: https://uk.wikipedia.org/wiki/Unity (дата звернення: 08.09.2024).

2. Методика розробки. URL: http://um.co.ua/8/8-7/8-7041.html (дата звернення: 08.09.2024).

3. Розробка ігор за допомогою UNITY3D. URL: https://ec.europa.eu/programmes/erasmus-plus/project-result-content/7c7d9761-4402-467c-adab-5b09579cb8fd/2017_KNTU_Module2.pdf(дата звернення 08.09.2024).

4. Artificial Intelligence for games. URL: https://habr.com/ru/companies/intel/articles/265679/ (дата звернення: 09.09.2024).

5. Опис аналога Cossacks 3. URL: https://store.steampowered.com/app/333420/Cossacks_3/(дата звернення: 10.09.2024).

6. Опис аналога Age of Empires II. URL: https://store.steampowered.com/app/813780/Age_of_Empires_II_Definitive_Edition/ (дата звернення 10.09.2024).

7. Опис аналога Stronghold Crusader 2. URL: https://store.steampowered.com/app/232890/Stronghold_Crusader_2/(дата звернення 10.09.2024).

8. Лекції по Unity. URL: https://unity.com/ru/learn/get-started(дата звернення: 11.09.2024).

9. C# документація. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 11.09.2024).

10. Visual Studio документація. URL: https://learn.microsoft.com/ru-ru/visualstudio/get-started/visual-studio-ide?view=vs-2022(дата звернення: 11.09.2024).

11. Unreal Engine. URL: https://vokigames.com/ua/unreal-engine(дата звернення: 12.09.2024).

12. CryEngine. URL: http://3das.com.ua/igrovij-dvizhok-vibrati-unity-udk-abo-cryengine/ (дата звернення: 12.09.2024).

13. Програма для моделювання 3д ігор - Cinema4D. URL : https://uk.wikipedia.org/wiki/Cinema_4D (дата звернення: 12.09.2024).

14. Програма для моделювання 3д ігор – Blender. URL: https://uk.wikipedia.org/wiki/Blender (дата звернення: 12.09.2024).

15. Написання Use Cases. URL: http://www.tsatu.edu.ua/kn/wp-content/uploads/sites/16/napysannja-usecases (дата звернення: 21.09.2024).

16. Конструювання UML-діаграм. URL: https://moodle3.chmnu.edu.ua/course/view.php?id=27767#section-6 (дата звернення: 22.09.2024)

17. Теоретичні відомості про UML-діаграми. URL: https://studfile.net/preview/5010027/page:4/#6 (дата звернення: 24.09.2024)

18. Creative Core: UI. URL: https://learn.unity.com/project/creative-core-ui?uv=6 (дата звернення: 10.10.2024)

19. NavMesh Agents. URL: https://learn.unity.com/tutorial/working-with-navmesh-agents?language=en (дата звернення: 17.10.2024)

20. Creative Core: Animations. URL: https://learn.unity.com/project/creative-core-animation?language=en (дата звернення: 24.10.2024)

21. Artificial Intelligence for Beginners. URL: https://learn.unity.com/course/artificial-intelligence-for-beginners?uv=2021.3 (дата звернення: 01.11.2024).

ДОДАТОК А

Скрипти управління юнітами

```
public class UnitController : MonoBehaviour
{
   private void OnGUI()
   {
      if (isSelecting)
      {
         selectionBox.x = Mathf.Min(startMousePosition.x, endMousePosition.x);
         selectionBox.y = Screen.height - Mathf.Max(startMousePosition.y, endMousePosition.y);
         selectionBox.width = Mathf.Abs(startMousePosition.x - endMousePosition.x);
         selectionBox.height = Mathf.Abs(startMousePosition.y - endMousePosition.y);
         GUI.color = selectionColor;
         GUI.DrawTexture(selectionBox, Texture2D.whiteTexture);
         GUI.color = Color.white;
      }
   }
   private void Update()
   {
      HandleUnitSelection();
      HandleUnitCommand();
   }
   private void HandleUnitSelection()
   {
      if (Input.GetMouseButtonDown(0))
      {
         if (EventSystem.current.IsPointerOverGameObject()){return;}
         Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
         RaycastHit hit;
         if (Physics.Raycast(ray, out hit)){
            GameObject clickedObject = hit.collider.gameObject;
            if (clickedObject.CompareTag("Unit")){
               ClearSelection();
               selectedUnits.Clear();
               SelectUnit(clickedObject);
               return;}
            if (clickedObject.CompareTag("Building")){
               ClearSelection();
               selectedUnits.Clear();
               SelectBuilding(clickedObject);
               return;}}
         isSelecting = true;
         startMousePosition = Input.mousePosition;
      }
      if (isSelecting){
         endMousePosition = Input.mousePosition;
         if (Input.GetMouseButtonUp(0)){
            ClearSelection();
            selectedUnits.Clear();
            SelectUnitsInBox();
            isSelecting = false;}}
   }
   private void HandleUnitCommand()
   {
      if (Input.GetMouseButtonDown(1))
      {
         Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
         RaycastHit hit;
         if (Physics.Raycast(ray, out hit)) {
            if (hit.collider.CompareTag("Wood") || hit.collider.CompareTag("Stone") || hit.collider.CompareTag("Gold") ||
hit.collider.CompareTag("Food")) {
               foreach (GameObject unit in selectedUnits) {
                  Worker worker = unit.GetComponent<Worker>();
                  if (worker != null){worker.MoveToResource(hit.collider.gameObject);}}
            }else if (hit.collider.CompareTag("Building")){
```

```
                Building building = hit.collider.GetComponent<Building>();
                if (building != null){
                    if (building.isBuilding) {
                        foreach (GameObject unit in selectedUnits) {
                            Worker worker = unit.GetComponent<Worker>();
                    if (worker != null){worker.buildingSystem.SendWorkerToConstruction (unit, building.gameObject);} } }
                    if(building.isBuilding || !building.isBuilding){
                        foreach (GameObject unit in selectedUnits) {
                            Warrior warrior = unit.GetComponent<Warrior>();
                            Worker worker = unit.GetComponent<Worker>();
                           if (unit.gameObject.layer == LayerMask.NameToLayer("Blue")){
                     if (hit.collider.gameObject.layer == LayerMask.NameToLayer("Red")){
                       if (warrior != null) warrior.SetTarget(hit.collider.gameObject);
                       if (worker != null) worker.SetTarget(hit.collider.gameObject);}}
                           if (unit.gameObject.layer == LayerMask.NameToLayer("Red")){
                     if (hit.collider.gameObject.layer == LayerMask.NameToLayer("Blue")){
                        if (warrior != null) warrior.SetTarget(hit.collider.gameObject);
                        if (worker != null) worker.SetTarget(hit.collider.gameObject);}}
                    }}}
                }else if (hit.collider.CompareTag("Unit")) {
                    foreach (GameObject unit in selectedUnits) {
                        Warrior warrior = unit.GetComponent<Warrior>();
                        Worker worker = unit.GetComponent<Worker>();
                        if(unit.gameObject.layer == LayerMask.NameToLayer("Blue")){
                        if (hit.collider.gameObject.layer == LayerMask.NameToLayer("Red")){
                          if (warrior != null)warrior.SetTarget(hit.collider.gameObject);
                         if (worker != null) worker.SetTarget(hit.collider.gameObject);}}
                        if (unit.gameObject.layer == LayerMask.NameToLayer("Red")){
                     if (hit.collider.gameObject.layer == LayerMask. NameToLayer("Blue")){
                          if (warrior != null)warrior.SetTarget(hit.collider.gameObject);
                         if (worker != null) worker.SetTarget(hit.collider.gameObject);}}}
                }else{MoveUnitsInFormation(hit.point);}
            }}}

    public void MoveUnitsInFormation(Vector3 destination)
    {
        if (selectedUnits.Count == 0) return;
        float formationSpacing = 2.0f;
        int unitsPerRow = Mathf.CeilToInt(Mathf.Sqrt(selectedUnits.Count));
        for (int i = 0; i < selectedUnits.Count; i++)
        {
            GameObject unit = selectedUnits[i];
            Worker worker = unit.GetComponent<Worker>();
            if (worker != null){
                worker.StopGathering();
                worker.StopFighting();
                if (worker.targetBuilding != null){ worker.StopConstruction();}}
            Warrior warrior = unit.GetComponent<Warrior>();
            if (warrior != null){warrior.StopFighting();}
            NavMeshAgent agent = unit.GetComponent<NavMeshAgent>();
            if (agent != null) {
                int row = i / unitsPerRow;
                int column = i % unitsPerRow;
                Vector3 offset = new Vector3(column * formationSpacing, 0, row * formationSpacing);
                agent.stoppingDistance = 0.5f;
                  NavMeshPath path = new NavMeshPath();
                if (agent.CalculatePath(destination + offset, path)){agent.SetPath(path);}}
        }
    }
```

ДОДАТОК Б

Скрипти системи добування ресурсів

```
public class Worker : MonoBehaviour
{
   private void Update()
   {
      if (targetResource != null && isSpotReserved) {
      float distance = Vector3.Distance(transform.position, gatherSpot);
      if (distance <= gatherRange && !isGathering){ StartGathering(); } }
      if (isGathering) {
      float distance = Vector3.Distance(transform.position, gatherSpot);
      if (distance > gatherRange) {
         isGathering = false;
         animatorController.CheckAnimation();
         agent.isStopped = false;
         CheckWeapon();
         agent.SetDestination(gatherSpot);
         Debug.Log($"проблема"); }
      gatherTimer -= Time.deltaTime;
      if (gatherTimer <= 0f){
         GatherResource();
         gatherTimer = gatherCooldown; } }
   }

   public void MoveToResource(GameObject resource)
   {
      StopGathering();
      targetResource = resource;
      isGathering = false;
      currentTask = ResourceType.None;
      ResourceHealth resourceHealth = resource.GetComponent<ResourceHealth>();
      if (resourceHealth != null && resourceHealth.TryGetNearestFreeSpot(transform.position, out gatherSpot)) {
         isSpotReserved = true;
         agent.stoppingDistance = 0.5f;
         NavMeshPath path = new NavMeshPath();
         if (agent.CalculatePath(gatherSpot, path)) { agent.SetPath(path); }
      }
      else {Debug.Log($"{name}: No points available for the resource {resource.name}.");}
   }
       private void StartGathering()
       {
          if (isGathering || agent.velocity != Vector3.zero) return;
          isGathering = true;
          ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
          if (resourceHealth != null) {
             currentTask = resourceHealth.resourceType;
             animatorController.CheckAnimation();
             gatherTimer = gatherCooldown;
             CheckWeapon();
             agent.isStopped = true;
             agent.transform.LookAt(targetResource.transform);}
       }
       private void GatherResource()
       {
          if (targetResource == null || !isGathering) return;
          ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
          if (resourceHealth != null) { resourceHealth.TakeDamage(gatherAmount); }
       }
       public void StopGathering()
       {
          if (isGathering || isSpotReserved) {
             ReleaseSpot();
             ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
             resourceHealth?.UpdateFreeSpots(); }
          currentTask = ResourceType.None;
```

```
            isGathering = false;
            animatorController.CheckAnimation();
            agent.isStopped = false;
            CheckWeapon();
            targetResource = null;
        }
        private void ReleaseSpot()
        {
            if (isSpotReserved && targetResource != null) {
                ResourceHealth resourceHealth = targetResource.GetComponent<ResourceHealth>();
                if (resourceHealth != null) { resourceHealth.ReleaseSpot(gatherSpot); }
                isSpotReserved = false; }
        }
}
```

```
public class ResourceHealth : MonoBehaviour
{
    private void GenerateGatherPoints()
    {
        for (int i = 0; i < maxGatherPoints; i++){
            float angle = i * Mathf.PI * 2 / maxGatherPoints;
            Vector3 pointPosition = transform.position + new Vector3(Mathf.Cos(angle), 0, Mathf.Sin(angle)) * gatherRadius;
            GatherPoint point = new GatherPoint {
                Position = pointPosition,
                IsOccupied = false };
            gatherPoints.Add(point); }
    }
    public bool TryGetNearestFreeSpot(Vector3 workerPosition, out Vector3 spot)
    {
        spot = Vector3.zero;
        float shortestDistance = float.MaxValue;
        GatherPoint nearestPoint = null;
        foreach (var point in gatherPoints) {
            if (!point.IsOccupied) {
                float distance = Vector3.Distance(workerPosition, point.Position);
                if (distance < shortestDistance){
                    shortestDistance = distance;
                    nearestPoint = point;} } }
        if (nearestPoint != null) {
            spot = nearestPoint.Position;
            nearestPoint.IsOccupied = true;
            return true; }
        return false;
    }
    public void ReleaseSpot(Vector3 spot)
    {
        foreach (var point in gatherPoints) {
            if (point.Position == spot) { point.IsOccupied = false; break; } }
    }
    public void TryClearInvalidReservations()
    {
        foreach (var point in gatherPoints) {
      if (point.IsOccupied && !IsWorkerNear(point.Position)) { point.IsOccupied = false;}}
    }
    private bool IsWorkerNear(Vector3 point)
    {
        Collider[] colliders = Physics.OverlapSphere(point, 1.0f);
        foreach (var col in colliders) {
            if (col.CompareTag("Unit") && col.GetComponent<Worker>()) { return true; }}
        return false;
    }
    public void UpdateFreeSpots()
    {
        TryClearInvalidReservations();
    }
    public bool IsDepleted => currentHealth <= 0;

    public void TakeDamage(float amount)
```

```
    {
        currentHealth -= amount;
        AddResourceToManager(resourceType, amount);
        if (currentHealth <= 0){
            AddResourceToManager(resourceType, resourceAmount);
            currentHealth = 0;
            FindNewResource();
            Debug.Log($"{resourceType} исчерпан!");
            UpdateFreeSpots();
            DestroyResource();  }
    }
    private void AddResourceToManager(ResourceType type, float amount)
    {
        switch (type) {
            case ResourceType.Wood:
                ResourceManager.Instance.AddResources(amount, 0, 0, 0); break;
            case ResourceType.Stone:
                ResourceManager.Instance.AddResources(0, amount, 0, 0); break;
            case ResourceType.Gold:
                ResourceManager.Instance.AddResources(0, 0, amount, 0); break;
            case ResourceType.Food:
                ResourceManager.Instance.AddResources(0, 0, 0, amount); break;}
    }
    private void NotifyWorkersToStop()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, gatherRadius);
        foreach (var col in colliders) {
            Worker worker = col.GetComponent<Worker>();
          if (worker != null && worker.targetResource == gameObject){worker.StopGathering();}}
    }
    private void FindNewResource()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, gatherRadius * 5);
        GameObject nearestResource = null;
        float shortestDistance = float.MaxValue;
        foreach (var col in colliders){
            ResourceHealth resource = col.GetComponent<ResourceHealth>();
        if (resource != null && resource != this && resource.resourceType == resourceType){
        float distance = Vector3.Distance(transform.position, resource.transform.position);
                if (distance < shortestDistance){
                    shortestDistance = distance;
                    nearestResource = resource.gameObject;}}}
        if (nearestResource != null){
    Collider[] workersColliders=Physics.OverlapSphere(transform.position, gatherRadius*1.5f);
            foreach (var col in workersColliders){
                Worker worker = col.GetComponent<Worker>();
                if (worker != null && worker.targetResource == gameObject){
                    worker.MoveToResource(nearestResource);}}}else NotifyWorkersToStop();
    }
    private void DestroyResource()
    {
        foreach (var point in gatherPoints){point.IsOccupied = false;}
        Destroy(gameObject);
    }
    private class GatherPoint
    {
        public Vector3 Position;
        public bool IsOccupied;
    }
}
```

ДОДАТОК В

Скрипти системи будування

```
public class BuildingSystem : MonoBehaviour
{
    public void StartBuildingMode(Worker.BuildingType type)
    {
        if (isBuildingMode) return;
        isBuildingMode = true; selectedBuildingType = type;
GameObject prefab = type == Worker.BuildingType.Tower ? towerPrefab : barracksPrefab;
        previewObject = Instantiate(prefab);
        SetPreviewMode(previewObject, true);worker.enabled = false;
        UnitController.Instance.enabled = false;}
    private void Update(){ if (isBuildingMode){HandleBuildingMode();}}
    private void HandleBuildingMode()
    {
        if (previewObject == null) return;
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (Physics.Raycast(ray, out RaycastHit hit)) {
            previewObject.transform.position = hit.point;
            float scrollInput = Input.GetAxis("Mouse ScrollWheel");
            if (scrollInput != 0){
                float rotationAngle = scrollInput > 0 ? 15f : -15f;
                previewObject.transform.Rotate(Vector3.up, rotationAngle, Space.World);}
            bool canBuild = CanPlaceBuilding();
            SetPreviewColor(previewObject, canBuild ? Color.green : Color.red);}
        if (Input.GetMouseButtonDown(0) && CanPlaceBuilding()){ConfirmBuildingPlacement();}
        if (Input.GetMouseButtonDown(1)){CancelBuildingMode();}}
    private bool CanPlaceBuilding()
    {
        if (previewObject == null) return false;
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (!Physics.Raycast(ray, out RaycastHit hit, Mathf.Infinity) || !hit.collider.CompareTag("Ground")){return false; }
        float checkInterval = 0.5f;
        List<Vector3> boundaryPoints = previewObject.GetComponent<Building>().GenerateBoundaryPoints(checkInterval);
        foreach (var point in boundaryPoints){
            Collider[] colliders = Physics.OverlapSphere(point, 0.5f);
            foreach (var detected in colliders){
    if (detected.gameObject!=previewObject&&(detected.CompareTag("Building")||detected.
CompareTag("Unit")||detected.gameObject.layer==LayerMask.NameToLayer("Resource"))){
                return false;}}}return true;}
    private void ConfirmBuildingPlacement()
    {
  GameObject prefab = selectedBuildingType == Worker.BuildingType.Tower ? towerPrefab : barracksPrefab; buildingToConstruct
= Instantiate(prefab, previewObject.transform.position, Quaternion.identity); SetPreviewMode(previewObject, false);
Destroy(previewObject);
  previewObject = null; buildingToConstruct.GetComponent<Building>().isBuilding = true;
        buildingToConstruct.GetComponent<Building>().GenerateConstructionPoints();
        if (selectedBuildingType == Worker.BuildingType.Tower){
    ResourceManager.Instance.UseResources(100, 50, 0, 0); }
        else if (selectedBuildingType == Worker.BuildingType.Barracks) {
            ResourceManager.Instance.UseResources(150, 100, 0, 0); }
        buildingToConstruct.GetComponent<Building>().currentHealth = 10;
        SendWorkerToConstruction(this.gameObject, buildingToConstruct);
        isBuildingMode = false; worker.enabled = true;
        UnitController.Instance.enabled = true;}
  private void CancelBuildingMode(){
        Destroy(previewObject); previewObject = null; isBuildingMode = false;
        worker.enabled = true;  UnitController.Instance.enabled = true; }
    public void SendWorkerToConstruction(GameObject workerObject, GameObject building){
        Worker worker = workerObject.GetComponent<Worker>();
        if (worker == null || building == null) return;
        Building buildingComponent = building.GetComponent<Building>();
        if (worker.isConstructionSpotReserved){
          StopWorkerConstruction(worker,  worker.targetBuilding.GetComponent<Building>());  }
        if (buildingComponent.isBuilding && buildingComponent.TryGetNearestFreeSpot( worker.transform.position, out Vector3
constructionSpot)) {
```

```
            worker.targetBuilding = building; worker.isConstructing = false;
            worker.isConstructionSpotReserved = true; worker.agent.stoppingDistance = 0.5f;
            NavMeshPath path = new NavMeshPath();
        if (worker.agent.CalculatePath(constructionSpot, path)) {worker.agent.SetPath(path);}
          worker.StartCoroutine(StartConstructionWhenArrived(worker, buildingComponent, constructionSpot));}
      }
   private IEnumerator StartConstructionWhenArrived(Worker worker, Building building, Vector3 constructionSpot)
      {
        while (Vector3.Distance(worker.transform.position, constructionSpot) > 0.7f){
if (!worker.isConstructionSpotReserved || worker.targetBuilding != building.gameObject){
              building.ReleaseSpot(constructionSpot);
             if (worker.isConstructing == true){worker.StopConstruction();}yield break; }
          yield return null;}
        while (worker.isConstructing || worker.agent.velocity != Vector3.zero){
         if (worker.isConstructing || worker.agent.velocity == Vector3.zero){yield break; }
          yield return null;}
        worker.agent.isStopped = true;
        worker.agent.transform.LookAt(building.gameObject.transform);
        worker.isConstructing = true;worker.currentTask = Worker.ResourceType.Build;
        worker.CheckWeapon();worker.animatorController.CheckAnimation();
        while (worker.isConstructing && building.isBuilding && building.currentHealth < building.maxHealth){
          yield return new WaitForSeconds(1.0f);building.currentHealth += 10;}
        if (building.currentHealth >= building.maxHealth){building.isBuilding = false;}
        StopWorkerConstruction(worker, building);
      }
      public void StopWorkerConstruction(Worker worker, Building building)
      {
        worker.isConstructing = false;worker.agent.isStopped = false;
        if (building != null && worker.isConstructionSpotReserved){
    building.ReleaseSpot(worker.agent.destination);building.UpdateConstructionPoints();
        worker.isConstructionSpotReserved = false;}
      }
      private void SetPreviewMode(GameObject obj, bool isPreview)
      {
        Renderer[] renderers = obj.GetComponentsInChildren<Renderer>();
        foreach (var renderer in renderers){
          renderer.material.color = isPreview ? new Color(0, 1, 0, 0.5f) : Color.white;}
        Collider[] colliders = obj.GetComponentsInChildren<Collider>();
        foreach (var collider in colliders){collider.enabled = !isPreview;}
        NavMeshObstacle obstacle = obj.GetComponentInChildren<NavMeshObstacle>();
        obstacle.enabled = !isPreview;
      }
      private void SetPreviewColor(GameObject obj, Color color)
      {
        Renderer[] renderers = obj.GetComponentsInChildren<Renderer>();
        foreach (var renderer in renderers){
          if (renderer is MeshRenderer meshRenderer){
            Material[] materials = meshRenderer.materials;
           for (int i = 0; i < materials.Length; i++){materials[i].color = color;}}else{
            if (renderer.material != null){
              renderer.material.color = color;}}}
}  }
public class Building : MonoBehaviour
{
       public void GenerateConstructionPoints()
       {
         constructionPoints.Clear();
         BoxCollider collider = GetComponent<BoxCollider>();
         if (collider == null){return;}
         Vector3 size = collider.size * transform.localScale.x;
         Vector3 center = transform.position;
         List<Vector3> boundaryCandidates = new List<Vector3>{
         center + new Vector3(-size.x / 2, 0, -size.z / 2),
         center + new Vector3(-size.x / 2, 0, size.z / 2),
         center + new Vector3(size.x / 2, 0, -size.z / 2),
         center + new Vector3(size.x / 2, 0, size.z / 2), };
         System.Random random = new System.Random();
         for (int i = 0; i < 3; i++){
```

```
            int randomIndex = random.Next(boundaryCandidates.Count);
            Vector3 point = boundaryCandidates[randomIndex];
            boundaryCandidates.RemoveAt(randomIndex);
          if (NavMesh.SamplePosition(point, out NavMeshHit hit, 1.0f, NavMesh.AllAreas)){
                constructionPoints.Add(new ConstructionPoint{
                   Position = hit.position, IsOccupied = false});}}
        }
        public List<Vector3> GenerateBoundaryPoints(float interval)
        {
           List<Vector3> boundaryPoints = new List<Vector3>();
           BoxCollider collider = GetComponent<BoxCollider>();
           if (collider == null){ return boundaryPoints;}
           Vector3 size = collider.size * transform.localScale.x;
           Vector3 center = transform.position;
           for (float x = -size.x / 2; x <= size.x / 2; x += interval){
              boundaryPoints.Add(center + new Vector3(x, 0, -size.z / 2));
              boundaryPoints.Add(center + new Vector3(x, 0, size.z / 2)); }
           for (float z = -size.z / 2; z <= size.z / 2; z += interval){
              boundaryPoints.Add(center + new Vector3(-size.x / 2, 0, z));
              boundaryPoints.Add(center + new Vector3(size.x / 2, 0, z)); }
           return boundaryPoints;
        }
        public bool TryGetNearestFreeSpot(Vector3 workerPosition, out Vector3 spot)
        {
           spot = Vector3.zero;float shortestDistance = float.MaxValue;
           ConstructionPoint nearestPoint = null;
           foreach (var point in constructionPoints){
              if (!point.IsOccupied){
                 float distance = Vector3.Distance(workerPosition, point.Position);
                 if (distance < shortestDistance){
                    shortestDistance = distance; nearestPoint = point;}}}
           if (nearestPoint != null){
              spot = nearestPoint.Position; nearestPoint.IsOccupied = true;
              return true;} return false;
        }
        public void ReleaseSpot(Vector3 spot) {
           foreach (var point in constructionPoints){
              if (point.Position == spot){point.IsOccupied = false; break;}}
        }
        public void TryClearInvalidConstructionPoints() {
           foreach (var point in constructionPoints){
         if(point.IsOccupied && !IsWorkerNear(point.Position)){point.IsOccupied=false;}}
        }
        private bool IsWorkerNear(Vector3 point) {
           Collider[] colliders = Physics.OverlapSphere(point, 1.0f);
           foreach (var col in colliders){
              if (col.CompareTag("Unit") && col.GetComponent<Worker>()?.targetBuilding == this.gameObject) {return
true;}}return false;
        }
        public void UpdateConstructionPoints() { TryClearInvalidConstructionPoints(); }
        private class ConstructionPoint
        {
           public Vector3 Position;
           public bool IsOccupied;
        }
}
```

ДОДАТОК Г

Скрипти системи атаки юнітів

```
public class Warrior : MonoBehaviour
{
    void Update()
    {
        if (currentTarget != null) {
            float distance = Vector3.Distance(transform.position, currentTarget.transform.position); bool isMoving =
agent.velocity.magnitude > 0.1f;
            if (distance <= attackRange) {
                agent.isStopped = true;
                if (!isMoving) { AttackTarget(); attackTimer -= Time.deltaTime;} }
            else{agent.isStopped = false; NavMeshPath path = new NavMeshPath();
                if (agent.CalculatePath(currentTarget.transform.position, path)){
                    agent.SetPath(path); }
    }
    public void SetTarget(GameObject target){ currentTarget = target; agent.SetDestination(currentTarget.transform.position);}
    public void StopFighting(){agent.isStopped = false;currentTarget = null;}
    private void AttackTarget()
    {
        agent.transform.LookAt(currentTarget.transform.position);
        if (attackTimer <= 0.0f && currentTarget != null){
            attackTimer = attackCooldown;
            Worker worker = currentTarget.GetComponent<Worker>();
            Warrior warrior = currentTarget.GetComponent<Warrior>();
            Building building = currentTarget.GetComponent<Building>();
if(worker!=null){worker.TakeDamage(attackDamage);worker.IsAttacked(this.gameObject);}
            if (warrior != null){
                warrior.TakeDamage(attackDamage);warrior.IsAttacked(this.gameObject);}
            if (building != null){building.TakeDamage(attackDamage);}}
    }
    public void TakeDamage(float damage)
    {
        health -= damage; healthBar.SetHealth(health, maxHealth);
        if (health <= 0){FindNewTarget();Die();}
    }
    private void FindNewTarget()
    {
    Collider[] colliders = Physics.OverlapSphere(transform.position, attackRange * 5);
        GameObject nearestEnemy = null;
        float shortestDistance = float.MaxValue;
        foreach (var col in colliders){
            Warrior warrior = col.GetComponent<Warrior>();
if(warrior!=null && warrior!=this && warrior.gameObject.layer==this.gameObject.layer){
    float distance = Vector3.Distance(transform.position, warrior.transform.position);
                if (distance < shortestDistance){
                    shortestDistance = distance;nearestEnemy = warrior.gameObject;}}
            Worker worker = col.GetComponent<Worker>();
if(warrior!=null && warrior!=this && warrior.gameObject.layer==this.gameObject.layer){
    float distance = Vector3.Distance(transform.position, warrior.transform.position);
                if (distance < shortestDistance){
                    shortestDistance = distance;nearestEnemy = warrior.gameObject;}}}
        if (nearestEnemy != null){
Collider[] unitColliders=Physics.OverlapSphere(transform.position, attackRange*1.5f);
            foreach (var col in unitColliders){
                Worker worker = col.GetComponent<Worker>();
                if (worker != null && worker.currentTarget == gameObject) {
                    worker.currentTarget = nearestEnemy;}
                Warrior warrior = col.GetComponent<Warrior>();
                if (warrior != null && warrior.currentTarget == gameObject) {
                    warrior.currentTarget = nearestEnemy; }}}
        else NotifyAttackersToStop();
    }
    private void NotifyAttackersToStop()
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, attackRange);
```

```
            foreach (var col in colliders)
            {
                Worker worker = col.GetComponent<Worker>();
                if (worker != null && worker.currentTarget == gameObject){
                    worker.agent.SetDestination(worker.gameObject.transform.position);
                    worker.StopFighting();}
                Warrior warrior = col.GetComponent<Warrior>();
                if (warrior != null && warrior.currentTarget == gameObject){
                    warrior.agent.SetDestination(warrior.gameObject.transform.position);
                    warrior.StopFighting();}}
        }
        public void IsAttacked(GameObject attacker)
        {
            if (currentTarget == null){
                bool isMoving = agent.velocity.magnitude > 0.1f;
                if (isMoving) { return; } currentTarget = attacker;}
            else{return;}
Collider[] nearbyColliders=Physics.OverlapSphere(transform.position, attackRange*4f);
            foreach (var col in nearbyColliders){
                Warrior nearbyWarrior = col.GetComponent<Warrior>();
                Worker nearbyWorker = col.GetComponent<Worker>();
                if (nearbyWarrior != null && nearbyWarrior != this && nearbyWarrior.gameObject.layer == gameObject.layer){
                    if (nearbyWarrior.currentTarget == null){
                        AssignNearestEnemy(attacker,  nearbyWarrior);}}
                if (nearbyWorker != null && nearbyWorker.currentTarget == null && nearbyWorker.gameObject.layer ==
gameObject.layer){
                    AssignNearestEnemy(attacker,  nearbyWorker);}}
        }
        private void AssignNearestEnemy(GameObject attacker, MonoBehaviour unit)
        {
            Collider[] potentialTargets = Physics.OverlapSphere(attacker.transform. position, attackRange * 4f);
            GameObject nearestEnemy = null;
            float shortestDistance = float.MaxValue;
            foreach (var targetCollider in potentialTargets){
                Warrior enemyWarrior = targetCollider.GetComponent<Warrior>();
                Worker enemyWorker = targetCollider.GetComponent<Worker>();
                Building enemyBuilding = targetCollider.GetComponent<Building>();
                if ((enemyWarrior != null || enemyWorker != null || enemyBuilding != null) && targetCollider.gameObject.layer !=
gameObject.layer){
                    float distance = Vector3.Distance(unit.transform.position, targetCollider.transform.position);
                    if (distance < shortestDistance){
                        shortestDistance = distance;
                        nearestEnemy = targetCollider.gameObject;}}}
            if (unit is Warrior warriorUnit){
                if (nearestEnemy != null){warriorUnit.SetTarget(nearestEnemy);}
                else{warriorUnit.SetTarget(attacker);}}
            else if (unit is Worker workerUnit){
                if (nearestEnemy != null){workerUnit.SetTarget(nearestEnemy);}
                else{workerUnit.SetTarget(attacker);}}
        }
        private void Die()
        {
            UnitController.Instance.ClearDieSelection(this.gameObject);
            Destroy(gameObject);
        }
}
```

ДОДАТОК Д

Скрипти управління камерою та міні-карти

```
public class CameraController : MonoBehaviour
{
   void Update()
   {
      Vector3 pos = transform.position;
      if (Input.GetKey("w")) pos.z += moveSpeed * Time.deltaTime;
      if (Input.GetKey("s")) pos.z -= moveSpeed * Time.deltaTime;
      if (Input.GetKey("d")) pos.x += moveSpeed * Time.deltaTime;
      if (Input.GetKey("a")) pos.x -= moveSpeed * Time.deltaTime;
      pos.x = Mathf.Clamp(pos.x, -panLimitM.x, panLimit.x);
      pos.z = Mathf.Clamp(pos.z, -panLimitM.y, panLimit.y);
      if (!isBuilding) {
         float scroll = Input.GetAxis("Mouse ScrollWheel");
         Camera.main.orthographicSize -= scroll * zoomSpeed;
         Camera.main.orthographicSize = Mathf.Clamp(Camera.main.orthographicSize, minZoom, maxZoom); }transform.position =
pos;
   }
}
public class MinimapCameraController : MonoBehaviour
{
   void Update()
   {
      UpdateCameraViewRect();
      HandleMinimapClick();
      ClampMainCameraIcon();
   }
   private void UpdateCameraViewRect()
   {
Vector3 viewportPosition=minimapCamera.WorldToViewportPoint(mainCamera.transform.position);
      if (mainCameraViewRect != null){
   mainCameraViewRect.anchorMin = viewportPosition - new Vector3(panLimit.x, panLimit.y);
 mainCameraViewRect.anchorMax = viewportPosition + new Vector3(panLimitM.x, panLimitM.y);}
      minimapCamera.orthographicSize = mainCamera.orthographicSize * 6;
}
   private void ClampMainCameraIcon()
   {
      Vector3 viewportPosition = minimapCamera.WorldToViewportPoint(mainCamera.transform.position);
      viewportPosition.x = Mathf.Clamp(viewportPosition.x, 0f + paddingTLR.x / minimapUI.rect.width, 1f - paddingTLR.y /
minimapUI.rect.width);
      viewportPosition.y = Mathf.Clamp(viewportPosition.y, 0f + paddingBLR.x / minimapUI.rect.height, 1f - paddingBLR.y /
minimapUI.rect.height);
      if (mainCameraViewRect != null){
    mainCameraViewRect.anchorMin = viewportPosition - new Vector3(panLimit.x, panLimit.y);
 mainCameraViewRect.anchorMax = viewportPosition + new Vector3(panLimitM.x, panLimitM.y);}
   }
   private void HandleMinimapClick()
   {
      if (Input.GetMouseButtonDown(0)){ Vector2 mousePosition = Input.mousePosition;
  if (!RectTransformUtility.RectangleContainsScreenPoint(minimapUI, mousePosition, null)){
            return;}
         if (RectTransformUtility.ScreenPointToLocalPointInRectangle(minimapUI, mousePosition, null, out Vector2 localPoint)){
            Vector2 normalizedPoint = new Vector2(
               (localPoint.x / minimapUI.rect.width) + 0.5f,
               (localPoint.y / minimapUI.rect.height) + 0.5f);
            Ray ray = minimapCamera.ViewportPointToRay(normalizedPoint);
            if (Physics.Raycast(ray, out RaycastHit hit)){
               Vector3 targetPosition = hit.point + mainCameraOffset;
               targetPosition.y = cameraHeight; targetPosition.z -= 50;
               mainCamera.transform.position = targetPosition;}}}
   }
}
```

ДОДАТОК Е

Скрипти механіки противника (ШІ)

```
public class AIController : MonoBehaviour
{
   private enum AIState{Expanding, Attacking}
   void Update()
   {
      if (Time.time >= nextDecisionTime) { MakeDecision();
         nextDecisionTime = Time.time + decisionInterval;}}
   void GatherGameData()
   {
      CheckResources();
      GameObject playerBase = GameObject.FindGameObjectWithTag("Building");
      if (playerBase != null && playerBase.GetComponent<Building>().type == Building.BuildingType.Main &&
playerBase.GetComponent<Building>().gameObject.layer == LayerMask.NameToLayer("Blue")){ playerBaseLocation =
playerBase.transform;}
   }
   void MakeDecision()
   {
      UpdateResourcePriority();
      switch (currentState){
         case AIState.Expanding:
            HandleExpansion(); break;
         case AIState.Attacking:
            HandleAttack(); break; }}
   void UpdateResourcePriority() {if (workers.Count >= 3 && !hasBuiltBarracks) {
         prioritizedResource = "Wood"; prioritizedResource2 = "Stone"; }
      else if (playerUnits.Count >= lastArmyCountForTower + 1) {
         prioritizedResource = "Wood"; prioritizedResource2 = "Stone"; }
      else { prioritizedResource = "Wood"; prioritizedResource2 = "Food"; }
   }
   void HandleExpansion()
   {
      if (NeedsMoreWorkers()){ SpawnWorker();}
      if (workers.Count >= 3 && !hasBuiltBarracks){
         if (HasResourcesForBuilding(100, 50, 0, 0)) {
 BuildStructure(barracksPrefab, buildPositionBarrack);
ResourceManager.Instance.UseAIResources(100, 50, 0, 0);hasBuiltBarracks = true;}}
      if (playerUnits.Count >= lastArmyCountForTower + 1){
         if (HasResourcesForBuilding(100, 50, 0, 0))  {
            BuildStructure(towerPrefab, buildPositionsTawer);
            ResourceManager.Instance.UseAIResources(100, 50, 0, 0);
            lastArmyCountForTower = playerUnits.Count; }}
      UpgradeMainBuildingIfNeeded();UpgradeWorkersIfNeeded();
      if (ShouldTrainArmy()){ TrainArmy();}
      if (armyUnits.Count >= minimumArmySize){currentState = AIState.Attacking;}
   }
       void HandleAttack()
       {
          foreach (var unit in armyUnits){
             Warrior warriorComponent = unit.GetComponent<Warrior>();
          if (warriorComponent != null){EngageClosestEnemyOrTarget(warriorComponent);}}
       }
       bool NeedsMoreWorkers(){return workers.Count < 10;}
       void SpawnWorker()
       {
          if (baseLocation == null) return;
          Building mainBuilding = baseLocation.GetComponent<Building>();
          if (mainBuilding != null && mainBuilding.type == Building.BuildingType.Main){
             mainBuilding.SpawnWorker();}}
       void CheckWorkerExist()
       {
          Worker[] allWorkers = FindObjectsOfType<Worker>();
         foreach (Worker worker in allWorkers) {if (worker != null && worker.gameObject.layer ==
LayerMask.NameToLayer("Red")) { if (!workers.Contains(worker.gameObject)){
workers.Add(worker.gameObject); Debug.Log($"Worker{worker.name}added to the list.");}}}}
```

```
    }
    void CheckWarriorExist()
    {
        Warrior[] allWarriors = FindObjectsOfType<Warrior>();
        foreach (Warrior warrior in allWarriors){
         if (warrior != null && warrior.gameObject.layer == LayerMask.NameToLayer("Red")){
        if (!armyUnits.Contains(warrior.gameObject)){armyUnits.Add(warrior.gameObject);}}}}
    public void CheckResources()
    {
        resourcePoints.Clear();
        foreach (Collider resource in Physics.OverlapSphere(transform.position, 1000f,
resourceLayer)){resourcePoints.Add(resource.transform);}}
    void AssignWorkersToResources()
    {
        foreach (var worker in workers){
            Worker workerComponent = worker.GetComponent<Worker>();
            if (workerComponent != null && workerComponent.targetResource == null){
            if (workerComponent.isConstructing == true || workerComponent.isConstructionSpotReserved == true) { break;}
                Transform resourceToAssign = FindNextResource(workerComponent.transform);
if(resourceToAssign!=null){workerComponent.MoveToResource(resourceToAssign.gameObject);}}}
    }
    Transform FindNextResource(Transform workerTransform){
        List<Transform> nearbyResources = Physics.OverlapSphere(workerTransform.position, 2000f, resourceLayer).Select(c =>
c.transform).Where(r => !r.GetComponent<ResourceHealth>().IsDepleted).ToList();
        if (nearbyResources.Count == 0) return null;
        string resourceTag = resourceAssignmentIndex % 2 == 0 ? prioritizedResource : prioritizedResource2;Transform
selectedResource = nearbyResources.FirstOrDefault(r => r.CompareTag(resourceTag));
        if (selectedResource != null){resourceAssignmentIndex++;return selectedResource;}
        foreach (var resource in nearbyResources){
            ResourceHealth resourceHealth = resource.GetComponent<ResourceHealth>();
            if (resourceHealth != null && !resourceHealth.IsDepleted && resourceHealth.HasFreeSpots()){return resource;}}return
nearbyResources.FirstOrDefault();
        }
        void ReassignIdleWorkers()
        {
            foreach (var worker in workers){
                Worker workerComponent = worker.GetComponent<Worker>();
                if (workerComponent != null){ if (workerComponent.isConstructing == true ||
workerComponent.isConstructionSpotReserved == true){break;}
                    if (workerComponent.targetResource == null){
                        Transform newTarget = FindNextResource(workerComponent.transform);
                if (newTarget != null){workerComponent.MoveToResource(newTarget.gameObject);}}
                    else if (workerComponent.agent.velocity.magnitude < 0.1f && workerComponent.currentTask ==
ResourceType.None){
                        workerComponent.MoveToResource(workerComponent.targetResource);}
                    else if (!workerComponent.targetResource.CompareTag(prioritizedResource) &&
!workerComponent.targetResource.CompareTag(prioritizedResource2)){
                        Transform newTarget = FindNextResource(workerComponent.transform);
                if (newTarget != null){workerComponent.MoveToResource(newTarget.gameObject);}}}}
        }
        void AssignWorkersToNeededResources(float requiredWood, float requiredStone, float requiredGold, float requiredFood)
            {
                foreach (var worker in workers){
                    Worker workerComponent = worker.GetComponent<Worker>();
                    if (workerComponent != null && workerComponent.targetResource == null){
                        if (requiredWood > 0){
                            Transform woodResource = FindResourceOrNearby("Wood");
                 if (woodResource != null){workerComponent.MoveToResource(woodResource.gameObject);
                            requiredWood -= 10; continue;} }
                        if (requiredStone > 0){
                            Transform stoneResource = FindResourceOrNearby("Stone");
                            if (stoneResource != null){
                                workerComponent.MoveToResource(stoneResource.gameObject);
                                requiredStone -= 10;continue;}}
                        if (requiredFood > 0){
                            Transform foodResource = FindResourceOrNearby("Food");
                            if (foodResource != null){
                                workerComponent.MoveToResource(foodResource.gameObject);
                                requiredFood -= 10;continue;}}}}}
```

```
    Transform FindResourceOrNearby(string tag)
    {
        Collider[] colliders = Physics.OverlapSphere(transform.position, 1000f);
        GameObject nearestResource = null;float shortestDistance = float.MaxValue;
        foreach (var col in colliders){
            ResourceHealth resource = col.GetComponent<ResourceHealth>();
            if (resource != null && resource != this){
                if (resource.gameObject.CompareTag(tag)){
    float distance = Vector3.Distance(transform.position, resource.transform.position);
                    if (distance < shortestDistance){shortestDistance = distance;
                        nearestResource = resource.gameObject;}}}}
    Transform primaryResource = nearestResource?.transform;
        if (primaryResource != null && primaryResource.GetComponent<ResourceHealth>(). HasFreeSpots()){return
primaryResource;}return null;}
    bool HasResourcesForBuilding(float wood, float stone, float gold, float food){
        return ResourceManager.Instance.HasEnoughAIResources(wood, stone, gold, food);}
    bool NeedsMoreBuildings(){return buildings.Count < desiredBuildingCount;}
    void BuildStructure(GameObject buildingPrefab, GameObject[] points)
    {
        foreach (var worker in workers){
            Worker workerComponent = worker.GetComponent<Worker>();
            if (workerComponent != null && !workerComponent.isConstructing){
                Building buildingComponent = buildingPrefab.GetComponent<Building>();
                if (buildingComponent != null && buildingComponent.type == Building.BuildingType.Barracks){Vector3 buildPosition =
points[0].transform.position;
                GameObject buildingToConstruct = Instantiate(buildingPrefab, buildPosition,
Quaternion.identity);buildingToConstruct.GetComponent<Building>().isBuilding = true;
                buildingToConstruct.GetComponent<Building>().GenerateConstructionPoints();
                    workerComponent.StopGathering();
                    if (buildingToConstruct.GetComponent<Building>().isBuilding &&
buildingToConstruct.GetComponent<Building>().TryGetNearestFreeSpot(workerComponent.transform.position, out Vector3
constructionSpot)){workerComponent.buildingSystem. SendWorkerToConstruction(workerComponent.gameObject,
buildingToConstruct);
workerComponent.StartCoroutine(workerComponent.buildingSystem.StartConstructionWhenArrived(workerComponent,
buildingToConstruct.GetComponent<Building>(), constructionSpot));}
                    buildings.Add(buildingToConstruct);break;}}}}  }
    void UpgradeMainBuildingIfNeeded()
    {
        Building mainBuilding = baseLocation.GetComponent<Building>();
        if (mainBuilding != null && mainBuilding.type == Building.BuildingType.Main && mainBuilding.level <
2){if(HasResourcesForBuilding(100, 50, 0, 0)){mainBuilding.Upgrade();}
            else{AssignWorkersToNeededResources(100, 50, 0, 0);}}}
    void UpgradeWorkersIfNeeded()
    {
        Building mainBuilding = baseLocation.GetComponent<Building>();
        if (mainBuilding != null && mainBuilding.type == Building.BuildingType.Main && mainBuilding.level >= 2){foreach (var
worker in workers){
                Worker workerComponent = worker.GetComponent<Worker>();
                if (workerComponent != null && workerComponent.level < mainBuilding.level){
                    if (HasResourcesForBuilding(50, 0, 0, 50)){mainBuilding.UpgradeWorker();
                        ResourceManager.Instance.UseAIResources(50, 0, 0, 50);}
                    else{AssignWorkersToNeededResources(50, 0, 0, 50);}} }}}
    bool ShouldTrainArmy() { return armyUnits.Count < minimumArmySize; }
    void TrainArmy()
    {
        foreach (var building in buildings){
            Building buildingComponent = building.GetComponent<Building>();
            if (buildingComponent != null && buildingComponent.type == Building.BuildingType.Barracks)
{buildingComponent.SpawnWarrior();break;}}
    }
}
```