Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота бакалавра

на тему: «Розробка гри у жанрі виживання на платформі Unity з використанням штучного інтелекту»

Виконав: студент групи ІПЗ20-2

Спеціальність 121 «Інженерія програмного забезпечення»

Редкач Ростислав Юрійович
(прізвище та ініціали)

Керівник к.т.н., доцент Чупілко Т.А.
(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент Університет митної справи та фінансів,
(місце роботи)
доцент кафедри кібербезпеки та інформаційних технологій
(посада)
к.т.н. Савченко Ю.В.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2024

АНОТАЦІЯ

Редкач Р.Ю. Розробка гри у жанрі виживання на платформі Unity з використанням штучного інтелекту.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення». – Університет митної справи та фінансів, Дніпро, 2024.

Кваліфікаційна робота присвячена розробці гри у жанрі виживання на платформі Unity з використанням штучного інтелекту (ШІ). Основною метою дослідження є створення інтерактивної гри, яка забезпечує високу ступінь реалістичності та захопливості ігрового процесу завдяки впровадженню ШІ.

У роботі розглядаються основні етапи розробки гри, починаючи з концептуального дизайну і закінчуючи реалізацією ігрових механік. Описується вибір платформи Unity як основного інструменту для розробки, а також її переваги для створення 3D-графіки та інтеграції ШІ. Особлива увага приділена алгоритмам ШІ, що використовуються для забезпечення адаптивної поведінки ігрових персонажів, зокрема для системи ворогів та NPC (неігрових персонажів).

Дослідження включає аналіз існуючих рішень у жанрі виживання, вивчення їх сильних і слабких сторін, а також опис створення унікальних ігрових механік, які підвищують глибину ігрового досвіду. Розроблені системи включають генерацію випадкових подій, управління ресурсами, а також моделювання поведінки персонажів на основі теорії ігор та машинного навчання.

У підсумку, кваліфікаційна робота демонструє комплексний підхід до розробки ігрового продукту з використанням сучасних технологій ШІ, що забезпечує інноваційність та конкурентоспроможність створеної гри.

Ключові слова: розробка гри, штучний інтелект, Unity, ігрові механіки, процедурна генерація.

ABSTRACT

Redkach R.Y. Development of a survival game on the Unity platform using artificial intelligence.

Diploma work for bachelor's degree in speciality 121 "Software Engineering." - University of Customs and Finance, Dnipro, 2024.

The thesis is devoted to the development of a survival game on the Unity platform using artificial intelligence (AI). The main goal of the study is to create an interactive game that provides a high degree of realism and engagement of the gameplay through the introduction of AI.

The paper discusses the main stages of game development, starting with conceptual design and ending with the implementation of game mechanics. The paper describes the choice of the Unity platform as the main development tool, as well as its advantages for creating 3D graphics and integrating AI. Particular attention is paid to AI algorithms used to ensure adaptive behaviour of game characters, in particular for the enemy system and NPCs (non-player characters).

The research includes an analysis of existing solutions in the survival genre, a study of their strengths and weaknesses, and a description of the creation of unique game mechanics that increase the depth of the gaming experience. The developed systems include random event generation, resource management, and character behaviour modelling based on game theory and machine learning.

In summary, the thesis demonstrates an integrated approach to the development of a game product using modern AI technologies, which ensures the innovation and competitiveness of the created game.

Keywords: game development, artificial intelligence, Unity, game mechanics, procedural generation.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасний ігровий ринок демонструє великий інтерес до ігор у жанрі виживання, які пропонують гравцям захопливий та інтерактивний досвід. Популярність цих ігор пояснюється їх можливістю комбінувати стратегічний план, дослідження та екшен.

Штучний інтелект (ШІ) є ключовим для успішних ігор, забезпечуючи реалістичну поведінку персонажів та динамічність ігрових ситуацій. Використання передових технологій ШІ покращує якість гри та підвищує її конкурентоспроможність на ринку.

Метою даної кваліфікаційної роботи є розробка гри у жанрі виживання на платформі Unity з використанням сучасних технологій ШІ. Це передбачає створення інтерактивної та адаптивної ігрової середовища, яка забезпечить гравцям реалістичний та захопливий досвід.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Провести аналіз сучасних ігор у жанрі виживання та визначити їх сильні і слабкі сторони.

2. Розробити концептуальний дизайн гри, включаючи ігрові механіки та сценарії.

3. Вибрати платформу Unity як основний інструмент для розробки гри та обґрунтувати цей вибір.

4. Реалізувати ігровий світ з використанням 3D-графіки та процедурної генерації.

5. Розробити та інтегрувати системи ШІ для управління поведінкою ворогів та NPC.

Об'єктом дослідження є процес розробки комп'ютерних ігор із застосуванням технологій штучного інтелекту.

Предметом дослідження є методи та засоби застосування штучного інтелекту для створення ігрових механік, поведінки персонажів та процедурної генерації контенту в середовищі Unity.

Практичне значення одержаних результатів полягає у підвищенні якості ігор та конкурентоспроможності на ринку.

Сучасний рівень розв'язання завдання. На сьогоднішній день існує багато успішних проектів у жанрі виживання, таких як "The Forest", "Subnautica" та "Don't Starve". Ці ігри використовують різні підходи до реалізації ШІ та створення ігрового світу, що забезпечує унікальний ігровий досвід. Однак, більшість існуючих рішень мають обмежені можливості через використання застарілих алгоритмів ШІ або недостатню адаптивність. Використання платформи Unity дозволяє розробникам впроваджувати сучасні алгоритми та методи ШІ, що забезпечує більшу гнучкість та можливості для інновацій.

Основні технічні характеристики продукту

Розроблена гра буде мати наступні технічні характеристики:

- Платформа: Unity
- Графіка: 3D
- Алгоритми ШІ: поведінкові дерева, машинне навчання, теорія ігор
- Ігрові механіки: процедурна генерація світу, адаптивна поведінка ворогів та NPC, управління ресурсами

Очікуваний технічно-економічний ефект

Впровадження передових технологій ШІ у розробку гри підвищить її якість і привабливість, що сприятиме збільшенню комерційного успіху проекту. Використання платформи Unity забезпечить ефективну реалізацію ігрових механік та знизить витрати на розробку завдяки великій кількості готових модулів та активній підтримці спільноти розробників.

Структура роботи:

Кваліфікаційна робота складається з трьох розділів. Перший розділ присвячений теоретичному аналізу жанру виживання та ролі ШІ у сучасних іграх. У другому розділі описується процес вибору платформи та інструментів розробки. Третій розділ детально розглядає розробку ігрових механік та систем ШІ. Висновки підсумовують результати роботи та окреслюють перспективи подальших досліджень у цій галузі.

В результаті виконання кваліфікаційної роботи було набуто фахові компетентності та підтверджені наступні результати навчання, які відповідають Освітньо-Професійній програмі за спеціальністю 121 «Інженерія програмного забезпечення»:

1. Здатність ідентифікувати, класифікувати та формулювати вимоги до програмного забезпечення.
2. Здатність розробляти архітектури, модулі та компоненти програмних систем.
3. Здатність формулювати та забезпечувати вимоги щодо якості програмного забезпечення у відповідності з вимогами замовника, технічним завданням та стандартами.
4. Здатність дотримуватися специфікацій, стандартів, правил і рекомендацій в професійній галузі при реалізації процесів життєвого циклу.
5. Здатність застосовувати фундаментальні і міждисциплінарні знання для успішного розв'язання завдань інженерії програмного забезпечення.
6. Здатність реалізовувати фази та ітерації життєвого циклу програмних систем та інформаційних технологій на основі відповідних моделей і підходів розробки програмного забезпечення.
7. Здатність обґрунтовано обирати та освоювати інструментарій з розробки та супроводження програмного забезпечення.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аналіз предметної області

Індустрія відеоігор є однією з найбільш динамічних і швидко розвиваючих галузей сучасного розважального сектору. Відеоігри еволюціонували від простих двовимірних платформерів до складних тривимірних симуляцій з високим рівнем деталізації та реалізму.

Для розуміння комп'ютерних ігор необхідно спершу зрозуміти, що собою представляють ігри в цілому. Що означає термін «гра»? Гра - це вид осмисленої, непродуктивної діяльності, де мотив лежить не у результаті, а в самому процесі. Термін «гра» також використовується для позначення набору предметів або програм, призначених для такої діяльності.

Проте, таке визначення є занадто вузьким. Воно стосується лише дитячих розваг, але не охоплює всі прояви гри. Давайте розглянемо це детальніше. Це визначення базується на двох ключових аспектах:

1. Гра є непродуктивною діяльністю (не приносить матеріальної користі).
2. Гра спрямована на саму себе, а не на досягнення результатів (не є засобом досягнення цілей, що протиставляє її терміну «праця»).

Але чи справді це так? Розглянемо це уважніше. Гра - це непродуктивна діяльність? За бажанням будь-яку продуктивну діяльність можна перетворити на гру. Наприклад, збір врожаю, що є типовою продуктивною діяльністю, можна зробити ігровим конкурсом, запропонувавши учасникам змагатися між собою. Таким чином, обидва аспекти, на яких базується визначення гри, виявляються неспроможними.

Як бачимо, відділення гри від продуктивної діяльності є досить умовним. Виходить, що визначення гри втрачає свою чіткість при практичному застосуванні. Переходимо до поглядів відомого фахівця у цій галузі.

1.2. Закономірності комп'ютерних ігор

Перераховуючи основні властивості гри, можна також побачити основні напрями жанрів комп'ютерних ігор:

- Казуальні ігри: Задоволення від процесу гри.
- Екшн: Суперництво і напруга.
- Стратегії: Чіткі правила і планування.
- Відкритий світ: Імпровізація і дослідження.
- Рольові ігри: Зміна ролей і занурення в інші світи.
- Симулятори: Відтворення реальних життєвих ситуацій.

Таким чином, властивості гри та їх прояви у різних жанрах комп'ютерних ігор показують, наскільки різноманітними можуть бути ігровий досвід, і як важливо розуміти основи гри для створення цікавих і захоплюючих ігрових проектів.

Залежно від цілей і напрямів ігри можна поділити на:

- ігри на майстерність;
- азартні ігри;
- логічні ігри;
- навчальні ігри.

Ігри на майстерність, як ігри-симулятори видів спорту, військові ігри, клавіатурні тренажери, тетріс та інші, засновані на управлінні ігровими об'єктами, все залежить від спритності гравця. В азартних же, все навпаки, результат гри

залежить від випадковості, ймовірності. Яскравим прикладом азартних ігор є карткові ігри та імітаційні, як кістки, рулетка.

Логічні ігри, наприклад, шахи, шашки або хрестики-нулики, містять стратегію, тактику поведінки гравця, що впливає на результат. В іграх на навчання гравцеві пропонується стати учнем і отримати деякі знання. Останні займають провідну роль у світі комп'ютерних ігор.

За способами реалізації ігри бувають:

- ігри з режимом реального часу;
- кінцеві та нескінченні;
- з випадковими подіями або детерміновані;
- для одного, для багатьох гравців;
- ігри з різними рівнями складності.

Так само можна виділити три основні ієрархічні рівні, які дають змогу побудувати схему гри:

- оперативний рівень (локальні мережі, що ускладнюють ігри);
- тактичний рівень (поточне керування клавішами);
- стратегічний рівень (кінець гри, фіксація результатів).

На першому, оперативному рівні відбувається зміна об'єктів в ігровому просторі за допомогою натискання клавіш клавіатури, миші або іншого керуючого пристрою (джойстик). У результаті на екрані дисплея користувачеві мають бути відображені всі переміщення та зміни.

Тактичний рівень включає в себе оперативний. Усі дії на цьому рівні ведуть до досягнення деякої певної мети. На цьому етапі відбувається зміна рівня складності, темпу гри.

Стратегічний рівень містить у собі тактичний із додаванням кількох самостійних блоків, таких як введення на ігровий простір усіх об'єктів для

визначення завдання та візуалізації їхніх початкових параметрів. Крім цього, на цьому рівні відбувається перевірка критеріїв закінчення гри, фіксації та візуалізації результатів усієї гри загалом і підбиття підсумків минулих ігор.

Ігри можуть належати одній платформі або бути мультиплатформеними.

Класифікація за платформами:

- персональний Комп'ютер;
- ігрові консолі;
- мобільні пристрої.

Також за кількістю платформ, на яких може запуститися гра:

- мультиплатформні - ігри, здатні запуститися на кількох платформах;
- одноплатформні - ексклюзивні ігри, створені тільки для однієї певної платформи.

1.3. Виникнення ігрових жанрів

Комп'ютерні ігри зародилися в 50-х роках минулого століття (понад 60 років тому), а ігрові жанри з'явилися лише 20 років тому.

На початковому етапі розвитку ігрової індустрії кожна гра була унікальна і неповторна. Але з часом ігор ставало все більше і більше. Для того, щоб орієнтуватися у всьому різноманітті ігор, люди стали розділяти гри на різні категорії.

Критерії поділу на категорії були найрізноманітніші: кількість гравців, вид ігрової камери. Але більшу популярність отримав спосіб поділу на ігрові жанри.

Жанри ігор формувалися безструктурно і інтуїтивно протягом тривалого часу. Розробники ігор проводили сміливі експерименти, створюючи нові ігрові механіки. Невдалі експерименти забувалися, а вдалі ігри ставали прикладом для інших розробників. Розробники копіювали популярну ігрову механіку, додаючи

трохи ідей від себе, таким чином навколо найбільш популярних ігор утворювалися цілі класи схожих між собою ігор, ці класи і стали називати ігровими жанрами.

Найпоширенішими жанрами комп'ютерних ігор на даний момент є:

- екшен - жанр комп'ютерних ігор, у якій основний акцент робиться на боротьбу, також включає в себе проходження певних рівнів для досягнення битви з босами;
- пригоди - це жанр ігор, де гравець стикається з інтерактивною історією. Основний акцент в іграх цього жанру зроблено на розповідь і дослідження світу, а ключову роль в ігровому процесі відіграє розв'язання головоломок і завдань;
- рольова гра - жанр комп'ютерних ігор, заснований на елементах ігрового процесу традиційних настільних рольових ігор. У рольовій грі гравець керує одним або кількома персонажами, кожен з яких описаний набором чисельних характеристик, списком здібностей і вмінь; прикладами таких характеристик можуть бути очки здоров'я, показники сили, спритності, інтелекту, захисту, ухилення, рівень розвитку тієї чи іншої навички тощо;
- симулятори - це ігри, завдання яких полягає в імітації управління будь-яким процесом, апаратом або транспортним засобом;
- стратегії - це жанр комп'ютерних ігор, де для досягнення поставлених цілей гравець змушений застосовувати стратегічне мислення, і воно протиставлено швидким діям і реакції, які, як правило, не обов'язкові для успіху в таких іграх;

Жанр виживання, зокрема, отримав значне визнання серед гравців завдяки своїй здатності поєднувати елементи дослідження, управління ресурсами, стратегії та екшену в одному захоплюючому пакеті.

Ігри у жанрі виживання зазвичай включають такі особливості:

- Складні ігрові механіки: Гравці повинні виживати у ворожих умовах, збирати ресурси, будувати укриття, створювати зброю та інші необхідні предмети.

- Динамічне середовище: Ігровий світ часто змінюється у відповідь на дії гравця, включаючи погодні умови, появу ворогів та інші випадкові події.

- Висока ступінь інтерактивності: Взаємодія з навколишнім середовищем і іншими персонажами (як гравцями, так і NPC) відіграє ключову роль.

- Використання штучного інтелекту: ШІ забезпечує реалістичну поведінку ворогів та NPC, які реагують на дії гравця та створюють динамічні ігрові ситуації.

1.4. Аналіз та обґрунтування актуальності розробки

Розробка гри у жанрі виживання є надзвичайно актуальною з наступних причин:

1. Популярність жанру: Ігри виживання, такі як "The Forest", "Subnautica" та "Don't Starve", демонструють високий попит серед гравців. Їхні успіхи на ринку свідчать про те, що жанр продовжує привертати увагу широкої аудиторії.

2. Інноваційні можливості: Сучасні методи ШІ дозволяють розробникам створювати більш реалістичні та інтерактивні ігрові середовища. Використання ШІ для створення адаптивної поведінки ворогів та NPC значно підвищує рівень занурення гравця у гру.

3. Технологічні досягнення: Платформа Unity є однією з найбільш використовуваних та універсальних платформ для розробки ігор. Вона пропонує потужні інструменти для створення 3D-графіки та інтеграції ШІ, що робить її ідеальним вибором для реалізації складних ігрових проектів.

Необхідність виконання кваліфікаційної роботи. Виконання даної кваліфікаційної роботи обумовлене кількома важливими потребами:

- Освітня необхідність: Розробка складного ігрового проекту дозволила поглибити знання та практичні навички у галузі програмування, графіки та штучного інтелекту.

- Індустріальна потреба: Створення інноваційного продукту, що відповідає сучасним вимогам ринку, сприятиме розвитку галузі та підвищенню конкурентоспроможності вітчизняних розробників.

- Науковий внесок: Впровадження нових методів та алгоритмів ШІ у розробку ігор має важливе наукове значення, оскільки може бути застосоване і в інших галузях, таких як робототехніка, симуляція та навчання.

1.5. Постановка задачі

Розробка гри у жанрі виживання з використанням ШІ є міждисциплінарним завданням, яке поєднує різні аспекти програмування, графіки, теорії ігор та штучного інтелекту. Ця робота базується на досвіді та досягненнях попередніх проектів у галузі розробки ігор та ШІ, зокрема:

- "The Forest": Ця гра відзначається своєю реалістичною симуляцією поведінки ворогів, які адаптуються до дій гравця. Аналізуючи цей проект, можна виявити ефективні стратегії для розробки адаптивної поведінки ворогів у власній грі.

- "Subnautica": Відомий за складну екосистему, де кожен організм має свою роль та поведінку. Це дає можливість вивчити підходи до моделювання взаємодії між різними елементами ігрового світу.

- "Don't Starve": Використовує процедурну генерацію світу та випадкові події, що забезпечує унікальний досвід у кожній новій грі. Ці методи можуть бути застосовані для створення унікальних ігрових ситуацій у розроблюваному проекті.

Основна задача даної кваліфікаційної роботи полягає у розробці гри у жанрі виживання на платформі Unity з використанням сучасних технологій ШІ. Для досягнення цієї мети необхідно вирішити наступні підзадачі:

1. Аналіз існуючих рішень: Провести детальний аналіз сучасних ігор у жанрі виживання, визначити їхні сильні та слабкі сторони, а також вивчити найкращі практики.

2. Розробка концептуального дизайну гри: Створити детальний план гри, що включатиме основні ігрові механіки, сюжетні лінії та дизайн ігрового світу.

3. Вибір технологій: Обрати оптимальні інструменти та технології для реалізації проекту, зокрема вибір платформи Unity та методів ШІ.

4. Реалізація ігрового світу: Розробити ігровий світ з використанням 3D-графіки та процедурної генерації для забезпечення унікальності кожної ігрової сесії.

5. Інтеграція систем ШІ: Створити та інтегрувати алгоритми ШІ, які забезпечать реалістичну поведінку ворогів та NPC, адаптовану до дій гравця.

6. Тестування та оптимізація: Провести комплексне тестування гри, виявити та усунути недоліки, а також оптимізувати продуктивність для забезпечення стабільної роботи.

Основні технічні характеристики продукту. Розроблювана гра буде мати наступні технічні характеристики, які забезпечують її функціональність, графічну привабливість і складність геймплею:

Платформа. Гра буде розроблена на платформі Unity, що є одним з найпопулярніших та найбільш гнучких інструментів для створення ігор. Unity дозволяє розробляти ігри для різних операційних систем, таких як Windows, macOS, Linux, а також для мобільних платформ iOS та Android. Це забезпечить широкий доступ до гри на різноманітних пристроях, включаючи комп'ютери, планшети та смартфони.

Графіка. Графічне оформлення гри буде виконано в 3D, що забезпечить реалістичність і занурення у віртуальний світ. Використання процедурної генерації дозволить створювати унікальні середовища, які будуть змінюватися під час

кожного сеансу гри. Це забезпечить неповторність кожного ігрового досвіду і підвищить інтерес гравців до дослідження світу гри. Крім того, графічний движок Unity підтримує сучасні технології освітлення, тіней та ефектів частинок, що додатково підвищує візуальну якість гри [5].

Алгоритми штучного інтелекту (ШІ). Для забезпечення адаптивної та реалістичної поведінки персонажів у грі будуть використані передові алгоритми штучного інтелекту:

- Поведінкові дерева (Behavior Trees): Цей метод дозволяє створювати складні моделі поведінки персонажів, що реагують на зміну умов середовища та дії гравця.
- Методи машинного навчання (Machine Learning): Використання алгоритмів машинного навчання дозволить персонажам вчитися на своїх помилках і адаптувати свої дії відповідно до ситуації, роблячи гру більш динамічною і непередбачуваною.
- Теорія ігор (Game Theory): Застосування теорії ігор допоможе моделювати стратегічну поведінку персонажів, що приймають рішення на основі аналізу дій гравця і інших NPC (неігрових персонажів).

Ігрові механіки. Гра буде включати різноманітні ігрові механіки, що забезпечать глибокий та захоплюючий геймплей:

- Система виживання: Гравець повинен підтримувати життєві параметри свого персонажа, такі як здоров'я, голод, спрага та температура тіла.
- Управління ресурсами: Гравець збиратиме ресурси, які можна використовувати для створення предметів, будівництва та підтримки життєдіяльності персонажа.
- Будівництво: Можливість будувати притулки, оборонні споруди та інші об'єкти, що допомагають вижити у ворожому середовищі.

- Створення предметів: Система дозволить гравцям створювати різноманітні предмети, такі як інструменти, зброя, одяг і їжа, використовуючи зібрані ресурси.
- Динамічна погода: Погодні умови у грі будуть змінюватися, впливаючи на ігровий процес і змушуючи гравців адаптувати свої стратегії.
- Випадкові події: У грі будуть генеруватися випадкові події, які додаватимуть непередбачуваності та викликів, роблячи кожен сеанс унікальним.

Завдяки цим технічним характеристикам, гра буде мати високий рівень інтерактивності, графічної якості та складності, що забезпечить захоплюючий та неповторний ігровий досвід для користувачів.

Очікуваний технічно-економічний ефект. Впровадження передових технологій штучного інтелекту у розробку гри дозволить забезпечити не тільки високий рівень реалістичності та інтерактивності, але й значно підвищити конкурентоспроможність продукту на ринку. Очікується, що створення такого інноваційного ігрового продукту матиме наступні технічно-економічні переваги:

- Збільшення комерційного успіху

Висока якість гри та унікальний ігровий досвід сприятимуть залученню більшої кількості гравців та збільшенню продажів. Завдяки використанню передових графічних технологій та інтеграції адаптивного штучного інтелекту, гравці зможуть насолоджуватися непередбачуваним і захоплюючим ігровим процесом, що буде привертати увагу як нових користувачів, так і ветеранів ігрової індустрії.

- Ефективність розробки

Використання Unity як платформи для розробки дозволить значно скоротити час та витрати на створення гри. Unity надає потужні інструменти для розробки, тестування та оптимізації, що забезпечує швидший цикл розробки та високу якість кінцевого продукту. Крім того, використання готових модулів і бібліотек для

процедурної генерації та ШІ знижує необхідність розробки з нуля, що дозволяє зосередитися на унікальних аспектах гри.

- Зниження витрат на підтримку

Інтеграція машинного навчання та поведінкових дерев дозволяє створювати самонавчальні системи, які зменшують потребу в частих оновленнях та коригуваннях. Це забезпечує довготривалу підтримку гри без значних додаткових витрат, а також дозволяє адаптувати ігровий процес до змін у вподобаннях гравців та ринкових тенденціях.

- Підвищення рівня задоволеності користувачів

Адаптивна поведінка персонажів та динамічні ігрові механіки забезпечать гравцям більш насичений та інтерактивний досвід, що сприятиме високому рівню задоволеності та лояльності користувачів. Це, в свою чергу, може призвести до позитивних відгуків та рекомендацій, що є важливими факторами для залучення нових гравців та зростання спільноти.

- Розширення можливостей монетизації

Завдяки інтеграції сучасних технологій і механік, гра може пропонувати різноманітні моделі монетизації, такі як продажі DLC (додаткового контенту), підписки, внутрішньоігрові покупки та рекламні інтеграції. Це дозволить забезпечити стабільний потік доходів після релізу гри та збільшити загальний економічний ефект від її реалізації.

- Інноваційність як конкурентна перевага

Інноваційні рішення, такі як використання штучного інтелекту і процедурної генерації, стануть важливим аспектом маркетингової стратегії гри. Висвітлення цих технологічних досягнень у рекламних кампаніях дозволить виділитися серед конкурентів та залучити аудиторію, що шукає нові та захоплюючі ігрові враження.

Загалом, впровадження передових технологій у розробку гри дозволить створити високоякісний продукт з потужними конкурентними перевагами, що сприятиме як технічному, так і економічному успіху на ринку.

1.6. Висновки до першого розділу

Розділ був присвячений аналізу предметної області з обґрунтуванням актуальності, необхідності та причинної зумовленості виконання кваліфікаційної роботи, а також постановці задачі. У цьому розділі ми розглянули фундаментальні аспекти, які формують основу нашого проекту, зокрема теоретичні основи ігор, історичний контекст розвитку ігор, сучасний стан ігрової індустрії та основні технічні характеристики розроблюваного продукту.

У розділі було висвітлено такі ключові аспекти:

1. Поняття гри: Було визначено основні характеристики ігор як виду діяльності, що відрізняється непродуктивністю у матеріальному сенсі, але має велике значення для соціального та культурного розвитку. Розглянуто різні визначення гри, зокрема відомого філософа Йогана Гейзінга, який описав гру як добровільну діяльність, що відбувається в умовах, обмежених часом і простором, за певними правилами.

2. Гра як історичний аспект: Проаналізовано історичний розвиток гри від ритуальних практик первісних людей до складних соціальних та культурних структур. Розглянуто роль гри у формуванні культури та цивілізації, її вплив на розвиток мови та суспільних інститутів. Гра сприяла розвитку людської фантазії та абстрактного мислення, що стало важливим етапом еволюції людини.

3. Сучасний стан ігрової індустрії: Проведено аналіз сучасної ігрової індустрії, яка є однією з найдинамічніших та найприбутковіших галузей. Розглянуто популярні жанри ігор, тенденції розвитку та технологічні інновації, що

впливають на створення нових ігрових продуктів. Особливу увагу приділено жанру виживання, який набуває все більшої популярності серед гравців завдяки своїй динаміці та глибокому ігровому процесу.

4. Основні технічні характеристики продукту: Описано технічні характеристики розроблюваної гри, включаючи вибір платформи Unity, використання 3D графіки з процедурною генерацією середовищ, застосування алгоритмів штучного інтелекту для забезпечення адаптивної поведінки персонажів, а також впровадження складних ігрових механік, таких як система виживання, управління ресурсами, будівництво, створення предметів, динамічна погода та випадкові події.

На основі проведеного аналізу можна зробити наступні висновки:

- Актуальність дослідження: Розробка гри в жанрі виживання є актуальною та затребуваною задачею, що відповідає сучасним тенденціям ігрової індустрії та потребам гравців. Висока реіграбельність та захоплюючий ігровий процес роблять такі ігри популярними та комерційно успішними.
- Наукова та практична значущість: Аналіз історичного та сучасного контексту розвитку ігор дозволяє глибше зрозуміти роль і значення ігрової діяльності у формуванні культури та суспільства. Це знання є важливим для розробки інноваційних ігрових продуктів, які не тільки розважають, але й сприяють соціальному та культурному розвитку.
- Вибір технологій: Вибір платформи Unity та використання передових методів процедурної генерації та штучного інтелекту забезпечить створення високоякісного ігрового продукту, який відповідатиме сучасним вимогам та очікуванням гравців.

РОЗДІЛ 2
ОБГРУНТУВАННЯ ВИБОРУ МЕТОДІВ СТВОРЕННЯ ГРИ

2.1. Огляд існуючих аналогів ігор на виживання

У сфері розробки ігор у жанрі виживання існує безліч різноманітних рішень, які застосовують різні технології та методи для забезпечення інтерактивного та захоплюючого ігрового досвіду. Розглянемо деякі з найвідоміших ігор цього жанру, а також технології, які вони використовують.

Minecraft. Є одним із найуспішніших проектів у жанрі виживання. Гра базується на процедурній генерації світу, що складається з блоків, які гравці можуть руйнувати і будувати заново. Основні характеристики Minecraft включають:

- Графіка: Піксельна графіка з 3D-блоками.
- Процедурна генерація: Створення унікальних світів кожного разу при новій грі.
- Ігрові механіки: Збір ресурсів, будівництво, крафтинг, виживання в умовах постійних загроз від монстрів.
- Мультиплеєр: Підтримка багатокористувацького режиму, що дозволяє гравцям взаємодіяти один з одним у загальному світі.

Minecraft підтримує модифікації, що дозволяють гравцям додавати нові функції та змінювати механіки гри за допомогою спільноти модерів. Також активно підтримується розробниками, які регулярно випускають оновлення з новими функціями та контентом, що забезпечує тривалу зацікавленість гравців.

Гра стала платформою для творчості та інновацій, надихаючи численні проекти та спільноти по всьому світу (рис. 2.1).

Рисунок 2.1 – Приклад ігрового процесу у Minecraft

Rust. Розроблений Facepunch Studios, представляє собою гру з високою якістю графіки та глибокими ігровими механіками. Основні характеристики Rust включають:

- Графіка: Сучасна 3D-графіка з реалістичними текстурами та освітленням.
- Процедурна генерація: Створення великих відкритих світів з різноманітними ландшафтами.
- Ігрові механіки: Виживання, будівництво баз, крафтинг, управління ресурсами, PvP-взаємодія.
- Штучний інтелект: Адаптивна поведінка NPC та диких тварин, що реагують на дії гравця.

Rust відомий своїм жорстким багатокористувацьким середовищем, де гравці можуть взаємодіяти як кооперативно, так і ворожо, створюючи власні альянси та конфлікти (рис. 2.2).

Рисунок 2.2 – Приклад ігрового процесу у Rust

The Forest. Гра, яка поєднує елементи виживання та хорору. Основні характеристики The Forest включають:

- Графіка: Висока якість графіки з детальними середовищами та реалістичними освітленням і тінями.
- Процедурна генерація: Динамічне створення ландшафтів та навколишнього середовища.
- Ігрові механіки: Виживання, будівництво укриттів, крафтинг, дослідження та взаємодія з ворогуючими NPC.
- Штучний інтелект: Вороги (канібали) мають складну поведінку, включаючи патрулювання, атаки та взаємодію один з одним.

The Forest відрізняється унікальною атмосферою жаху та постійної загрози, що додає гравцям додаткового виклику. Гравці можуть співпрацювати в кооперативному режимі, що підсилює відчуття взаємодопомоги та спільного подолання труднощів (рис. 2.3).

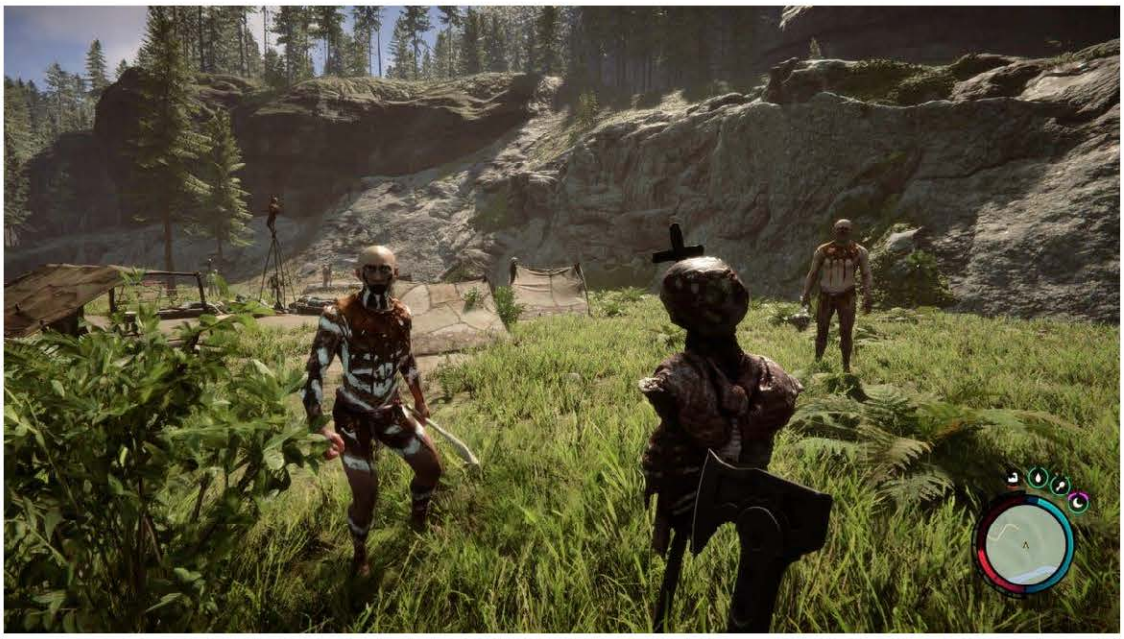

Рисунок 2.3 – Приклад ігрового процесу у The Forest

ARK: Survival Evolved. Поєднує виживання з науковою фантастикою. Основні характеристики ARK включають:

- Графіка: Висока якість 3D-графіки з детальними моделями динозаврів та середовищем.
- Процедурна генерація: Великі, відкриті світи з різноманітними біомами та екосистемами.
- Ігрові механіки: Виживання, приручення та виховання динозаврів, будівництво, крафтинг, управління ресурсами.
- Штучний інтелект: Поведінка динозаврів та інших істот, що залежить від їхнього типу та навколишніх умов.

ARK відомий своєю глибиною геймплею, різноманіттям динозаврів та можливістю гравців створювати власні бази та спільноти. Гравці можуть також об'єднуватися у племена, що дозволяє співпрацювати та розширювати вплив у світі гри (рис. 2.4).

Рисунок 2.4 – Приклад ігрового процесу у ARK: Survival Evolved

2.2. Аналіз існуючих рішень

Аналіз існуючих рішень у жанрі виживання показує, що ключовими аспектами успішних ігор є процедурна генерація, висока якість графіки, розширені ігрові механіки та адаптивний штучний інтелект [12]. Розглянемо детально кожен з цих аспектів.

Процедурна генерація, є важливою технологією в сучасних іграх, що дозволяє створювати великі, детальні та унікальні світи. Ця технологія забезпечує:

- Унікальність кожного проходження: Завдяки процедурній генерації, кожен раз, коли гравець починає нову гру, створюється унікальний світ. Це підвищує реіграбельність гри, оскільки гравці можуть щоразу досліджувати нові місця та зустрічати нові виклики.
- Зниження обсягу даних: Процедурно згенеровані світи не потребують зберігання великих обсягів даних. Це зменшує розмір гри та дозволяє швидше завантажувати нові рівні або зони.

- Різноманітність середовищ: Процедурна генерація дозволяє створювати різноманітні ландшафти, екосистеми та погодні умови, що робить гру більш динамічною та цікавою.

Прикладом успішного використання процедурної генерації є гра Minecraft, де гравці можуть досліджувати нескінченний світ, що постійно змінюється.

Висока якість графіки є критичним аспектом для залучення та утримання гравців. Основні елементи, що забезпечують високу якість графіки, включають:

- Деталізація текстур: Використання високоякісних текстур робить об'єкти та середовища більш реалістичними. Це особливо важливо для ігор у жанрі виживання, де гравці очікують відчувати себе частиною живого світу.
- Реалістичне освітлення та тіні: Динамічне освітлення та тіні додають глибині та реалістичності ігровим середовищам. Наприклад, у грі The Forest освітлення змінюється залежно від часу доби та погодних умов, що створює атмосферу занурення.
- Анімації персонажів та NPC: Реалістичні анімації роблять рухи персонажів природними та правдоподібними. Це підвищує рівень занурення та робить взаємодію з NPC більш інтерактивною.

Ігри, як Rust та ARK: Survival Evolved, використовують сучасні графічні технології для створення вражаючих візуальних ефектів та реалістичних середовищ.

Розширені ігрові механіки. Ігрові механіки є серцем будь-якої гри, і в жанрі виживання вони повинні бути глибокими та різноманітними, щоб забезпечити тривалий інтерес гравців. Основні компоненти розширених ігрових механік включають:

- Механіки виживання: Гравці повинні збирати ресурси, такі як їжа, вода та матеріали для будівництва, щоб вижити в агресивному середовищі. Наприклад,

у грі The Long Dark гравці повинні слідкувати за своїм здоров'ям, температурою тіла та рівнем голоду.

- Будівництво та крафтинг: Можливість будувати укриття та створювати предмети є важливою частиною гри. Це дозволяє гравцям адаптуватися до навколишнього середовища та захищати себе від небезпек.
- Управління ресурсами: Гравці повинні ефективно управляти своїми ресурсами, щоб вижити та розвиватися. Це включає збір, зберігання та використання ресурсів для крафтингу та будівництва.
- Динамічна погода та випадкові події: Зміни погодних умов та випадкові події додають грі непередбачуваності та викликів. Наприклад, у грі Subnautica погодні умови впливають на видимість та поведінку підводних істот.

Ці механіки створюють глибокий та захоплюючий ігровий процес, який утримує гравців на тривалий час.

Адаптивний штучний інтелект (ШІ) забезпечує створення динамічних і непередбачуваних взаємодій у грі. Основні аспекти адаптивного ШІ включають:

- Поведінкові дерева: Використання поведінкових дерев дозволяє моделювати складні моделі поведінки NPC. Це включає патрулювання, реагування на дії гравця та взаємодію з іншими NPC [8].
- Машинне навчання: Методи машинного навчання можуть бути використані для створення більш адаптивних та розумних NPC. Наприклад, NPC можуть вчитися на основі дій гравця та адаптувати свою поведінку, щоб створювати нові виклики.
- Теорія ігор: Використання теорії ігор дозволяє моделювати стратегічні взаємодії між гравцями та NPC. Це може включати співпрацю, конкуренцію та конфлікти [14].

Ігри, як Rust та The Forest, використовують складні алгоритми ШІ для створення реалістичних та викликаючих взаємодій між гравцями та NPC.

2.3. Вибір методів вирішення

Після аналізу існуючих рішень та визначення ключових аспектів успішних ігор у жанрі виживання, необхідно обрати методи та технології, які найкраще відповідають вимогам нашого проєкту. У цьому розділі розглянемо вибір платформи, методи процедурної генерації, алгоритми штучного інтелекту та розробку ігрових механік.

Порівняння різних платформ та вибір однієї для подальшої розробки. Вибір платформи для розробки гри залежить від багатьох факторів, таких як потреби проекту, навички розробника, фінансові можливості та технічні вимоги [4]. Порівняємо деякі варіанти між собою:

1. Unity

Особливості:

- Підтримка 2D та 3D: Unity підтримує як 2D, так і 3D розробку, що робить його універсальним інструментом для різних жанрів ігор [15].
- Мультиплатформеність: Unity дозволяє експортувати проекти на численні платформи, включаючи Windows, macOS, Linux, Android, iOS, ігрові консолі (PlayStation, Xbox, Nintendo Switch) і навіть WebGL [15].
- Великий вибір інструментів та плагінів: Unity Asset Store пропонує безліч ресурсів, від моделей і анімацій до скриптів і повних шаблонів ігор [15].
- Простота використання: Завдяки інтуїтивно зрозумілому інтерфейсу і великій кількості навчальних матеріалів, Unity підходить як для новачків, так і для професіоналів.

Переваги:

- Велика спільнота розробників та потужна підтримка.
- Регулярні оновлення та нові функції.

- Підтримка VR/AR.

Недоліки:

- Може бути вимогливим до ресурсів комп'ютера.
- Безкоштовна версія має деякі обмеження.

2. Unreal Engine

Особливості:

- Висока якість графіки: Unreal Engine відомий своїми потужними можливостями для створення реалістичної графіки, завдяки чому він широко використовується в AAA-іграх [2].
- Blueprints: Візуальне програмування через Blueprints дозволяє розробникам створювати ігрову логіку без написання коду [2].
- Підтримка C++: Для більш складних проектів Unreal Engine дозволяє використовувати мову програмування C++.

Переваги:

- Потужний графічний рушій для створення реалістичних ігор.
- Інструменти для створення великих відкритих світів.
- Безкоштовний доступ з відсотком від продажів після досягнення певного доходу.

Недоліки:

- Високий поріг входу для новачків.
- Вимогливість до апаратного забезпечення.

3. Godot

Особливості:

- Відкритий вихідний код: Godot є проектом з відкритим вихідним кодом, що дозволяє розробникам повністю контролювати свої проекти [4].

- Скриптинг на GDScript: Godot використовує власну мову скриптів GDScript, яка схожа на Python [4].
- Підтримка 2D та 3D: Хоча Godot добре підходить для 3D, він особливо популярний для розробки 2D ігор.

Переваги:

- Повністю безкоштовний без роялті.
- Легкість у навчанні та використанні.
- Регулярні оновлення та активна спільнота.

Недоліки:

- Менш потужний графічний рушій у порівнянні з Unity та Unreal.
- Обмежені ресурси та плагіни в порівнянні з іншими платформами.

4. CryEngine

Особливості:

- Потужний графічний рушій: CryEngine відомий своїми високими графічними можливостями, що робить його популярним для створення реалістичних ігор.
- Вбудовані інструменти для дизайну рівнів: CryEngine пропонує потужні інструменти для створення та редагування рівнів.

Переваги:

- Висока якість графіки.
- Безкоштовний доступ з роялті від доходу.

Недоліки:

- Складний у використанні для новачків.
- Менша спільнота у порівнянні з Unity та Unreal.

Порівнявши всі представлені платформи для розробки ігор, вибір Unity видається наибільш обгрунтованим та вигідним. Unity є одним із найпопулярніших

і потужних ігрових рушіїв, який пропонує широкий спектр можливостей для створення ігор різних жанрів. Основні причини вибору Unity включають:

- Мультиплатформеність: Unity підтримує розробку для різних платформ, включаючи ПК, консолі, мобільні пристрої та віртуальну реальність. Це дозволить нашій грі досягти широкої аудиторії.
- Гнучкість та розширюваність: Unity має велику кількість готових компонентів і плагінів, що спрощують розробку. Крім того, можливість написання власних скриптів на C# дозволяє реалізувати унікальні ігрові механіки та функції.
- Сильна підтримка спільноти: Велика спільнота розробників Unity забезпечує доступ до численних ресурсів, уроків та форумів, що полегшує вирішення технічних проблем та обмін досвідом.

Методи процедурної генерації. Процедурна генерація є важливим аспектом нашої гри, оскільки вона забезпечить унікальність кожного проходження та збільшить реіграбельність. Ми використовуватимемо наступні методи процедурної генерації:

Шум Перліна: Шум Перліна є класичним методом процедурної генерації ландшафтів, який дозволяє створювати природні та органічні форми. Використання цього методу дозволить нам генерувати різноманітні ландшафти з плавними переходами між різними типами місцевості [13].

Вороні діаграми: Цей метод дозволяє створювати розподіл територій з різними біомами та кліматичними зонами. Використання Вороні діаграм допоможе забезпечити різноманітність середовищ та умов виживання.

Фрактальні алгоритми: Фрактальні алгоритми можуть бути використані для створення складних і детальних структур, таких як гори, печери та інші геологічні утворення. Це додасть грі глибини та різноманітності.

Алгоритми штучного інтелекту. Адаптивний штучний інтелект є ключовим елементом нашої гри, що забезпечує динамічні та непередбачувані взаємодії. Для реалізації ШІ ми обрали наступні методи:

Поведінкові дерева (Behavior Trees): Поведінкові дерева є потужним інструментом для моделювання складних моделей поведінки NPC. Вони дозволяють розробляти ієрархічні стани та умови, що забезпечують гнучку та адаптивну поведінку персонажів [1].

Методи машинного навчання: Використання алгоритмів машинного навчання, таких як нейронні мережі та Q-навчання, дозволить NPC навчатися на основі дій гравця та адаптувати свою поведінку для створення нових викликів.

Теорія ігор: Використання теорії ігор для моделювання стратегічних взаємодій між гравцями та NPC. Це дозволить реалізувати складні сценарії співпраці, конкуренції та конфліктів [14].

Розробка ігрових механік. Ігрові механіки є серцем будь-якої гри. Для нашої гри ми обрали наступні механіки, що забезпечать глибокий та захоплюючий ігровий процес:

Система виживання: Гравці повинні збирати ресурси, такі як їжа, вода та матеріали для будівництва, щоб вижити в агресивному середовищі. Ця система включає моніторинг стану здоров'я персонажа, рівня голоду, спраги та енергії.

Управління ресурсами: Ефективне управління ресурсами є важливим для виживання. Гравці повинні вирішувати, як розподіляти свої ресурси між різними потребами, такими як будівництво, крафтинг та підтримка життєдіяльності.

Будівництво та крафтинг: Гравці зможуть будувати укриття, створювати інструменти та зброю. Це дозволить їм адаптуватися до навколишнього середовища та захищати себе від небезпек.

Динамічна погода та випадкові події: Зміни погодних умов та випадкові події додають грі непередбачуваності та викликів. Гравці повинні будуть адаптувати свою стратегію в залежності від змін навколишнього середовища.

Соціальна взаємодія: Гравці можуть взаємодіяти з іншими персонажами та NPC, формуючи альянси або вступаючи у конфлікти. Це додасть грі соціальної динаміки та глибини.

2.4. Висновки до другого розділу

Розділ був присвячений аналізу існуючих рішень для задачі розробки гри в жанрі виживання та вибору найефективніших методів для реалізації проекту. Проведений аналіз дозволив визначити сильні та слабкі сторони сучасних ігор у цьому жанрі, а також виявити найуспішніші підходи та технології, що застосовуються у провідних ігрових продуктах.

У розділі було розглянуто такі ключові аспекти:

Аналіз існуючих рішень: Було проаналізовано популярні ігри жанру виживання, такі як "Minecraft", "ARK: Survival Evolved", "Rust", та "Don't Starve". Особливу увагу приділено ігровим механікам, системам виживання, управлінню ресурсами, крафтингу та взаємодії з навколишнім середовищем. Аналіз показав, що успішні ігри цього жанру мають високу реіграбельність, динамічний ігровий процес та добре реалізовані механіки виживання.

Вибір методів вирішення: На основі проведеного аналізу було обрано методи та технології, які найкраще підходять для нашого проекту:

Платформа Unity: Вибір Unity обумовлений її мультиплатформеністю, гнучкістю та потужністю. Unity дозволяє створювати високоякісні 3D-ігри та має велику спільноту підтримки.

Процедурна генерація: Використання методів процедурної генерації, таких як шум Перліна, Вороні діаграми та фрактальні алгоритми, дозволить створювати унікальні та різноманітні ігрові середовища.

Штучний інтелект: Застосування поведінкових дерев, методів машинного навчання та теорії ігор забезпечить адаптивну та непередбачувану поведінку NPC, підвищуючи рівень інтерактивності та реалістичності гри.

Ігрові механіки: Розробка системи виживання, управління ресурсами, будівництва та крафтингу, динамічної погоди та випадкових подій створить глибокий та захоплюючий ігровий процес.

На основі отриманих результатів можна зробити наступні висновки:

Комплексний підхід: Вибір Unity як основної платформи, використання процедурної генерації та передових методів штучного інтелекту дозволить створити інноваційну гру з високим рівнем реалістичності та інтерактивності.

Конкурентоспроможність: Використання найкращих практик та технологій забезпечить конкурентоспроможність нашого продукту на ринку ігор у жанрі виживання.

Реіграбельність та залученість: Процедурна генерація та адаптивний штучний інтелект підвищать реіграбельність гри, забезпечуючи унікальний ігровий досвід для кожного користувача.

Таким чином, проведений аналіз та обрані методи дозволять створити високоякісний ігровий продукт, який відповідатиме сучасним вимогам та потребам гравців у жанрі виживання.

РОЗДІЛ 3
РОЗРОБКА ГРИ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

3.1. Розробка та реалізація сцени та її дизайн

Для створення ігрового поля у грі використовувався інструмент Terrain, наявний у Unity. Цей інструмент дозволяє створювати великий, детально опрацьований ландшафт із різними природними елементами, такими як пагорби, долини, ліси та водоймища [15] (рис. 3.1).

Рисунок 3.1 – Створення ігрового поля

Написано скрипт для поведінки предметів у воді. Складається він з двох функцій: OnTriggerStay та OnTriggerExit.

OnTriggerStay. Викликається, коли об'єкт перебуває у воді. Перевіряється, чи інший об'єкт є водою (вода має значення шару під номером 4). Розраховується, як високо об'єкт повинен піднятися, і додається сила вгору, щоб він плавав. У кінці додається опір до руху об'єкта, щоб він поводився, як у воді (рис. 3.2).

```
private void OnTriggerStay(Collider other)
{
    if(other.gameObject.layer == 4)
    {
        float difference = (other.transform.position.y - transform.position.y) * floatUpSpeed;
        rigidbody.AddForce(new Vector3(0f, Mathf.Clamp((Mathf.Abs(Physics.gravity.y) * difference),0,
            Mathf.Abs(Physics.gravity.y) * floatUpSpeedLimit),0f), ForceMode.Acceleration);
        rigidbody.drag = 0.99f;
        rigidbody.angularDrag = 0.8f;
    }
}
```

Рисунок 3.2 – Метод поводження об'єкта на воді

OnTriggerExit. викликається автоматично, коли інший об'єкт, який був у зоні тригера, виходить з цієї зони. У нашому випадку, коли об'єкт виходить з води, цей метод забезпечує відповідну реакцію для об'єкта, щоб він припинив поводитися, як у воді, і почав поводитися, як на суші. Знову перевіряється, чи інший об'єкт належить до шару 4. Якщо умова є істинною, тобто якщо об'єкт виходить з води, ми змінюємо параметри фізики нашого об'єкта. Скидаються лінійний та кутовий опори об'єкта до нуля (рис. 3.3).

```
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.layer == 4)
    {
        rigidbody.drag = 0f;
        rigidbody.angularDrag = 0f;
    }
}
```

Рисунок 3.3 – Метод виходу об'єкта з води

Було додане поселення, завдяки якому гравець зможе пережити перші дні виживання та взаємодії з мешканцями у подальших оновленнях гри (рис. 3.4).

Рисунок 3.4 – Створення поселення місцевих мешканців

Для більш динамічної гри, були додані зміна дня та ночі та дощ. Це дає можливість користувачу опинитися у різних ситуаціях, при яких треба діяти по різному (рис. 3.5).

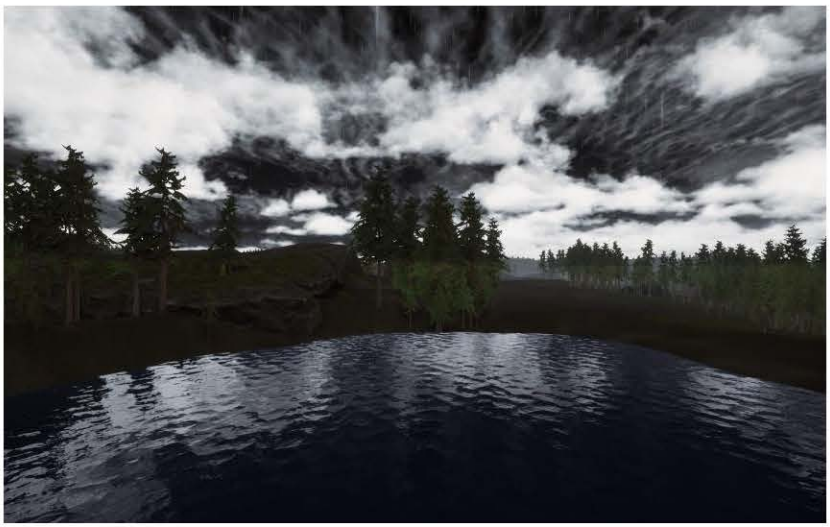

Рисунок 3.5 – Середовище у нічний час під час дощу

3.2. Розробка та реалізація інтерфейсу

Інтерфейс. Розробка інтерфейсу - один із найважливіших етапів створення гри. Інтуїтивно зрозумілий інтерфейс дає змогу гравцеві глибше зануритися у геймплей. Від якості інтерфейсу залежить кількість задоволення, яке отримає користувач під час ігрового процесу.

Реалізована можливість бачити на екрані - наскільки гравець ситий, кількість води в організмі та який відсоток здоров'я в персонажа (рис. 3.6).

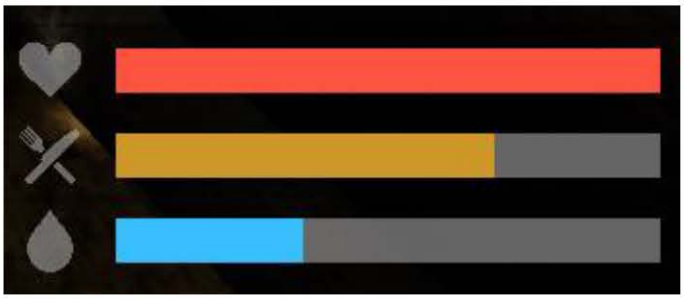

Рисунок 3.6 – Індикатори здоров’я, голоду та спраги

Методи роботи індикаторів:

- Метод оновлення водного індикатора. Перевіряється, чи персонаж у воді і чи натиснута клавіша "E". Якщо так, викликається метод для збільшення кількості води.
- Оновлення кількості їжі. Якщо їжа є, вона поступово зменшується, а інтерфейс оновлюється. Якщо їжа закінчується, інтерфейс встановлюється на 0.
- Оновлення кількості води: Аналогічно до їжі, вода поступово зменшується, а інтерфейс оновлюється. Якщо вода закінчується, інтерфейс встановлюється на 0.
- Зниження здоров'я: Якщо їжі або води недостатньо, здоров'я зменшується. Показник здоров'я поступово оновлюється в інтерфейсі.

Також було реалізовано відображення інвентарю, який має при собі гравець. У нашому випадку гравець може підбирати різноманітні матеріали, інструменти та інше. На малюнку 3.7 можна побачити, який вигляд має інвентар, коли в ньому зібрані предмети.

Рисунок 3.7 – Інвентар гравця

Метод видалення предмету з комірки інвентаря. Метод отримує доступ до слота інвентаря, використовуючи індекс. Умова перевіряє, чи тип одягу в слоті не є None (тобто слот містить одяг) і чи слот не є порожнім. Якщо умова істинна, метод проходить через кожен об'єкт ClothAdder у списку _clothAdders і викликає метод RemoveClothes для видалення одягу, представлений clothingPrefab з елемента слота.

Після слот очищується: видаляється посилання на предмет, встановлюється порожнім, кількість предметів у слоті встановлюється на 0, а іконка і кількість предметів видаляються з інтерфейсу користувача (рис. 3.8).

```
public void RemoveItemFromSlot(int slotId)
{
    InventorySlot slot = slots[slotId];

    if (slot.clothType != ClothType.None && !slot.isEmpty)
    {
        foreach (ClothAdder clothAdder in _clothAdders)
        {
            clothAdder.RemoveClothes(slot.item.clothingPrefab);
        }
    }
    slot.item = null;
    slot.isEmpty = true;
    slot.amount = 0;
    slot.iconGO.GetComponent<Image>().color = new Color(1, 1, 1, 0);
    slot.iconGO.GetComponent<Image>().sprite = null;
    slot.itemAmountText.text = "";
}
```

Рисунок 3.8 – Метод видалення предмету з комірки інвентаря

Метод AddItemToSlot додає предмет до певного слота інвентаря, при цьому:

- Призначає предмету слот і оновлює його іконку.
- Встановлює правильну кількість предметів у слоті, враховуючи максимальну допустиму кількість для цього типу предметів.
- Якщо предмет є одягом, він додається до відповідних об'єктів ClothAdder (рис. 3.9).

```
    if (_amount <= _item.maximumAmount)
    {
        slot.amount = _amount;
        if (slot.item.maximumAmount != 1) // added this if statement for single items
        {
            slot.itemAmountText.text = slot.amount.ToString();
        }
    }
    else
    {
        slot.amount = _item.maximumAmount;
        _amount -= _item.maximumAmount;
        if (slot.item.maximumAmount != 1) // added this if statement for single items
        {
            slot.itemAmountText.text = slot.amount.ToString();
        }
    }

    if (slot.clothType != ClothType.None)
    {
        foreach (ClothAdder clothAdder in _clothAdders)
        {
            clothAdder.addClothes(slot.item.clothingPrefab);
        }
    }
}
```

Рисунок 3.9 – Метод додавання предмету до інвентаря

Для більш комфортної гри були створені «швидкі» слоти. П'ять слотів із яких можна швидко взяти предмет до рук та використати його. Між ними можна переключатися за кнопками від 1 до 5, або за допомогою колеса миші (рис. 3.10).

Рисунок 3.10 – Слоти для швидкої зміни інструментів

Методи скрипту «швидких» слотів:

SelectSlot. Вибирає слот швидкого доступу і показує предмет, який знаходиться в ньому. Змінює зображення слота на зображення предмету. Зберігає обраний слот як activeSlot. Викликає методи для відображення предмета в руках і будівельного блоку, якщо він є (рис. 3.11).

```
private void SelectSlot()
{
    quickslotParent.GetChild(currentQuickslotID).GetComponent<Image>().sprite = selectedSprite;
    activeSlot = quickslotParent.GetChild(currentQuickslotID).GetComponent<InventorySlot>();
    ShowItemInHand();
    ShowBuildingBlock();
}
```

Рисунок 3.11 – Метод SelectSlot

ShowBuildingBlock. Включає або вимикає систему будівництва в залежності від типу предмета в активному слоті.

Якщо активний слот або предмет в ньому відсутні, система будівництва вимикається.

Якщо тип предмета BuildingBlock, система будівництва включається, і поточний будівельний блок встановлюється відповідно до активного предмета (рис. 3.12).

```
public void ShowBuildingBlock()
{
    if (activeSlot == null)
    {
        buildingSystem.enabled = false;
        return;
    }

    if(activeSlot.item == null)
    {
        buildingSystem.enabled = false;
        return;
    }

    if (activeSlot.item.itemType == ItemType.BuildingBlock)
    {
        buildingSystem.enabled = true;
        buildingSystem.currentBuildingBlock = activeSlot.item.itemPrefab;
        buildingSystem.ChangeBuildingBlock();
    }
    else
    {
        buildingSystem.enabled = false;
    }
}
```

Рисунок 3.12– Метод ShowBuildingBlock

RemoveConsumableItem. Видаляє або зменшує кількість споживаного предмета в слоті швидкого доступу.

Якщо кількість предмета в слоті менша або дорівнює 1, слот очищується. Інакше кількість предмета зменшується на 1 і відповідний текст оновлюється (рис. 3.13).

```
private void RemoveConsumableItem()
{
    if (quickslotParent.GetChild(currentQuickslotID).GetComponent<InventorySlot>().amount <= 1)
    {
        quickslotParent.GetChild(currentQuickslotID).GetComponentInChildren<DragAndDropItem>()
            .NullifySlotData();
    }
    else
    {
        quickslotParent.GetChild(currentQuickslotID).GetComponent<InventorySlot>().amount--;
        quickslotParent.GetChild(currentQuickslotID).GetComponent<InventorySlot>().itemAmountText
            .text = quickslotParent.GetChild(currentQuickslotID).GetComponent<InventorySlot>()
            .amount.ToString();
    }
}
```

Рисунок 3.13– Метод RemoveConsumableItem

CheckItemInHand. Перевіряє наявність активного слота і відображає предмет в руках або ховає його.

Якщо активний слот існує, викликається метод ShowItemInHand. Інакше, викликається метод HideItemsInHand (рис. 3.14).

```
public void CheckItemInHand()
{
    if (activeSlot != null)
    {
        ShowItemInHand();
    }
    else
    {
        HideItemsInHand();
    }
}
```

Рисунок 3.14– Метод CheckItemInHand

ShowItemInHand. Показує предмет в руках персонажа, якщо він є.

Ховає всі предмети в руках. Якщо предмет у активному слоті відсутній, метод завершується. Інакше, активується відповідний предмет з усіх можливих (рис. 3.15).

```
private void ShowItemInHand()
{
    HideItemsInHand();
    if (activeSlot.item == null)
    {
        return;
    }
    for (int i = 0; i < allWeapons.childCount; i++)
    {
        if (activeSlot.item.inHandName == allWeapons.GetChild(i).name)
        {
            allWeapons.GetChild(i).gameObject.SetActive(true);

        }
    }
}
```

Рисунок 3.15– Метод ShowItemInHand

HideItemsInHand. Ховає всі предмети, які можуть бути в руках персонажа (рис. 3.16).

```
private void HideItemsInHand()
{
    for (int i = 0; i < allWeapons.childCount; i++)
    {
        allWeapons.GetChild(i).gameObject.SetActive(false);
    }
}
```

Рисунок 3.16– Метод HideItemsInHand

У вікні інвентаря буди додані окремі комірки для одягу персонажу. Всього виділено п’ять комірок під одяг (голова, корпус, ноги та окремо взуття). Також буда додана модель персонажа, щоб можна було одразу побачити, як виглядає одяг на гравці (рис. 3.17).

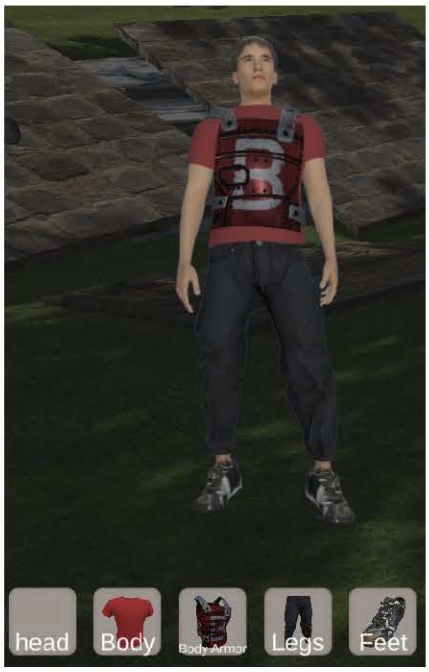

Рисунок 3.17 – Комірки під одяг та модель гравця

3.3. Створення неігрових персонажів та налаштування штучного інтелекту

Для більш цікавого та різноманітного проведення часу у грі, створено неігрових персонажів. На рисунку приведено приклад двох тварин, кабана та вовка (рис. 3.18).

Рисунок 3.18 – Неігрові персонажі вовк та кабан

За допомогою вбудованого інструменту Animator додаємо анімації персонажів при різних ситуаціях, які відбуваються навколо них (рис. 3.19).

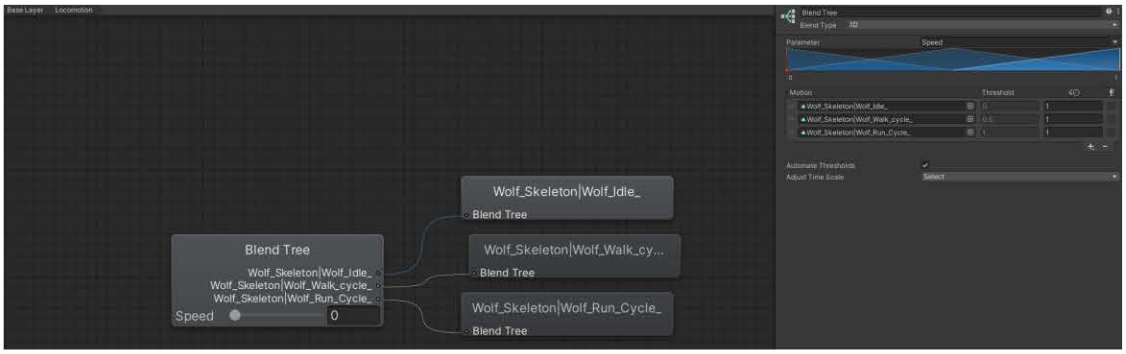

Рисунок 3.19 – Створення дерева анімацій на прикладі вовка

Наступним етапом проводимо тренування тварин. На прикладі того самого вовка покажемо етапи тренування неігрових персонажів.

1. Створюється зона, де персонаж може пересуватися (рис. 3.20). Місця де стоять об'єкти бажано відокремити, щоб потім тварини не застрягали.

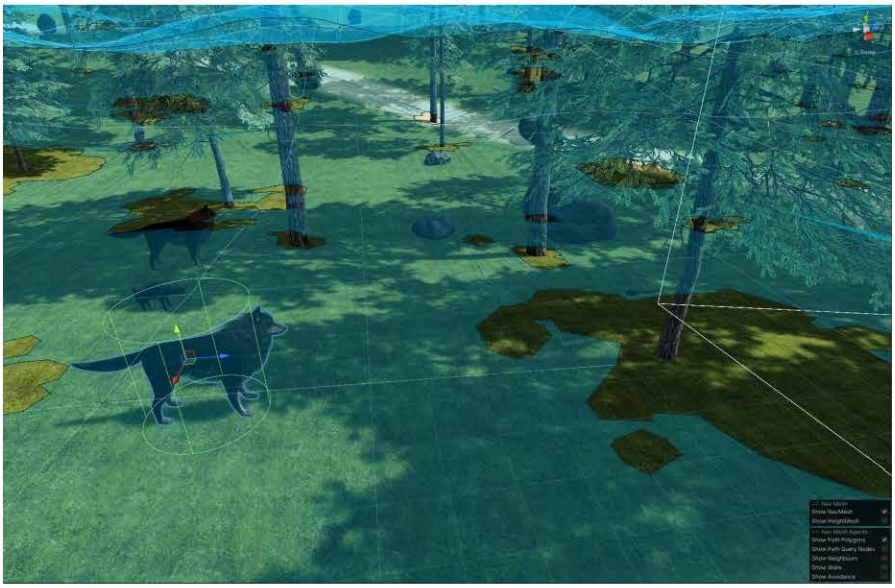

Рисунок 3.20 – Зона пересування неігрового персонажу

2. Прописуємо скрипт пересування тварин. Використовуючи вбудовану бібліотеку UnityEngine.AI, прописуємо необхідні правила для поведінки неігрового персонажу (додаток А). Суть полягає в тому, що в певному радіусі, який задає розробник, обирається випадково точка. Якщо ця точка знаходиться у межах доступної зони, персонаж починає пересуватися до неї. У іншому випадку обирається інша точка (рис. 3.21).

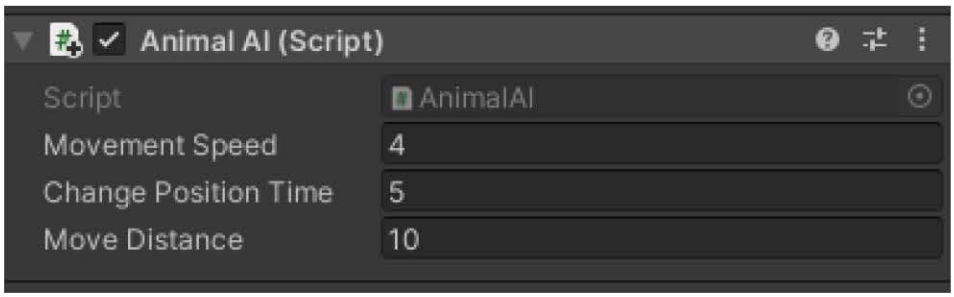

Рисунок 3.21 – Властивості переміщення. Змінні швидкість, час та дистанція переміщення

3.4. Основні механіки гри

Переміщення персонажу. Переміщення гравця відбувається за допомогою клавіш WASD, де W – вперед, A – вліво, S – назад та D – вправо. За допомогою клавіши Space, персонаж виконує стрибок, а натиснувши Shift – зробить прискорення.

Створення інструментів. Для створення інструментів та інших предметів, розробимо крафтове вікно. Вікно умовно поділене на дві частини. З лівої сторони меню доступних речей для крафту. З іншої сторони вказується кількість предметів котрі гравець хоче зробити, час за який це відбудеться та кількість необхідних матеріалів. Регулювати кількість інструментів можна за допомогою двох кнопок

«+» (додати) та «-» (відняти). Створення предметів відбувається після взаємодії з кнопкою «Craft» (рис. 3.22).

Рисунок 3.22 – Вікно створення предметів

Розглянемо, як працює метод додавання предметів до черги крафту в грі. Метод виконує кілька кроків:

Підготовка до крафту: Перевіряється, які ресурси потрібні для створення певного предмета. Розглядається кожен ресурс, який потрібно використовувати.

Перевірка наявності ресурсів: Для кожного ресурсу визначається, скільки саме ресурсу необхідно взяти з інвентаря гравця.

Взаємодія з інвентарем: Потім метод перевіряє інвентар гравця, щоб знайти необхідні ресурси. Він забирає ці ресурси з інвентаря, щоб гравець міг використовувати їх для створення предмету.

Додавання у чергу крафту: Після цього додається предмет до черги крафту. Якщо цей предмет вже є в черзі, то збільшує кількість цього предмета у черзі. Якщо його ще немає у черзі, то він створює новий елемент у черзі.

Відображення інформації: Після додавання предмета у чергу, метод відображає інформацію про цей предмет, таку як зображення, кількість та час, необхідний для крафту.

Оновлення інформації про предмет: Нарешті, метод оновлює інформацію про поточний предмет, щоб гравець міг бачити, що він зараз створює.

Будування. У вікні крафту предметів можна створити частини будинку. Після створення, вони з'являються в інвентарі гравця. Перемістивши їх до панелі швидкого доступу, у гравця з'являється можливість поставити матеріали на землі. Можливість поставити наприклад фундамент з'являється тільки тоді, коли йому нічого не заважає (рис. 3.23). При спробі поставити його в дерево чи камінь, матеріал починає світитися червоним кольором, та можливість встановити матеріал зникає (рис. 3.24).

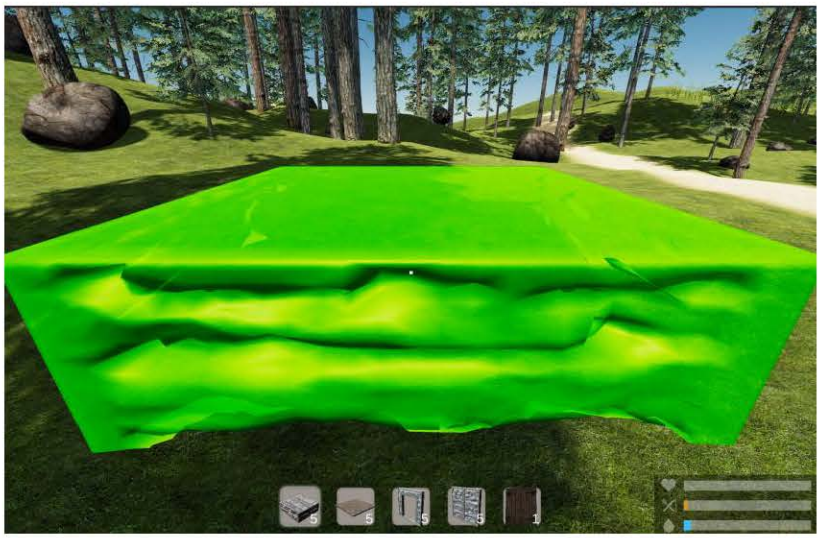

Рисунок 3.23 – Встановлення матеріалу фундамент

Рисунок 3.24 – Спроба встановити у хибному місті

Методи будівництва з коду (Додаток А):

SetConnectionTypeBasedOnBuildingBlock(): Метод встановлює тип з'єднання на основі блока будівництва. Він перевіряє тип поточного будівельного блоку (Foundation, EdgeBlock, Ceiling, Door) і встановлює відповідний тип з'єднання.

RemoveFirstBlockFromCollidersList() і RemoveHouseBlocksFromCollidersList(): Ці методи видаляють перший блок або всі будівельні блоки зі списку колайдерів. Вони використовуються для керування колізіями, коли будівельний блок встановлюється на сцену.

ChangeBuildingBlock(): Змінюється будівельний блок. Він знищує попередній блок і створює новий.

PlaceBlock(): Розміщує будівельний блок на сцені. Він включає функціонал розміщення, активації колайдерів, увімкнення з'єднань, зміни матеріалу блоку, а також виконання певних дій в залежності від типу з'єднання (наприклад, видалення крайового з'єднання або активація окремих колайдерів для дверей).

CanPlace(): Цей метод перевіряє, чи можна розмістити будівельний блок.

OnEnable() і OnDisable(): Ці методи викликаються, коли об'єкт стає активним або неактивним відповідно. Вони викликають методи ChangeBuildingBlock() і Destroy(_buildingBlockOnScene) відповідно.

Відновлення життєвих показників. Якщо гравець перестає слідкувати за життєвими показниками (здоров'я, голод та спрага), гравець втрачає життя та з'являється у точці старту. Тому необхідно розробити можливість відновлювати показники за допомогою їжі та води.

Їжу можна знайти у лісі. На прикладі яблука розглянемо як це працює. Знайшовши яблуко у лісі, гравцю треба натиснути клавішу підбору предмету (за замовчуванням клавіша E). Після чого яблуко потрапляє у інвентар гравця (рис. 3.25).

Рисунок 3.25 – Яблуко на землі та в інвентарі гравця

Перемістивши його у слот швидкого доступу та натиснувши ліву кнопку миші, гравець відновлює собі частину індикатору їжі (рис. 3.26).

а)  б) 

Рисунок 3.26 – а) Індикатор до прийому їжі. б) Індикатор після прийому їжі

Індикатор спраги працює майже так само. Для його відновлення, гравцю потрібно підійти до водойму та натиснути клавішу взаємодії (за замовчуванням клавіша E). Спрацює скрипт, та індикатор спраги оновиться (рис. 3.27).

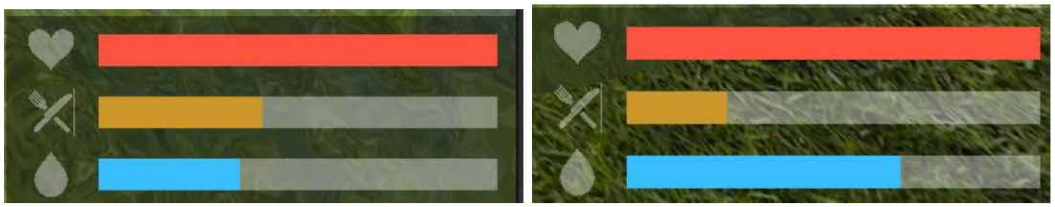

Рисунок 3.27 –Індикатор до та після водопою.

3.5. Висновки до третього розділу

У третьому розділі розглянуто процес розробки гри із використанням технологій штучного інтелекту. Описано етапи створення террейну за допомогою інструменту Terrain в Unity, впровадження ігрових механік, алгоритмів ШІ для управління поведінкою персонажів та інших аспектів гри.

Розробка гри продемонструвала високу ефективність використання Unity та технологій штучного інтелекту для створення складних ігрових середовищ. Основними перевагами є можливість створення детальних і динамічних ландшафтів, забезпечення адаптивної поведінки персонажів та інтерактивних ігрових механік. Це дозволяє підвищити якість ігрового досвіду та залученість користувачів.

Серед недоліків можна виділити значну складність процесу розробки та потребу в великих обчислювальних ресурсах для обробки алгоритмів ШІ та процедурної генерації контенту. Крім того, оптимізація гри для забезпечення її стабільної роботи потребує значних зусиль.

На основі отриманих результатів рекомендовано зосередитись на подальшій оптимізації гри, вдосконаленні алгоритмів ШІ та дослідженні можливостей використання гібридних методів для підвищення ефективності ігрових механік. Це сприятиме створенню ще більш захоплюючих та реалістичних ігрових середовищ, що позитивно вплине на комерційний успіх продукту.

Узагальнюючи, процес створення гри з використанням штучного інтелекту був цікавим досвідом, який допоміг краще розуміти потенціал і виклики застосування ШІ в ігровій промисловості. Впровадження алгоритмів поведінкових дерев і простих методів машинного навчання продемонстрували свою ефективність у створенні реалістичної поведінки неігрових персонажів та динамічного геймплею. Однак цей досвід також показав потребу у подальшої оптимізації, використання більш просунутих технік штучного інтелекту та розширення можливостей взаємодії між гравцем та ігровим світом для досягнення ще більшого занурення та унікального ігрового досвіду.

ВИСНОВКИ

У першому розділі досліджено основні поняття та визначення, пов'язані з комп'ютерними іграми, розглянуто їхню історичну роль та еволюцію. Було проаналізовано різні теорії щодо гри, включаючи підходи Йогана Гейзінга.

Результати аналізу підтвердили, що ігри є складним явищем, яке поєднує розважальні та освітні елементи. Це підкреслює важливість інтеграції новітніх технологій, таких як штучний інтелект, для підвищення цінності ігор. Основною перевагою є здатність створювати багатогранні ігрові досвіди, які задовольняють різноманітні потреби користувачів. Викликом залишається оптимізація складних систем для забезпечення їх ефективної роботи.

У другому розділі проведено аналіз існуючих рішень щодо інтеграції штучного інтелекту в комп'ютерні ігри. Розглянуто різні підходи, включаючи поведінкові дерева, нейронні мережі та алгоритми машинного навчання.

Впровадження ШІ дозволяє створювати реалістичні та захоплюючі ігрові середовища, що підвищує конкурентоспроможність продукту. Недоліком є висока складність розробки та потреба у значних ресурсах для тестування. Рекомендовано використовувати гібридні методи для досягнення балансу між реалістичністю та продуктивністю.

У третьому розділі описано процес розробки гри з використанням штучного інтелекту, зокрема створення террейну в Unity, впровадження ігрових механік та алгоритмів ШІ.

Використання інструменту Terrain в Unity дозволяє створювати детальні ландшафти. Процедурна генерація та алгоритми ШІ сприяють створенню унікальних ігрових середовищ. Основною перевагою є високий рівень залучення користувачів. Недоліками є потреба в значних обчислювальних ресурсах та час для

оптимізації. Рекомендується впровадження додаткових інструментів для оптимізації та масштабування гри.

В ході виконання кваліфікаційної роботи було досягнуто поставлених цілей: розглянуто основні аспекти комп'ютерних ігор, проаналізовано існуючі рішення щодо впровадження штучного інтелекту, та здійснено розробку гри з використанням новітніх технологій.

Результати роботи свідчать про високий потенціал використання штучного інтелекту в комп'ютерних іграх для створення реалістичних та інтерактивних середовищ. Основними перевагами є підвищення комерційного успіху та якості ігрового досвіду. Недоліками є складність розробки та потреба в значних ресурсах для налаштування ШІ-систем. Рекомендується подальша оптимізація та масштабування розробленого продукту, а також дослідження можливостей використання гібридних методів для підвищення ефективності ігрових механік.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Buckland M. Programming Game AI by Example / Mat Buckland., 2018
2. Chastine J. Game Engine Gems 3 / J. Chastine, T. Lum., 2016
3. Goodfellow I. Deep Learning / I. Goodfellow, Y. Bengio, A. Courville., 2016
4. Gregory J. Game Engine Architecture, Third Edition / Jason Gregory., 2019
5. Ismail A. Procedural Generation in Game Design / Ismail., 2016
6. Lane N. Unity Virtual Reality Projects / N. Lane, C. Goerlich., 2016
7. Lee J. Deep Learning for Natural Language Processing / J. Lee, L. Deng., 2018
8. Millington I. Artificial Intelligence for Games / I. Millington, J. Funge., 2016
9. Nystrom R. Game Programming Patterns / Robert Nystrom., 2014
10. Patel S. Unity AI Programming Essentials / S. Patel., 2017
11. Rabin S. Game AI Pro: Collected Wisdom of Game AI Professionals / S. Rabin., 2015
12. Schell J. The Art of Game Design: A Book of Lenses / Jesse Schell., 2015
13. Shaker N. Procedural Content Generation in Games / N. Shaker, J. Togelius, M. J. Nelson., 2016
14. Togelius J. Artificial Intelligence and Games / J. Togelius, G. N. Yannakakis., 2018
15. Unity Manual [Електронний ресурс] – Режим доступу до ресурсу: https://docs.unity3d.com/Manual/index.html.

ДОДАТОК А

Код програми

AnimalAi.cs

```
using UnityEngine;
using UnityEngine.AI;


public class AnimalAI : MonoBehaviour
{
   private NavMeshAgent _navMeshAgent;
   private Animator _animator;
   [SerializeField] private float movementSpeed;

   [SerializeField] private float changePositionTime = 5f;
   [SerializeField] private float moveDistance = 10f;



   private void Start()
   {
      _navMeshAgent = GetComponent<NavMeshAgent>();
      _navMeshAgent.speed = movementSpeed;
      _animator = GetComponent<Animator>();
      InvokeRepeating(nameof(MoveAnimal),changePositionTime,changePositionTime);
   }

   private void Update()
   {
      _animator.SetFloat("Speed",_navMeshAgent.velocity.magnitude/movementSpeed);
   }

   Vector3 RandomNavSphere (float distance) {
      Vector3 randomDirection = UnityEngine.Random.insideUnitSphere * distance;

      randomDirection += transform.position;

      NavMeshHit navHit;

      NavMesh.SamplePosition (randomDirection, out navHit, distance, -1);

      return navHit.position;
   }

   private void MoveAnimal()
   {
      _navMeshAgent.SetDestination(RandomNavSphere(moveDistance));
   }
}
```

BuildingSystem.cs

```
using UnityEngine;

public class BuildingSystem : MonoBehaviour
{
    [Header("Raycast Settings")] [SerializeField]
    private Camera _mainCamera;

    [SerializeField] private float _reachDistance = 2f;

    private Transform _blockConnection;
    private string _connectionType = "";

    private GameObject _buildingBlockOnScene;
    [HideInInspector] public GameObject currentBuildingBlock;

    void Update()
    {
        Ray ray = _mainCamera.ViewportPointToRay(new Vector3(0.5f, 0.5f, 0));
        RaycastHit hit;

        SetConnectionTypeBasedOnBuildingBlock();

        if (Physics.Raycast(ray, out hit, _reachDistance, LayerMask.GetMask("Terrain",
_connectionType)))
        {
            if (_buildingBlockOnScene == null)
            {
                CreateNewBlock();
            }

            Transform buildingBlockTransform = _buildingBlockOnScene.transform;

            _buildingBlockOnScene.GetComponent<ICheckColliders>().CheckColliders();

            if (LayerMask.LayerToName(hit.collider.gameObject.layer) == _connectionType)
            {
                var blockConnection = hit.collider.transform;
                _blockConnection = blockConnection;

                buildingBlockTransform.position = blockConnection.position;
                buildingBlockTransform.rotation = blockConnection.rotation;

                if (_blockConnection.parent.GetComponentInParent<HouseManager>() != null)
                {
                    RemoveHouseBlocksFromCollidersList();
                }
                else
                {
                    RemoveFirstBlockFromCollidersList();
                }
            }
            else
            {
                _blockConnection = null;

                buildingBlockTransform.position = hit.point;
                buildingBlockTransform.rotation =
Quaternion.Euler(buildingBlockTransform.rotation.eulerAngles.x,
```

```
                    _mainCamera.transform.rotation.eulerAngles.y,
buildingBlockTransform.rotation.eulerAngles.z);
            }
        }
        else
        {
            if (_buildingBlockOnScene != null)
            {
                Destroy(_buildingBlockOnScene);
            }
        }
    }

    private void SetConnectionTypeBasedOnBuildingBlock()
    {
        if (currentBuildingBlock.GetComponent<Foundation>() != null)
        {
            _connectionType = "FoundationConnection";
        }
        else if (currentBuildingBlock.GetComponent<EdgeBlock>() != null)
        {
            _connectionType = "EdgeConnection";
        }
        else if (currentBuildingBlock.GetComponent<Ceiling>() != null)
        {
            _connectionType = "CeilingConnection";
        }
        else if (currentBuildingBlock.GetComponent<Door>() != null)
        {
            _connectionType = "DoorConnection";
        }
    }

    private void RemoveFirstBlockFromCollidersList()
    {
        if (_buildingBlockOnScene.GetComponent<BuildingBlock>()

.detectedColliders.Contains(_blockConnection.parent.GetComponent<Collider>()))
        {
            _buildingBlockOnScene.GetComponent<BuildingBlock>()

.detectedColliders.Remove(_blockConnection.parent.GetComponent<Collider>());
        }
    }

    private void RemoveHouseBlocksFromCollidersList()
    {
        foreach (BuildingBlock buildingBlock in
_blockConnection.parent.GetComponentInParent<HouseManager>()
            .buildingBlocks)
        {
            if (_buildingBlockOnScene.GetComponent<BuildingBlock>()
                .detectedColliders.Contains(buildingBlock.GetComponent<Collider>()))
            {
                _buildingBlockOnScene.GetComponent<BuildingBlock>()
                    .detectedColliders.Remove(buildingBlock.GetComponent<Collider>());
            }
        }
```

```
    }

    public void ChangeBuildingBlock()
    {
        if (currentBuildingBlock == null) return;

        Destroy(_buildingBlockOnScene);
        CreateNewBlock();
    }

    private void CreateNewBlock()
    {
        if (currentBuildingBlock == null)
            return;

        _buildingBlockOnScene = Instantiate(currentBuildingBlock);
        if(_buildingBlockOnScene.GetComponent<Collider>() != null)
            _buildingBlockOnScene.GetComponent<Collider>().enabled = false;
    }

    public void PlaceBlock()
    {
        _buildingBlockOnScene.GetComponent<BuildingBlock>().isPlaced = true;

        if(_buildingBlockOnScene.GetComponent<Collider>() != null)
            _buildingBlockOnScene.GetComponent<Collider>().enabled = true;

        if(_buildingBlockOnScene.GetComponent<IHaveConnections>() != null)
            _buildingBlockOnScene.GetComponent<IHaveConnections>().TurnOnConnections();

        _buildingBlockOnScene.GetComponent<BuildingBlock>().ChangeToBlockMaterial();

        if (_blockConnection != null)
        {
            // Перша частина
            if (_blockConnection.parent.parent == null)
            {
                GameObject house = Instantiate(new GameObject("House"),
_buildingBlockOnScene.transform.position,
                    Quaternion.identity);
                house.AddComponent<HouseManager>();
                _blockConnection.parent.SetParent(house.transform);
            }
            // Друга частина
            _buildingBlockOnScene.transform.position = _blockConnection.position;
            _buildingBlockOnScene.transform.rotation = _blockConnection.rotation;
            _buildingBlockOnScene.transform.SetParent(_blockConnection.parent.parent);

            // Третя частина
            switch (_connectionType)
            {
                case "EdgeConnection":
                    if (_blockConnection.parent.GetComponent<Foundation>() != null)
                    {
                        GameObject edgeConnection = _blockConnection.gameObject;

_blockConnection.parent.GetComponent<Foundation>().edgeConnections.Remove(edgeConnection)
;
```

```
                        Destroy(edgeConnection);
                    }

                    break;
                case "DoorConnection":

_buildingBlockOnScene.GetComponent<Door>().TurnOnSeparateDoorColliders();
                    Destroy(_blockConnection.gameObject);
                    break;
            }

_buildingBlockOnScene.transform.parent.GetComponent<HouseManager>().UpdateBuildingBlocks(
);

        }

        CreateNewBlock();
    }

    public bool CanPlace()
    {
        if (_buildingBlockOnScene == null) return false;

        return _buildingBlockOnScene.GetComponent<BuildingBlock>().canPlace;
    }

    private void OnEnable()
    {
        ChangeBuildingBlock();
    }

    private void OnDisable()
    {
        Destroy(_buildingBlockOnScene);
    }
}
```

CraftManager

```
public class CraftManager : MonoBehaviour
{
    public bool isOpened;
    public GameObject craftingPanel;
    public GameObject inventoryAndClothingPanel;


    public Transform craftItemsPanel;
    public GameObject craftItemButtonPrefab;

    public GameObject UIBG;
    public GameObject crosshair;
    public CinemachineVirtualCamera CVC;
    public Button craftBtn;
    public FillCraftItemDetails currentCraftItem;

    public KeyCode openCloseCraftButton;
```

```
    public List<CraftScriptableObject> allCrafts;

    [Header("Craft Item Details")]
    public TMP_Text craftItemName;
    public TMP_Text craftItemDescription;
    public Image craftItemImage;
    public TMP_Text craftItemDuration;
    public TMP_Text craftItemAmount;
    // Start is called before the first frame update
    void Start()
    {
        GameObject craftItemButton = Instantiate(craftItemButtonPrefab, craftItemsPanel);
        craftItemButton.GetComponent<Image>().sprite = allCrafts[0].finalCraft.icon;
        craftItemButton.GetComponent<FillCraftItemDetails>().currentCraftItem =
allCrafts[0];
        craftItemButton.GetComponent<FillCraftItemDetails>().FillItemDetails();
        Destroy(craftItemButton);

        craftingPanel.gameObject.SetActive(false);
    }
    public void FillItemDetailsHelper()
    {
        currentCraftItem.FillItemDetails();
    }
    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(openCloseCraftButton))
        {
            isOpened = !isOpened;

            GetComponent<InventoryManager>().isOpened = false;
            inventoryAndClothingPanel.gameObject.SetActive(false);
            if (isOpened)
            {
                craftingPanel.SetActive(true);
                UIBG.SetActive(true);
                crosshair.SetActive(false);


CVC.GetCinemachineComponent<CinemachinePOV>().m_HorizontalAxis.m_InputAxisName = "";

CVC.GetCinemachineComponent<CinemachinePOV>().m_VerticalAxis.m_InputAxisName = "";

CVC.GetCinemachineComponent<CinemachinePOV>().m_HorizontalAxis.m_InputAxisValue = 0;

CVC.GetCinemachineComponent<CinemachinePOV>().m_VerticalAxis.m_InputAxisValue = 0;
                // Прекрепляем курсор к середине экрана
                Cursor.lockState = CursorLockMode.None;
                // и делаем его невидимым
                Cursor.visible = true;
            }
            else
            {
                craftingPanel.SetActive(false);
                UIBG.SetActive(false);
                crosshair.SetActive(true);
```

```
CVC.GetCinemachineComponent<CinemachinePOV>().m_HorizontalAxis.m_InputAxisName = "Mouse
X";

CVC.GetCinemachineComponent<CinemachinePOV>().m_VerticalAxis.m_InputAxisName = "Mouse Y";
                // Прекрепляем курсор к середине экрана
                Cursor.lockState = CursorLockMode.Locked;
                // и делаем его невидимым
                Cursor.visible = false;
            }

        }

    }

    public void LoadCraftItems(string craftType)
    {
        for (int i = 0; i < craftItemsPanel.childCount; i++)
        {
            Destroy(craftItemsPanel.GetChild(i).gameObject);
        }
        foreach (CraftScriptableObject cso in allCrafts)
        {
            if (cso.craftType.ToString().ToLower() == craftType.ToLower())
            {
                GameObject craftItemButton = Instantiate(craftItemButtonPrefab,
craftItemsPanel);
                craftItemButton.GetComponent<Image>().sprite = cso.finalCraft.icon;
                craftItemButton.GetComponent<FillCraftItemDetails>().currentCraftItem =
cso;
            }
        }
    }
}
```

InventoryManager.cs

```
public class InventoryManager : MonoBehaviour
{
    public GameObject UIBG;
    public GameObject crosshair;
    public Transform inventoryPanel;
    public Transform inventoryAndClothingPanel;
    public GameObject craftPanel;
    public Transform quickslotPanel;
    public List<InventorySlot> slots = new List<InventorySlot>();
    public bool isOpened;
    public float reachDistance = 3f;
    private Camera mainCamera;
    public CinemachineVirtualCamera CVC;
    private CraftManager craftManager;
    [SerializeField] private Transform player;
    [SerializeField] private List<ClothAdder> _clothAdders = new List<ClothAdder>();
    // Start is called before the first frame update
```

```
    private void Awake()
    {
        UIBG.SetActive(true);
    }
    void Start()
    {
        mainCamera = Camera.main;
        craftManager = FindObjectOfType<CraftManager>();


slots.AddRange(inventoryAndClothingPanel.GetComponentsInChildren<InventorySlot>());

        for (int i = 0; i < quickslotPanel.childCount; i++)
        {
            if (quickslotPanel.GetChild(i).GetComponent<InventorySlot>() != null)
            {
                slots.Add(quickslotPanel.GetChild(i).GetComponent<InventorySlot>());
            }
        }

        UIBG.SetActive(false);
        inventoryAndClothingPanel.gameObject.SetActive(false);//new line
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.I))
        {
            isOpened = !isOpened;
            craftPanel.gameObject.SetActive(false);
            craftManager.isOpened = false;
            if (isOpened)
            {
                UIBG.SetActive(true);
                inventoryAndClothingPanel.gameObject.SetActive(true); // new line
                crosshair.SetActive(false);
                CinemachinePOV pov = CVC.GetCinemachineComponent<CinemachinePOV>();
                pov.m_HorizontalAxis.m_InputAxisName = "";
                pov.m_VerticalAxis.m_InputAxisName = "";
                pov.m_HorizontalAxis.m_InputAxisValue = 0;
                pov.m_VerticalAxis.m_InputAxisValue = 0;
                // Прикріплюємо курсор до середини екрана
                Cursor.lockState = CursorLockMode.None;
                // і робимо його невидимим
                Cursor.visible = true;
            }
            else
            {
                UIBG.SetActive(false);
                inventoryAndClothingPanel.gameObject.SetActive(false); // new line
                crosshair.SetActive(true);

CVC.GetCinemachineComponent<CinemachinePOV>().m_HorizontalAxis.m_InputAxisName = "Mouse
X";

CVC.GetCinemachineComponent<CinemachinePOV>().m_VerticalAxis.m_InputAxisName = "Mouse Y";
                // Прикріплюємо курсор до середини екрана
```

```
                Cursor.lockState = CursorLockMode.Locked;
                // і робимо його невидимим
                Cursor.visible = false;

                DragAndDropItem[] dadi = FindObjectsOfType<DragAndDropItem>();
                foreach(DragAndDropItem slot in dadi)
                {
                    slot.ReturnBackToSlot();
                }

            }
        }
        Ray ray = mainCamera.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if (Input.GetKeyDown(KeyCode.E))
        {
            if (Physics.Raycast(ray, out hit, reachDistance))
            {
                if (hit.collider.gameObject.GetComponent<Item>() != null)
                {
                    AddItem(hit.collider.gameObject.GetComponent<Item>().item,
hit.collider.gameObject.GetComponent<Item>().amount);
                    craftManager.currentCraftItem.FillItemDetails();
                    Destroy(hit.collider.gameObject);
                }
            }
        }
    }

    public void RemoveItemFromSlot(int slotId)
    {
        InventorySlot slot = slots[slotId];

        if (slot.clothType != ClothType.None && !slot.isEmpty)
        {
            foreach (ClothAdder clothAdder in _clothAdders)
            {
                clothAdder.RemoveClothes(slot.item.clothingPrefab);
            }
        }
        slot.item = null;
        slot.isEmpty = true;
        slot.amount = 0;
        slot.iconGO.GetComponent<Image>().color = new Color(1, 1, 1, 0);
        slot.iconGO.GetComponent<Image>().sprite = null;
        slot.itemAmountText.text = "";
    }
    public void AddItemToSlot(ItemScriptableObject _item, int _amount, int slotId)
    {
        InventorySlot slot = slots[slotId];
        slot.item = _item;
        slot.isEmpty = false;
        slot.SetIcon(_item.icon);

        if (_amount <= _item.maximumAmount)
        {
            slot.amount = _amount;
```

```
            if (slot.item.maximumAmount != 1) // added this if statement for single items
            {
                slot.itemAmountText.text = slot.amount.ToString();
            }
        }
        else
        {
            slot.amount = _item.maximumAmount;
            _amount -= _item.maximumAmount;
            if (slot.item.maximumAmount != 1) // added this if statement for single items
            {
                slot.itemAmountText.text = slot.amount.ToString();
            }
        }

        if (slot.clothType != ClothType.None)
        {
            foreach (ClothAdder clothAdder in _clothAdders)
            {
                clothAdder.addClothes(slot.item.clothingPrefab);
            }
        }
    }
    public void AddItem(ItemScriptableObject _item, int _amount)
    {

        int amount = _amount;
        foreach (InventorySlot slot in slots)
        {
            // Стакаємо предмети разом
            // У слоті вже є цей предмет
            if (slot.item == _item)
            {
                if (slot.amount + amount <= _item.maximumAmount) {
                    slot.amount += amount;
                    slot.itemAmountText.text = slot.amount.ToString();
                    return;
                }
                else
                {
                    amount -= _item.maximumAmount - slot.amount;
                    slot.amount = _item.maximumAmount;
                    slot.itemAmountText.text = slot.amount.ToString();
                }
                //break;
                continue;
            }
        }

        bool allFull = true;
        foreach (InventorySlot inventorySlot in slots)
        {
            if (inventorySlot.isEmpty)
            {
                allFull = false;
                break;
            }
        }
```

```
        if (allFull)
        {
            GameObject itemObject = Instantiate(_item.itemPrefab, player.position +
Vector3.up + player.forward, Quaternion.identity);
            itemObject.GetComponent<Item>().amount = _amount;
        }

        foreach (InventorySlot slot in slots)
        {
            if (amount <= 0)
                return;
            // додаємо предмети у вільні комірки
            if(slot.isEmpty == true)
            {
                slot.item = _item;
                //slot.amount = amount;
                slot.isEmpty = false;
                slot.SetIcon(_item.icon);

                if (amount <= _item.maximumAmount)
                {
                    slot.amount = amount;
                    if (slot.item.maximumAmount != 1) // added this if statement for
single items
                    {
                        slot.itemAmountText.text = slot.amount.ToString();
                    }
                    break;
                }
                else
                {
                    slot.amount = _item.maximumAmount;
                    amount -= _item.maximumAmount;
                    if (slot.item.maximumAmount != 1) // added this if statement for
single items
                    {
                        slot.itemAmountText.text = slot.amount.ToString();
                    }
                }

                allFull = true;
                foreach (InventorySlot inventorySlot in slots)
                {
                    if (inventorySlot.isEmpty)
                    {
                        allFull = false;
                        break;
                    }
                }
                if (allFull)
                {
                    GameObject itemObject = Instantiate(_item.itemPrefab, player.position
+ Vector3.up + player.forward, Quaternion.identity);
                    itemObject.GetComponent<Item>().amount = amount;
                    Debug.Log("Throw out");
                    return;
                }
```

```
                // continue;
            }
        }
    }
}
```

MenuManager.cs

```
ublic class MenuManager : MonoBehaviour
{
    [SerializeField] private CinemachineVirtualCamera _mainCamera;
    [SerializeField] private CinemachineVirtualCamera _settingsCamera;
    [SerializeField] private GameObject _menuPanel;
    [SerializeField] private GameObject _settingsPanel;
    public float _currentSensitivity;
    private int _currentResolutionIndex;


    #region SettingsVars
    [Header("Settings Menu")]
    [SerializeField] private AudioMixer _audioMixer;
    [SerializeField] private TMP_Dropdown _resolutionDropdown;
    [SerializeField] private TMP_Dropdown _qualityDropdown;
    [SerializeField] private Toggle _toggle;

    private Resolution[] _resolutions;
    [SerializeField] private Transform _moveableNailTransform;

     #endregion
     // max x value -1.841574
     // min x value -0.136
     private void Start()
     {
         PopulateResolutionDropdown();
         LoadSettings(_currentResolutionIndex);
     }

     #region Settings Funcs
        private void PopulateResolutionDropdown()
        {
            _resolutionDropdown.ClearOptions();
            List<string> options = new List<string>();
            _resolutions = Screen.resolutions;
            _currentResolutionIndex = 0;

            for (int i = 0; i < _resolutions.Length; i++)
            {
                string option = _resolutions[i].width + "x" + _resolutions[i].height;
                options.Add(option);
                if (_resolutions[i].width == Screen.currentResolution.width &&
                    _resolutions[i].height == Screen.currentResolution.height)
                    _currentResolutionIndex = i;
            }
            _resolutionDropdown.AddOptions(options);
            _resolutionDropdown.RefreshShownValue();
        }
```

```
        public void SetFullscreen(bool isFullscreen)
        {
            Screen.fullScreen = isFullscreen;
        }
        public void SetResolution(int resolutionIndex)
        {
            Resolution resolution = _resolutions[resolutionIndex];
            Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
        }
        public void SetQuality(int qualityIndex)
        {
            QualitySettings.SetQualityLevel(qualityIndex);
        }
        public void SetVolume()
        {
            float desiredVolume = Mathf.InverseLerp(-0.136f, -1.841574f,
_moveableNailTransform.localPosition.x);
            if (desiredVolume <= 0.0001f)
                desiredVolume = 0.0001f;

            _audioMixer.SetFloat("MasterVolume",Mathf.Log10(desiredVolume) * 20);
        }
        public void SetSensitivity(float desiredSensitivity)
        {
            PlayerPrefs.SetFloat("Sensitivity",desiredSensitivity);
            _currentSensitivity = desiredSensitivity;
        }
        public void SaveSettings()
        {
            PlayerPrefs.SetInt("QualitySettings", _qualityDropdown.value);
            PlayerPrefs.SetInt("Resolution", _resolutionDropdown.value);
            PlayerPrefs.SetInt("Fullscreen", System.Convert.ToInt32(Screen.fullScreen));

            float volume;
            _audioMixer.GetFloat("MasterVolume", out volume);
            PlayerPrefs.SetFloat("Volume", volume);
        }
        public void LoadSettings(int currentResolutionIndex)
        {
            if (PlayerPrefs.HasKey("QualitySettings"))
                _qualityDropdown.value = PlayerPrefs.GetInt("QualitySettings");
            else
                _qualityDropdown.value = 3;

            SetQuality(_qualityDropdown.value);

            if (PlayerPrefs.HasKey("Resolution"))
                _resolutionDropdown.value = PlayerPrefs.GetInt("Resolution");
            else
                _resolutionDropdown.value = currentResolutionIndex;

            SetResolution(_resolutionDropdown.value);

            if (PlayerPrefs.HasKey("Fullscreen"))
                Screen.fullScreen =
System.Convert.ToBoolean(PlayerPrefs.GetInt("Fullscreen"));
            else
                Screen.fullScreen = true;
```

```
            _toggle.isOn = Screen.fullScreen;

            if (PlayerPrefs.HasKey("Volume"))
            {
                _audioMixer.SetFloat("MasterVolume", PlayerPrefs.GetFloat("Volume"));
                _moveableNailTransform.localPosition =
                    new Vector3(Mathf.Lerp(-0.136f, -1.841574f,
Mathf.Pow(10,PlayerPrefs.GetFloat("Volume")/20)), 0, 0);
            }
            else
            {
                _audioMixer.SetFloat("MasterVolume", 0);
                _moveableNailTransform.localPosition =
                    new Vector3(Mathf.Lerp(-0.136f, -1.841574f, 1), 0, 0);
            }
        }
        #endregion

        public void Play()
        {
            SceneManager.LoadScene(1);
        }

        public void Quit()
        {
            Application.Quit();
            Debug.Log("Quit game");
        }

        public void Settings()
        {
            _mainCamera.Priority = 0;
            _menuPanel.SetActive(false);
            _settingsCamera.Priority = 1;

            StartCoroutine(OpenSettingsAfterTime());
        }
        public void Back()
        {
            _settingsCamera.Priority = 0;
            _mainCamera.Priority = 1;
            _settingsPanel.SetActive(false);

            StartCoroutine(OpenMainMenuAfterTime());
        }

        IEnumerator OpenSettingsAfterTime()
        {
            yield return new WaitForSeconds(2);
            _settingsPanel.SetActive(true);
        }
        IEnumerator OpenMainMenuAfterTime()
        {
            yield return new WaitForSeconds(2);
            _menuPanel.SetActive(true);
        }
}
```

MyFloater.cs

```
public class MyFloater : MonoBehaviour
{
    private Rigidbody rigidbody;
    public float floatUpSpeedLimit = 1.15f;
    public float floatUpSpeed = 1f;

    private void Start()
    {
        rigidbody = GetComponent<Rigidbody>();
    }
    private void OnTriggerStay(Collider other)
    {
        if(other.gameObject.layer == 4)
        {
            float difference = (other.transform.position.y - transform.position.y) *
floatUpSpeed;
            rigidbody.AddForce(new Vector3(0f, Mathf.Clamp((Mathf.Abs(Physics.gravity.y)
* difference),0, Mathf.Abs(Physics.gravity.y) * floatUpSpeedLimit),0f),
ForceMode.Acceleration);
            rigidbody.drag = 0.99f;
            rigidbody.angularDrag = 0.8f;
        }
    }
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.layer == 4)
        {
            rigidbody.drag = 0f;
            rigidbody.angularDrag = 0f;
        }
    }
}
```

CustomCharacterController.cs

```
public class CustomCharacterController : MonoBehaviour
{
    public Animator anim;
    public Rigidbody rig;
    public Transform mainCamera;
    public float jumpForce = 3.5f;
    public float walkingSpeed = 2f;
    public float runningSpeed = 6f;
    public float swimmingSpeed = 1.5f;
    public float currentSpeed;
    private float animationInterpolation = 1f;
    public InventoryManager inventoryManager;
    public QuickslotInventory quickslotInventory;
    public CraftManager craftManager;
    public Indicators indicators;
```

```
    public Transform aimTarget;
    public Transform hitTarget;
    public Vector3 hitTargetOffset;

    public bool isNeckUnderWater;
    [SerializeField] private Transform _neckObj;
    [SerializeField] private float _waterLevel = -1;

    [SerializeField] private Transform _groundObj;
    // Start is called before the first frame update
    void Start()
    {
        // Прикріплюємо курсор до середини екрана
        Cursor.lockState = CursorLockMode.Locked;
        // і робимо його невидимим
        Cursor.visible = false;

    }

    bool isGrounded() => Physics.Raycast(_groundObj.position, Vector3.down, 0.1f,
LayerMask.GetMask("Default","Terrain"));

    bool IsSwimming()
    {
        isNeckUnderWater = _waterLevel > _neckObj.position.y;

        if (isGrounded() && !isNeckUnderWater)
        {
            return false;
        }
        else if (isGrounded() && isNeckUnderWater)
        {
            return true;
        }
        else if (!isGrounded() && isNeckUnderWater)
        {
            return true;
        }

        return false;
    }

    void Run()
    {
        animationInterpolation = Mathf.Lerp(animationInterpolation, 1.5f, Time.deltaTime
* 3);
        anim.SetFloat("x", Input.GetAxis("Horizontal") * animationInterpolation);
        anim.SetFloat("y", Input.GetAxis("Vertical") * animationInterpolation);

        currentSpeed = Mathf.Lerp(currentSpeed, runningSpeed, Time.deltaTime * 3);
    }
    void Walk()
    {
        // Mathf.Lerp - відповідає за те, щоб кожен кадр число animationInterpolation(у
цьому випадку) наближалося до числа 1 зі швидкістю Time.deltaTime * 3.
        // Time.deltaTime - це час між цим кадром і попереднім кадром. Це дає змогу
плавно переходити з одного числа до другого.
```

```
        animationInterpolation = Mathf.Lerp(animationInterpolation, 1f, Time.deltaTime *
3);
        anim.SetFloat("x", Input.GetAxis("Horizontal") * animationInterpolation);
        anim.SetFloat("y", Input.GetAxis("Vertical") * animationInterpolation);

        currentSpeed = Mathf.Lerp(currentSpeed, walkingSpeed, Time.deltaTime * 3);
    }

    void Swim()
    {
        animationInterpolation = Mathf.Lerp(animationInterpolation, 1f, Time.deltaTime *
3);
        anim.SetFloat("x", Input.GetAxis("Horizontal") * animationInterpolation);
        anim.SetFloat("y", Input.GetAxis("Vertical") * animationInterpolation);

        currentSpeed = Mathf.Lerp(currentSpeed, swimmingSpeed, Time.deltaTime * 3);
    }

    public void ChangeLayerWeight(float newLayerWeight)
    {
        StartCoroutine(SmoothLayerWeightChange(anim.GetLayerWeight(1), newLayerWeight,
0.3f));
    }

    IEnumerator SmoothLayerWeightChange(float oldWeight, float newWeight, float
changeDuration)
    {
        float elapsed = 0f;
        while(elapsed < changeDuration)
        {
            float currentWeight = Mathf.Lerp(oldWeight, newWeight, elapsed /
changeDuration);
            anim.SetLayerWeight(1, currentWeight);
            elapsed += Time.deltaTime;
            yield return null;
        }
        anim.SetLayerWeight(1, newWeight);
    }

    private void Update()
    {
        // Встановлюємо поворот персонажа коли камера повертається
        transform.rotation =
Quaternion.Euler(transform.rotation.eulerAngles.x,mainCamera.rotation.eulerAngles.y,
transform.rotation.eulerAngles.z);
        if (IsSwimming())
        {
            rig.useGravity = false;
            Swim();
            anim.SetBool("isSwimming",true);
        }
        else
        {
            rig.useGravity = true;
            if (Input.GetKeyDown(KeyCode.Mouse0))
            {
                if (quickslotInventory.activeSlot != null)
                {
```

```
                    if (quickslotInventory.activeSlot.item != null)
                    {
                        if (quickslotInventory.activeSlot.item.itemType ==
ItemType.Instrument)
                        {
                            if (inventoryManager.isOpened == false &&
craftManager.isOpened == false)
                            {
                                anim.SetBool("Hit", true);
                            }
                        }
                    }
                }
            }

            if (Input.GetKeyUp(KeyCode.Mouse0))
            {
                anim.SetBool("Hit", false);
            }

            // Чи затиснуті кнопки W і Shift?
            if (Input.GetKey(KeyCode.W) && Input.GetKey(KeyCode.LeftShift))
            {
                // Чи затиснуті ще кнопки A S D?
                if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.S) ||
Input.GetKey(KeyCode.D))
                {
                    // Якщо так, то ми йдемо пішки
                    Walk();
                }
                // Якщо ні, то тоді біжимо!
                else
                {
                    Run();
                }
            }
            // Якщо W & Shift не затиснуті, то ми просто йдемо пішки
            else
            {
                Walk();
            }

            //Якщо затиснутий пробіл, то в аніматорі відправляємо повідомлення тригеру,
який активує анімацію стрибка
            if (Input.GetKeyDown(KeyCode.Space))
            {
                anim.SetTrigger("Jump");
            }
        }

        Ray desiredTargetRay = mainCamera.GetComponent<Camera>()
                .ScreenPointToRay(new Vector2(Screen.width / 2, Screen.height / 2));
            Vector3 desiredTargetPosition =
                desiredTargetRay.origin + desiredTargetRay.direction * 1.5f; // changed
from 0.7 to 1.5
            aimTarget.position = desiredTargetPosition;
```

```
            //hitTarget.position = new Vector3(desiredTargetPosition.x +
hitTargetOffset.x, desiredTargetPosition.y + hitTargetOffset.y, desiredTargetPosition.z +
hitTargetOffset.z);

    }

    // Update is called once per frame
    void FixedUpdate()
    {
        // Задаємо рух персонажа залежно від напрямку, в який дивиться камера
        // Зберігаємо напрямок вперед і вправо від камери
        Vector3 camF = mainCamera.forward;
        Vector3 camR = mainCamera.right;
        camF.y = 0;
        camR.y = 0;
        // Щоб напрямки вперед і вправо не залежали від того, чи дивиться камера вгору
або вниз, інакше коли ми дивимося вперед, персонаж буде йти швидше, ніж коли дивиться
вгору або вниз.

        Vector3 movingVector;
        if (IsSwimming())
        {
            movingVector =
                Vector3.ClampMagnitude(
                    camF.normalized * Input.GetAxis("Vertical") * currentSpeed +
                    camR.normalized * Input.GetAxis("Horizontal") * currentSpeed,
currentSpeed);
            Debug.DrawRay(mainCamera.position, movingVector, Color.blue);
        }
        else
        {
            // Множимо наше натискання на кнопки W & S на напрямок камери вперед і
додаємо до натискань на кнопки A & D і множимо на напрямок камери вправо
            movingVector =
                Vector3.ClampMagnitude(
                    camF.normalized * Input.GetAxis("Vertical") * currentSpeed +
                    camR.normalized * Input.GetAxis("Horizontal") * currentSpeed,
currentSpeed);
        }
        // Magnitude - це довжина вектора. ділимо довжину на currentSpeed так як множимо
цей вектор на currentSpeed на 86 рядку.
        anim.SetFloat("magnitude", movingVector.magnitude / currentSpeed);
        // Рухаємо персонажа! Встановлюємо рух тільки по x & z тому що ми не хочемо щоб
наш персонаж злітав у повітря
        rig.velocity = new Vector3(movingVector.x, rig.velocity.y, movingVector.z);
        rig.angularVelocity = Vector3.zero;
    }
    public void Jump()
    {
        // Виконуємо стрибок за командою анімації.
        rig.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
    public void Hit()
    {
        foreach(Transform item in quickslotInventory.allWeapons)
        {
            if (item.gameObject.activeSelf)
            {
```

```
                item.GetComponent<GatherResources>().GatherResource();
                craftManager.currentCraftItem.FillItemDetails();
            }
        }
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.layer == LayerMask.NameToLayer("Water"))
        {
            indicators.isInWater = true;
            _waterLevel = other.transform.position.y; //+
other.GetComponent<BoxCollider>().size.y/2;
        }
    }
    private void OnTriggerExit(Collider other)
    {
        if (other.gameObject.layer == LayerMask.NameToLayer("Water"))
        {
            indicators.isInWater = false;
            _waterLevel = -1;
            //isNeckUnderWater = false;
        }
    }
}
```