МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА МЕРЕЖЕВОЇ ГРИ З ВИКОРИСТАННЯМ ІГРОВОГО РУШІЯ UNITY»**

Виконав: студент 4 курсу, групи 6.1261
спеціальності 126 Інформаційні системи та технології
(шифр і назва спеціальності)
освітньої програми Інформаційні системи та технології
(назва освітньої програми)
А.В. Міщенко
(ініціали та прізвище)

Керівник доцент кафедри комп'ютерних наук, PhD, Калюжняк А.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент декан математичного факультету, професор, д.т.н. Гоменюк С.І.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 126 Інформаційні системи та технології
(шифр і назва)

Освітня програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук,
д.т.н., доцент

_____ Шило Г.М.
(підпис)

" ____ " _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТЦІ

Міщенку Артему Віталійовичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка мережевої гри з використанням ігрового рушія Unity

керівник роботи Калюжняк Анастасія Вікторівна, PhD
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 26.05.2025

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка мережевої гри з використанням ігрового рушія Unity

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 01.01.2025

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	01.01.2025	
2.	Збір вихідних даних.	10.01.2025	
3.	Обробка методичних та теоретичних	12.02.2025	
	джерел.		
4.	Розробка першого та другого розділу.	15.03.2025	
5.	Розробка третього розділу.	20.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	20.06.2025	

Студент _____ А.В. Міщенко
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.В. Калюжняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г.Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка мережевої гри з використанням ігрового рушія Unity»: 54 с., 51 рис., 20 джерел.

ВІДЕОГРА, ГЕЙМДИЗАЙН, ІГРОВИЙ РУШІЙ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, МЕРЕЖЕВЕ З'ЄДНАННЯ.

Об'єкт дослідження – процес розробки багатокористувацьких ігор за допомогою ігрового рушія Unity.

Мета роботи: розробити мережеву гру з використанням ігрового рушія Unity, що забезпечує стабільну багатокористувацьку взаємодію через інтернет. Робота спрямована на дослідження різних аспектів розробки відеоігор та використання різноманітних інструментів рушія.

Метод дослідження – аналіз та синтез.

У ході виконання кваліфікаційної роботи було створено функціональну багатокористувацьку кооперативну гру на Unity, де гравці виступають у ролі техніків, що забезпечують проведення музичного концерту. Реалізовано основні ігрові механіки, такі як рух у 3D-просторі, інвентар, ремонт інструментів, мережеву синхронізацію та динамічний рахунок, що адаптується до дій гравців. Отримані результати можуть бути використані для навчання, демонстрацій або як основа для подальшої розробки комерційного проєкту.

SUMMARY

Bachelor’s Qualifying Theses « Development of a Network Game Using the Unity Game Engine»: 54 pages, 51 figures, 20 references.

VIDEO GAME, GAME DESIGN, GAME ENGINE, CLIENT-SERVER ARCHITECTURE, NETWORK CONNECTION.

Object of the study – the process of developing multiplayer games using Unity game engine.

Aim of the study: to develop a network game using Unity game engine that provides stable multiplayer interaction via Internet. The work focuses on researching various aspects of video game development and utilizing diverse engine tools.

Methods of research – analysis of existing solutions, synthesis of original approach, experimental testing of network mechanics.

During the qualification project, a functional multiplayer cooperative game was developed using Unity, where players take on the roles of technicians ensuring the smooth running of a music concert. Key gameplay mechanics were implemented, including 3D movement, inventory management, instrument repair, network synchronization, and a dynamic score that adapts to player performance. The results can be used for educational purposes, demonstrations, or as a foundation for further development of a commercial project.

ЗМІСТ

Завдання на кваліфікаційну роботу……..…..2
Реферат 4
Summary 5
Вступ..... 8
1 Аналіз предметної області та постановка задачі 9
1.1 Аналіз предметної області 9
1.2 Вибір технологій для створення відеогри 13
1.3 Постановка задачі 17
1.4 Висновок за першим розділом..... 17
2 Проєктування інформаційної системи..... 18
2.1 Вимоги до системи..... 18
2.2 Діаграма класів..... 20
2.3 Мережева логіка..... 22
2.4 Висновок за другим розділом 23
3 Програмна реалізація..... 24
3.1 Створення проєкту..... 24
3.2 Рух гравця 26
3.3 Поворот гравця..... 29
3.4 Взаємодія гравця 29
3.5 Предмети..... 32
3.6 Об’єкти для зберігання предметів..... 34
3.6.1 Сховища..... 35
3.6.2 Транзитні столи..... 36
3.7 Музиканти..... 37
3.8 InstrumentManager 39
3.9 Ремонт інструментів 40
3.10 Музика..... 42

3.11 Інтерфейс 43
3.12 Мережева логіка 45
3.12.1 NetworkManager 45
3.12.2 Меню підключення 46
3.12.3 Синхронізація подій 47
3.12.4 Ігровий рахунок 48
3.13 Висновок за третім розділом 50
Висновки 51
Перелік посилань 53

ВСТУП

На сьогоднішній день, індустрія розваг стала невід'ємною частиною життя. Сучасна людина кожен день зіштовхується зі стресом та іншими проблемами і розваги у будь-якому вигляді є одним із способів зняти напругу та відволіктися від реальності.

Сьогодні існує багато способів розважитися: від прослуховування музики до перегляду фільмів. Однак за останні роки в індустрії розваг статистично домінують відеоігри. Ігри – порівняно сучасний вид розваг. Але, на сьогоднішній день це не лише спосіб вбити час, але й значна частина культури. Вони об'єднують людей різного віку та соціального статусу і виконують соціальну функцію, дозволяючи взаємодіяти не виходячи з дому. Ігри активно інтегруються в наше життя та набувають важливості навіть у таких сферах як освіта. Активно розвиваючись та адаптуючи нові технології відеоігри залишаються актуальними у стрімкому потоці розважального сектору.

Актуальність теми дослідження зумовлена стрімким розвитком ігрової індустрії, зокрема сегменту мережевих багатокористувацьких ігор. Зростаючий попит на інтерактивні розваги та нові форми соціальної взаємодії вимагають сучасних підходів до створення ігор. Проєктування та розробка багатокористувацьких застосунків із використанням рушія Unity дозволяє ефективно реалізовувати сучасні рішення, що відповідають потребам користувача.

Основною метою кваліфікаційної роботи виступає проєктування та розробка мережевої відеогри для декількох гравців з використанням інструментарію ігрового рушія Unity.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області

Відеогра – це електронна гра, у якій гравець використовує інтерфейс або пристрій введення, такий як клавіатура або комп’ютерна мишка, для отримання зворотної інформації та взаємодії [1]. Існує безліч видів та жанрів комп’ютерних ігор: починаючи від простих аркад, які існують вже майже 50 років, і закінчуючи онлайн іграми на сотні гравців.

Комп’ютерні ігри можна класифікувати за типом взаємодії гравців – одиночні та мережеві. Розберемо кожен з видів детальніше.

Мережеві ігри – це відеоігри, у яких два і більше користувачів можуть взаємодіяти один з одним у реальному часі використовуючи мережеве з’єднання. Вони дозволяють гравцям об’єднуватися у команди, змагатися між собою або спільно досліджувати ігровий світ незалежно від їх фізичного розташування [2].

Ігри з мережевим з’єднанням підрозділяються на:

– кооперативні, у яких основою ігрового процесу є спільне виконання завдань;
– гравець проти гравця;
– масова багатокористувацька гра.

Прикладом таких ігор є It Takes Two, Dota 2, League Of Legends та Counter-Strike 2.

Одиночні, або одноосібні ігри розраховані лише на одного гравця. Тому, основна увага ігор такого типу приділяється сюжету, персонажам та ігровому процесу [3]. До одиночних ігор належать The Wither 3, Elden Ring, Red Dead Redemption 2 та Persona 5 Royal.

Проаналізувавши ігри, що були популярні останні декілька років можна побачити, що основна частина гравців надає перевагу проектам з можливістю багатокористувацької гри (рис. 1.1).



Рисунок 1.1 – Співвідношення одиночних та багатокористувацьких ігор

Для того, щоб гравці могли приєднуватися і взаємодіяти у ігровому світі, їм потрібно приєднатися до один одного. У сучасній сфері відеоігор використовуються два основних типів з'єднання: з використанням локальної мережі (LAN), та за допомогою мережі інтернет [4].

У локальних мережах взаємодія користувачів реалізується за допомогою локального (часто P2P – гравець до гравця) з'єднання у одній мережі. Це дозволяє обмінюватися даними без використання серверів, що знижує затримку та забезпечує більш швидкий відгук [5]. Недоліком такого з'єднання його обмеженість однією фізичною мережею. Якщо ж такої мережі немає, можна використати спеціальні емулятори віртуальної локальної мережі, що може вплинути на якість та стабільність з'єднання.

З'єднання за допомогою мережі інтернет дозволяє підключення з будь якої точки, незалежно від фізичного розташування користувача. Така свобода відкриває широкі можливості для організації взаємодії між віддаленими

користувачами. Більшість сучасних багатокористувацьких ігор використовує саме інтернет з’єднання для реалізації взаємодій між гравцями [6].

Окремо потрібно виділити асинхронне з’єднання. У ньому, як можна зрозуміти з назви, не передбачена взаємодія у реальному часі. Тобто, між діями користувачів є затримка, яка різниться у залежності від конкретної гри. Цей вид з’єднання вкрай рідко зустрічається у наш час, але раніше від був одним з основних способів мережевої взаємодії між гравцями [7].

Основною особливістю ігор з мережевим з’єднанням є обмін даними. Мережеві ігри передбачають, що усі гравці повинні бачити однакову картину світу. Тобто, обов’язковою є синхронізація станів гравців та ігрових об’єктів. Така синхронізація досягається за допомогою обміну даних між клієнтом та сервером. Найбільш популярної архітектурою мережевого з’єднання у комп’ютерних іграх є клієнт-серверна модель [8]. Вона передбачає під’єднання клієнтів до центрального сервера, який розподіляє інформацію між усіма користувачами (рис. 1.2).

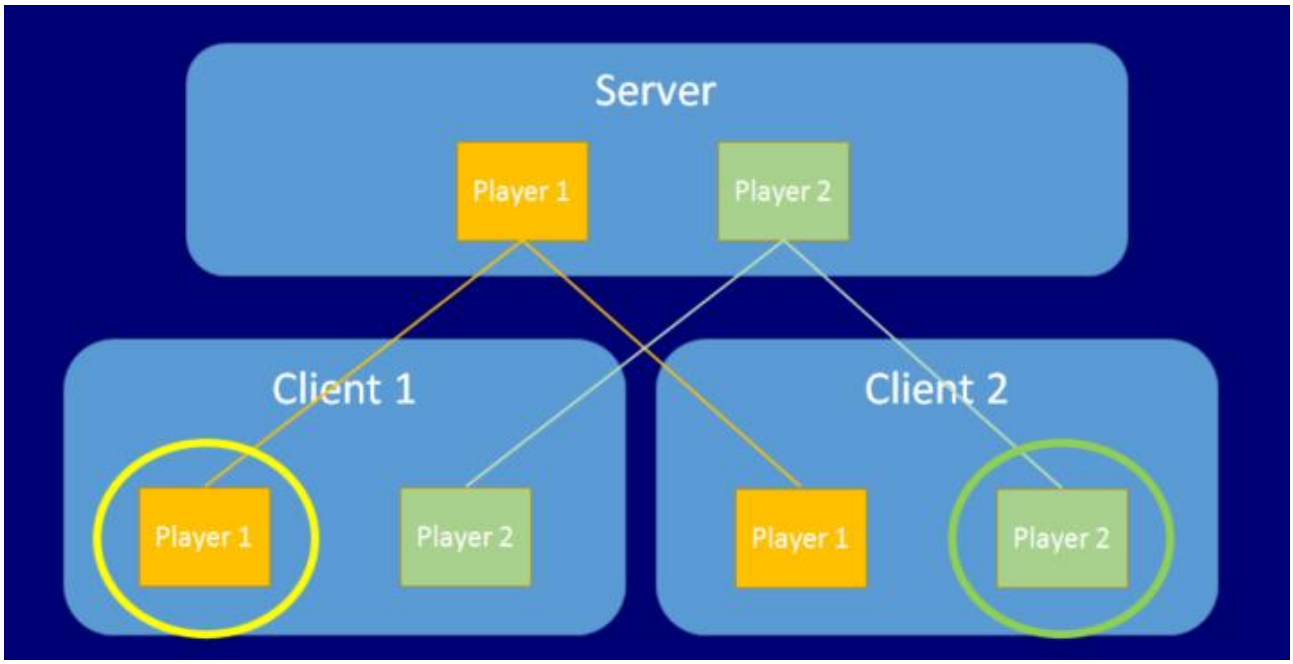


Рисунок 1.2 – Діаграма взаємодії двох клієнтів та сервера

В онлайн-іграх сервером зв’язку може виступати як виділена машина, так і сам користувач. У останньому випадку сервером стає перший клієнт.

Перевагами такої моделі є стабільність та можливість масштабування. Серед недоліків – залежність від якості з’єднання та, у випадку виділеного сервера, висока вартість утримання [9].

Прикладом популярної багатокористувацької гри, яка використовує клієнт-серверну архітектуру є «Overcooked! 2» (рис. 1.3).



Рисунок 1.3 – Кадр ігрового процесу «Overcooked! 2»

Overcooked! 2 – це кооперативна кулінарна аркада, де гравці керують кухарями. Метою є приготування та своєчасна доставка страв відповідно до замовлень. Гра вимагає тісної координації між гравцями, розподілу обов’язків та швидкої реакції, оскільки кожен рівень ускладнюється різними перешкодами та мінливими умовами.

Іншим прикладом буде схожа гра – «PlateUp!» (рис. 1.4). Ця кооперативна гра використовує жанр ресторанного бізнесу, де гравці працюють разом, щоб готувати страви, обслуговувати клієнтів та розвивати власний ресторан. Ігровий процес поєднує швидкість та стратегічне планування: гра потребує розподілення ресурсів та постійного вдосконалення кухні для успішного виконання постійно зростаючої кількості замовлень від потоку клієнтів.

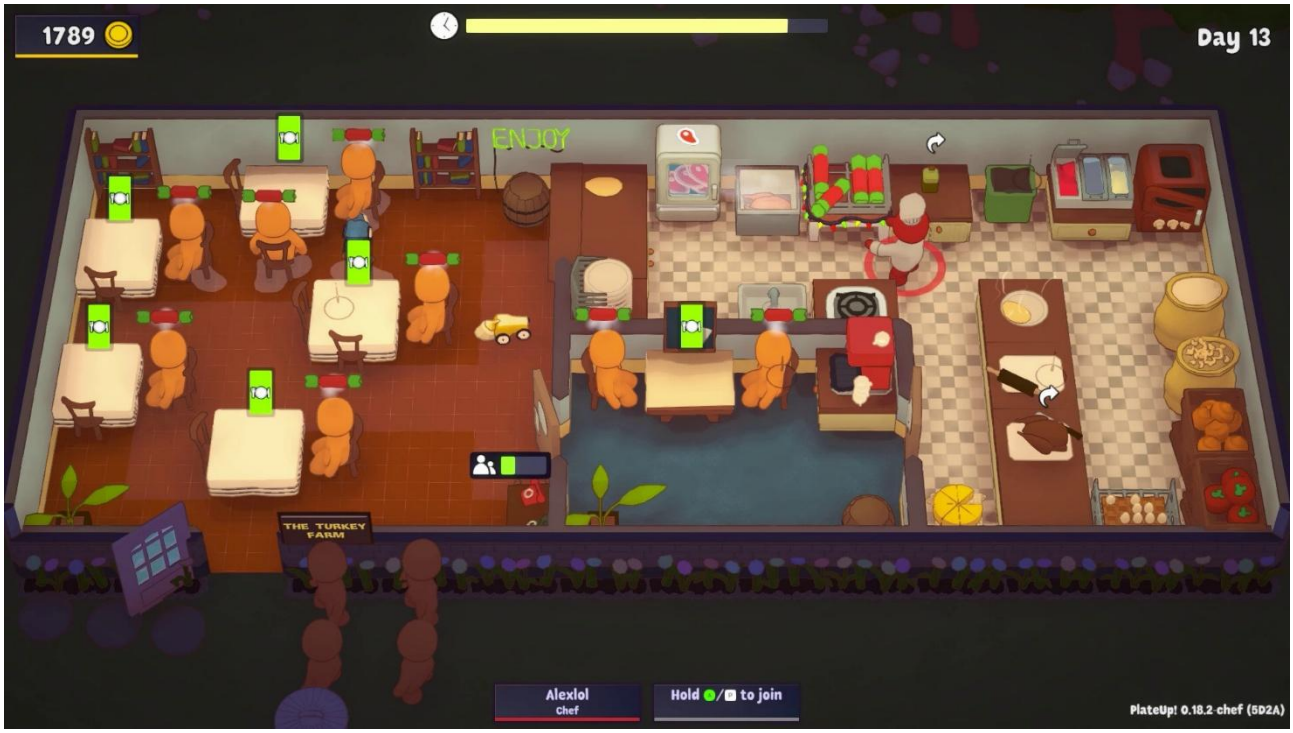


Рисунок 1.4 – Ігровий процес «PlateUp!»

Окрім ресторанної тематики, обидві ігри об’єднує потреба у командній роботі, простий візуальний стиль та випадкові події, які дозволяють уникнути монотонності у ігровому процесі.

1.2 Вибір технологій для створення відеогри

Раніше усі ігри були написані як єдина програма на одній із мов програмування. Іншими словами, усі об’єкти, гравці, їх взаємодія та навіть графіка знаходилися у одному програмному коді [10]. Очевидними недоліками такого підходу є важкість підтримки готового продукту та високий поріг входу в розробку, так як, наприклад, відображення графіки або реалізація фізики без допомоги спеціального інструментарію є доволі складною задачею. Спроби облегшити задачу розробника відеоігор привели до появи так званих ігрових рушіїв – програмного забезпечення, створеного спеціально для розробки ігор. Рушій включає в себе необхідні бібліотеки та інструменти, такі як графічний

редактор або навіть спеціальні засоби тренування та подальшого використання алгоритмів машинного навчання та штучного інтелекту.

Загальний функціонал, який надає ігровий рушій охоплює рендеринг зображення, виділений фізичний рушій, інструменти для роботи зі звуком, фреймворк для створення мережевого з'єднання [11].

Покладаючись на концепти об'єктно-орієнтованого програмування (ООП), ігровий рушій дозволяє с легкістю зосередитися на одній задачі без потреби рефакторингу загального коду або ігрової логіки у цілому.

На сьогоднішній день існує безліч ігрових рушіїв із різним функціоналом та своїми особливостями. Деякі компанії та команди розробників створюють свої власні рушії для конкретних потреб. Загалом ж, основними ігровими рушіями, якими користуються як інді – проекти без значної фінансової підтримки, так і великі корпорації це Unreal Engine, Unity, Godot та Source. Наведені рушії дозволяють безкоштовно створювати та видавати відеоігри за умови, якщо після подолання заданого порогу продажів розробник буде виплачувати роялті у виді процентів від загального прибутку. Така система ефективно створює відносини між користувачем та компанією, яка надає підтримку ігровому рушію та дозволяє кожному використовувати програмне забезпечення без необхідності сплачувати за ліцензію з самого початку.

Для виконання поставленої задачі був обраний рушій Unity (рис. 1.5). Серед його переваг:

- низький поріг входу;
- велика спільнота користувачів;
- постійна підтримка та виправлення недоліків у нових версіях рушія;
- сучасна та проста у вивченні мова програмування C#;
- можливість встановлення надбудов на рушій;
- маркетплейс, який дозволяє отримати технічні рішення та графічні ресурси.

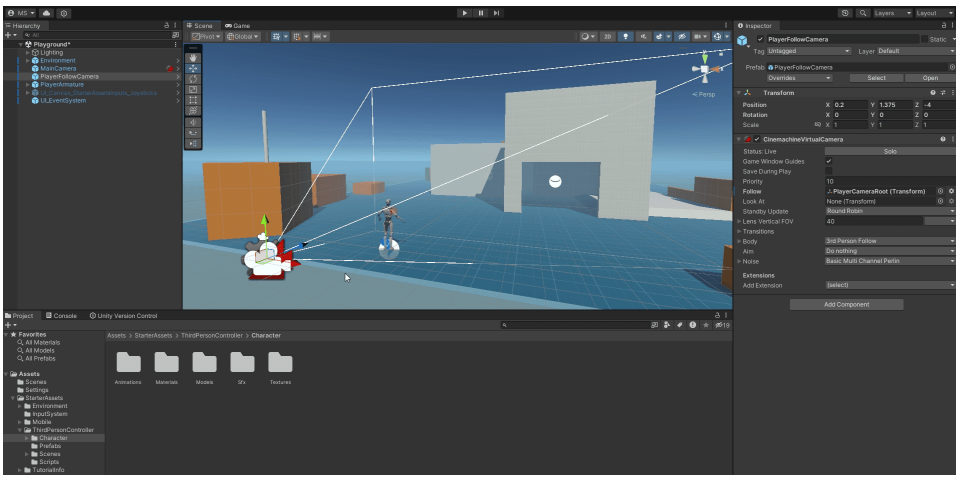


Рисунок 1.5 – Інтерфейс ігрового рушія Unity

Unity з самого початку створювався для широкої аудиторії, тому для розробки якісного продукту треба мінімум вузькопрофільних спеціалістів, що дозволяє отримати результат без залучення значних фінансів [12].

Для реалізації мережевого з’єднання була використана бібліотека Unity Netcode. Вона застосовується для створення кооперативних багатокористувацьких ігор. Netcode дозволяє синхронізувати ігрові сцени, об’єкти та інформацію для клієнтів, для відтворення потрібного стану гравця [13].

Для створення графічних ресурсів було використане програмне забезпечення для створення тривимірних об’єктів Blender (рис. 1.6).

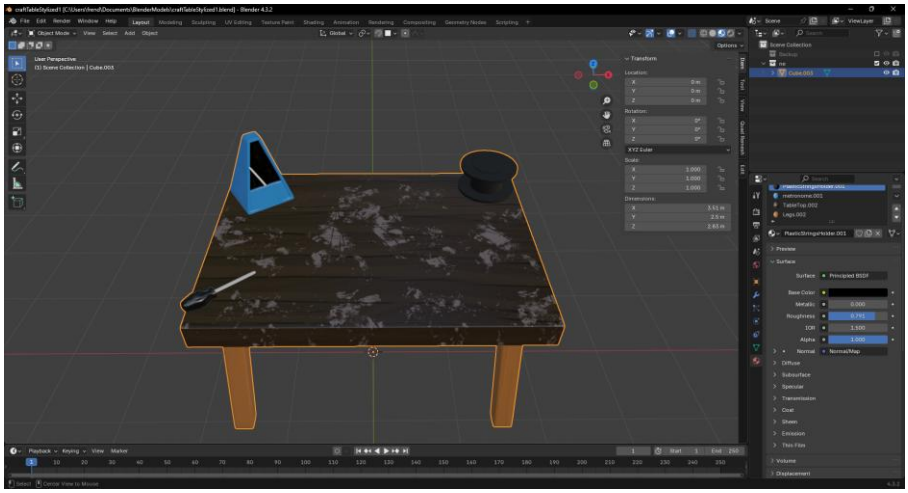


Рисунок 1.6 – Інтерфейс Blender

Він являє собою професійний графічний редактор з відкритим кодом та включає в себе потужний інструментарій для моделювання, скульптури, анімації, рендерингу та створення текстур. Blender підтримує процедурну генерацію геометрії та матеріалів і дозволяє використовувати складні математичні функції для отримання бажаного результату [14]. Серед очевидних переваг можна виділити постійну підтримку програмного забезпечення та безкоштовну модель поширення.

Для створення звукового супроводу, ефектів та музики були використані комерційні програмні забезпечення Adobe Audition та FL Studio. Аудіоредактор Audition від компанії Adobe – це професійне програмне забезпечення для роботи зі звуком. Воно дозволяє створювати, редагувати та міксувати аудіо контент. FL Studio являє собою DAW (digital audio workstation) – програмне забезпечення для запису, синтезу та редагуванню аудіо. FL Studio має широкий набір інструментів для роботи зі звуком та дозволяє виконувати дії будь-якого степеню складності із аудіофайлами (рис. 1.7).

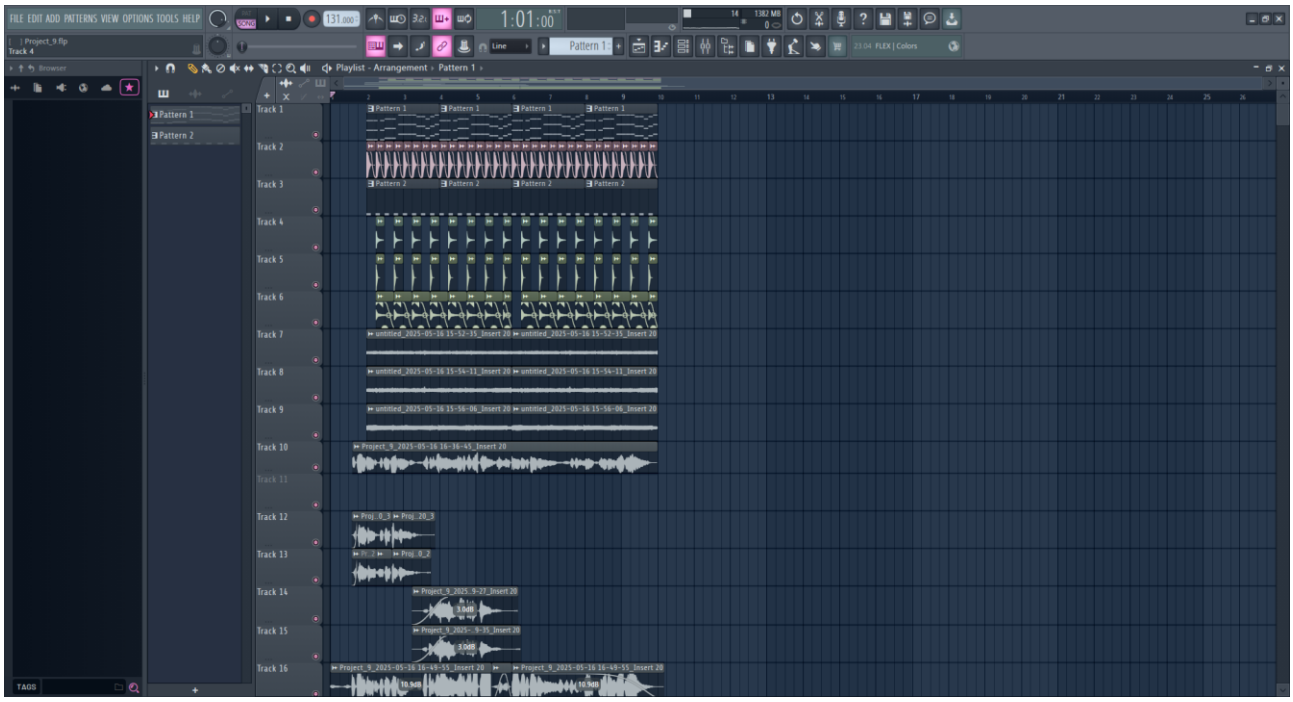


Рисунок 1.7 – Інтерфейс FL Studio

1.3 Постановка задачі

Метою даної роботи є створення мережевої відеогри використовуючи наведені у минулому підрозділі інструменти. Задумана комп’ютерна гра буде для двох гравців. Її концепція полягає у тому, що користувачі будуть грати за техніків на музичному концерті. Їх основна задача – вирішувати проблеми, які будуть виникати у ході виступу. У центрі буде стояти сцена з входом, який веде до складського приміщення, у якому знаходяться запасні ресурси та деталі музичних інструментів. На сцені будуть виступати NPC (неігрові персонажі) і протягом концерту у них будуть виходити зі строю їх інструменти. Техніки повинні оперативно забрати інструменти, виправити їх на складі та замінити на сцені. Якщо ж не виправляти проблеми, то з часом ігровий рахунок буде падати. Коли рахунок буде дорівнювати нулю – гра закінчиться.

Для досягнення поставленої мети передбачається виконання наступних завдань:

– змоделювати сцену, набір інструментів та апаратуру, створити анімації;
– реалізувати рух гравців та взаємодію з ігровими об’єктами;
– написати мережевий код та реалізувати з’єднання;
– провести синхронізацію між клієнтами;
– створити звуковий та музичний супровід.

1.4 Висновок за першим розділом

У першому розділі роботи було проведено аналіз предметної області – підтверджено попит користувачів на багатокористувацькі ігри, приведено різні способи з’єднання та огляд архітектури. У розділі наведено приклад схожих відеоігор, а також, описаний набір технологій, які будуть використовуватися у процесі розробки.

2 ПРОЄКТУВАННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Вимоги до системи

Перед початком роботи потрібно сформувати функціональні вимоги до проектованої інформаційної системи та визначити основні елементи, які будуть наявні у такій системі. Виділення таких елементів під час проєктування – один з найважливіших кроків, що дозволяє структурувати процес розробки та чітко окреслити майбутню функціональність.

Основними елементами системи є:

– гравець;
– NPC – неігрові персонажі, музиканти;
– об’єкти;
– сховища;
– спеціальний стіл для ремонту інструментів;
– графічний інтерфейс;
– ігровий об’єкт для виконання логіки.

Розглянемо елементи детальніше.

Гравець являє собою користувача комп’ютерної гри. Він використовує сигнал с пристрою введення для виконання дій у просторі. Він може: рухатися у тривимірному просторі, піднімати та переносити об’єкти, ремонтувати інструменти. Також до основних дій належить взаємодія з графічним інтерфейсом, меню. У меню підключення гравець має можливість змінити ім’я, яке буде відображатися у грі, налаштувати параметри мережевого з’єднання. Основні дії гравця зображені на діаграмі використання (рис. 2.1).

Діаграма використання – це вихідна концептуальна модель системи в процесі її проектування і розробки, на якій зображуються відношення між зовнішніми агентами і варіантами використання. Зовнішнім агентом виступає

користувач, тобто гравець. Варіанти використання – основні дії, що може виконувати гравець в межах відеогри.

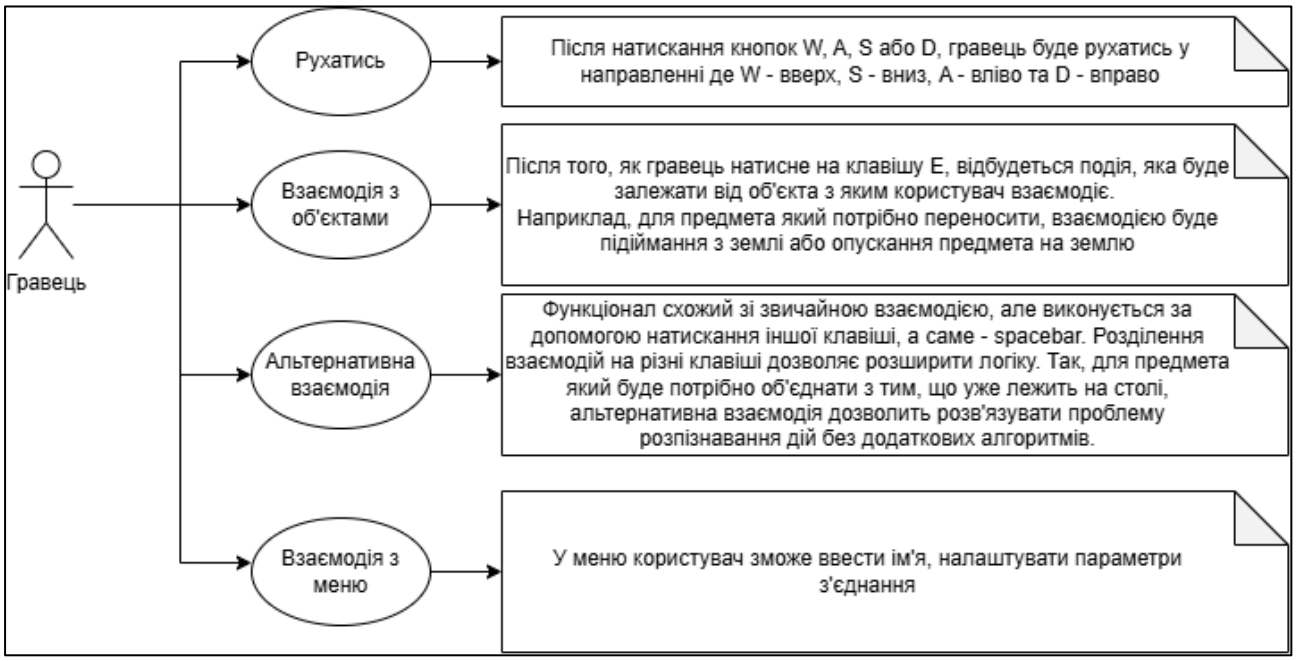


Рисунок 2.1 – Діаграма варіантів використання

NPC, або неігрові персонажі будуть виступати як музиканти у проектованій грі. На таких сутностях буде лежати відповідність за отримання інформації від об’єкта для обробки ігрової логіки, передача та прийняття інструментів від гравця, програвання анімацій у залежності від стану персонажа. Наприклад, якщо у персонажа є справний інструмент, то анімація буде відображати гру на ньому. Якщо ж інструмент буде у несправному стані, то анімація зміниться на іншу, при якій музикант не вже не буде грати на зламаному інструменті.

Об’єктами даної інформаційної системи будуть інструменти та їх складові частини, за допомогою яких ми будемо виправляти будь-які несправності самих музичних інструментів. Тобто, кожен об’єкт повинен мати свій тип, наприклад, струна для гітари. Такі об’єкти можуть бути переміщені, збережені та використані під час ігрового процесу.

Для зберігання та організації складових музичних інструментів будуть використані спеціальні сховища. При взаємодії з ними гравець зможе отримати відповідний ігровий об’єкт та використовувати його у подальшій грі.

Стіл для ремонту інструментів буде одним із найголовніших елементів проектованої системи. Його логіка полягає у тому, що такий стіл отримує інформацію від музичного інструмента про несправні частини та відображає їх за допомогою графічного інтерфейсу. Якщо гравець взаємодіє зі столом, на якому знаходиться несправний інструмент, а сам користувач має при собі ремонту частину, яка потребує заміни, то ця частина музичного інструмента буде вважатися виправленою.

Графічний інтерфейс гри – це усі текстові елементи, кнопки та поля для вводу, іконки та інші графічні складники. Інформація на таких об’єктах буде обновлюватися у реальному часі після того, як гравець здійснить певну дію.

Головним об’єктом у системі є сутність для обробки ігрової логіки. Хоча цей елемент не буде напряму відображатися у грі, він буде відповідати за майже усі ігрові події: поломка інструментів, таймер для визначення ігрового рахунку, тощо.

2.2 Діаграма класів

Діаграма класів – представлення структури у мові моделювання UML. Вона відображає такі елементи моделі, такі як: класи, типи даних, їх зміст та відношення. Класи представлені у вигляді прямокутників, які містять усю необхідну інформацію – назва класу, атрибути та методи [15].

На рисунку 2.2 зображено загальну діаграму класів проектованої системи для демонстрації логіки взаємодії. На цій діаграмі зображена схема взаємодії та наслідування класів, яка буде використовуватися під час реалізації логіки для руху гравця, отримання та передача рухомих об’єктів та ремонт і заміна зламаних музичних інструментів.

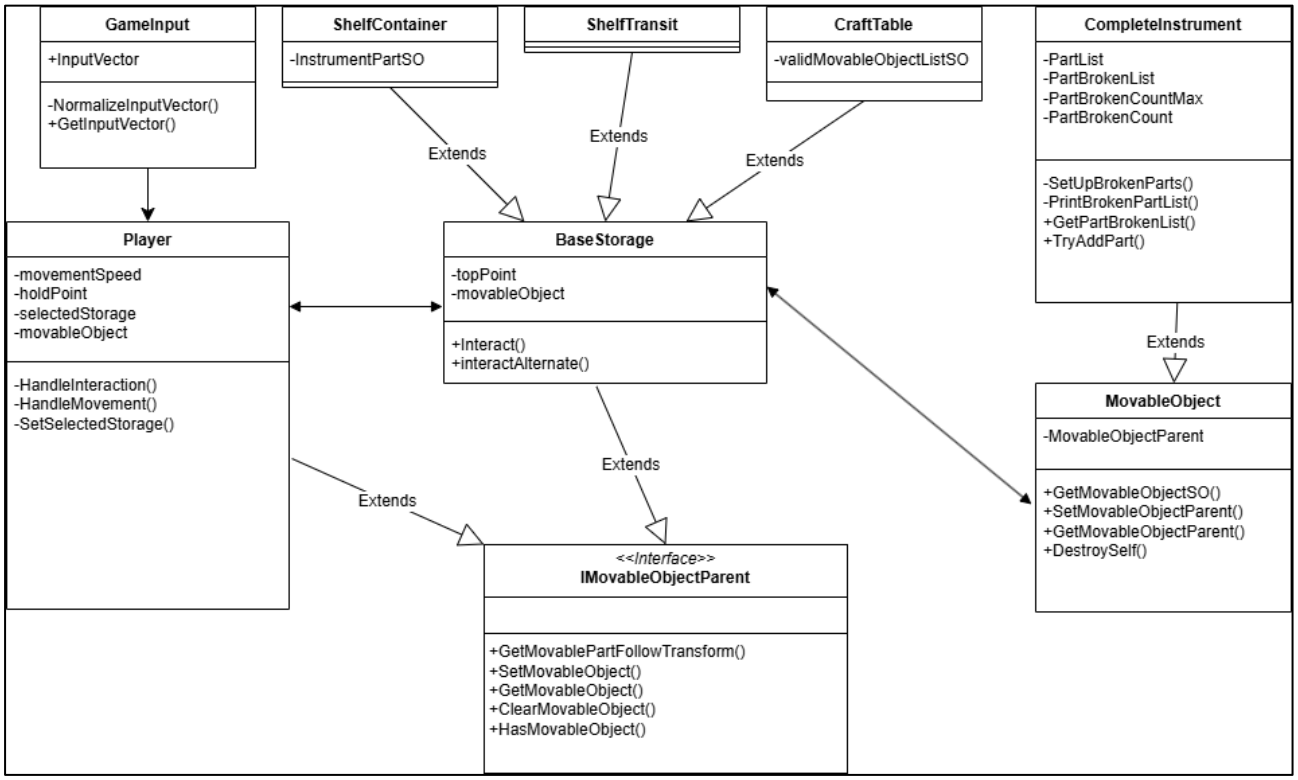


Рисунок 2.2 – Діаграма класів взаємодії з об’єктами

Для обробки вхідного сигналу з клавіатури у системі використовується клас GameInput. Всередині цього класу знаходиться вектор направлення та загальнодоступний метод, який повертає цей вектор.

Клас гравця – Player, він має посилання на клас GameInput та використовує вектор направлення для взаємодії та руху.

ShelfContainer та ShelftTransit – це класи сховищ, де перший відповідає за зберігання і видачу певних об’єктів, а другий дозволяє перекласти на себе уже існуючи об’єкти.

CraftTable відповідає за логіку виправлення та заміни несправних деталей музичного інструмента. Разом з двома сховищами, він наслідує базовий клас – BaseStorage. Цей клас містить усі необхідні функції для взаємодії гравця.

Рухомі об’єкти мають клас MovableObject. Цей клас дозволяє отримати посилання на сховище, поточного власника. Також має метод знищення – destroySelf().

Клас CompleteInstrument відповідає музичному інструменту. Всередині він містить усю необхідну інформацію: список усіх складових деталей, список зламаних частин та їх загальна кількість.

Для простої реалізації передачі об’єктів від гравця на різні столи або сховища, усі ці класи (Player, BaseStorage) наслідують інтерфейс IMovableObjectParent, який дозволяє встановлювати нового власника об’єкта без ускладнення логіки.

2.3 Мережева логіка

Використовуючи бібліотеку Unity NetCode for GameObjects ми зобов’язані створити клас NetworkManager, який буде відповідати за створення серверу, підключення клієнтів та динамічне генерування об’єктів, які ми бажаємо передавати по мережі, а також гравців. Окрім цього, для оптимізації роботи з мережевим кодом та подальшої синхронізації нам будуть потрібні додаткові об’єкти , такі як: управління ігровим рахунком, глобальний таймер та сутність для поломки музичних інструментів, тощо. На рисунку 2.3 зображена загальна схема мережевої частини інформаційної системи.

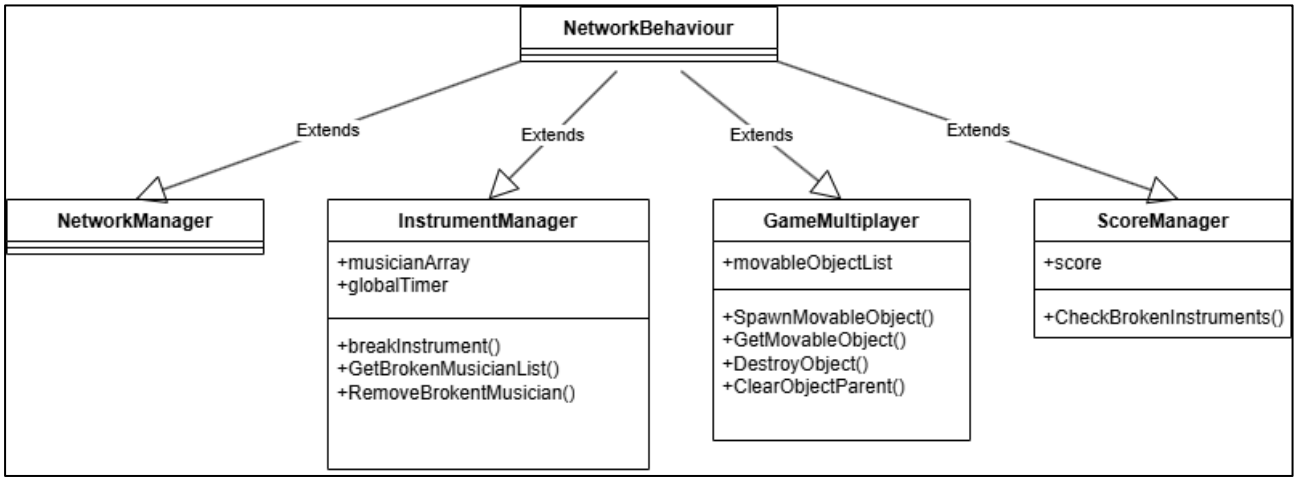


Рисунок 2.3 – Діаграма класів для мережевої взаємодії

Кожен клас на даній схемі повинен бути лише один. Це досягається за допомогою дизайн шаблону Singleton [16]. Він гарантує, що клас матиме тільки один екземпляр, і забезпечує глобальну точку доступу до нього. Розглянемо кожен з цих класів детальніше.

Як можна побачити, усі класи наслідують базовий клас – NetworkBehaviour. Цей клас поставляється разом із бібліотекою NetCode та сигналізує ігровому рушію, що усі об’єкти, які наслідують такий клас, мають передавати свою інформацію по мережі. У подальшому усім потрібним об’єктам також буде встановлено наслідування NetworkBehaviour.

InstrumentManager використовується для запам’ятовування усіх музикантів у ігровій сцені, а також для ведення глобального таймеру між поломками музичних інструментів. Він відповідає за логіку всіх поломок.

GameMultiplayer містить у собі усі об’єкти, які ми будемо генерувати або знищувати. Бібліотека NetCode вимагає щоб до створення або знищення предметів мав відношення лише сервер, тому ми використовуємо спеціальний клас для ідентифікації об’єктів у мережі.

Клас ScoreManager, як можна зрозуміти з назви, веде облік ігрового рахунку та усіх пов’язаних з ним змін. Після того як спеціальна глобальна змінна, яка відповідає за рахунок, дійде до 0, цей клас також буде відповідати за вивід нової сцени з повідомленням про закінчення гри.

2.4 Висновок за другим розділом

У другому розділі роботи були сформовані вимоги до інформаційної системи, наведені необхідні для роботи функції та спроектована архітектура взаємодії об’єктів у розроблюваній системі.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

Існує два підходи до розробки багатокористувацьких ігор. Перший – реалізація базового мережевого з’єднання з самого початку. Вся логіка гри одразу враховує мережеву архітектуру. Другий ж спосіб передбачає розробку логіки у режимі одного гравця, а потім на базі створеної системи додавати інших користувачів. Такий підхід дозволяє спростити та прискорити початковий темп розробки. Для реалізації поставлених у даній роботі задач був обраний другий спосіб розробки.

3.1 Створення проєкту

Для початку роботи над будь-яким програмним продуктом у ігровому рушії Unity потрібно створити проєкт (рис. 3.1).

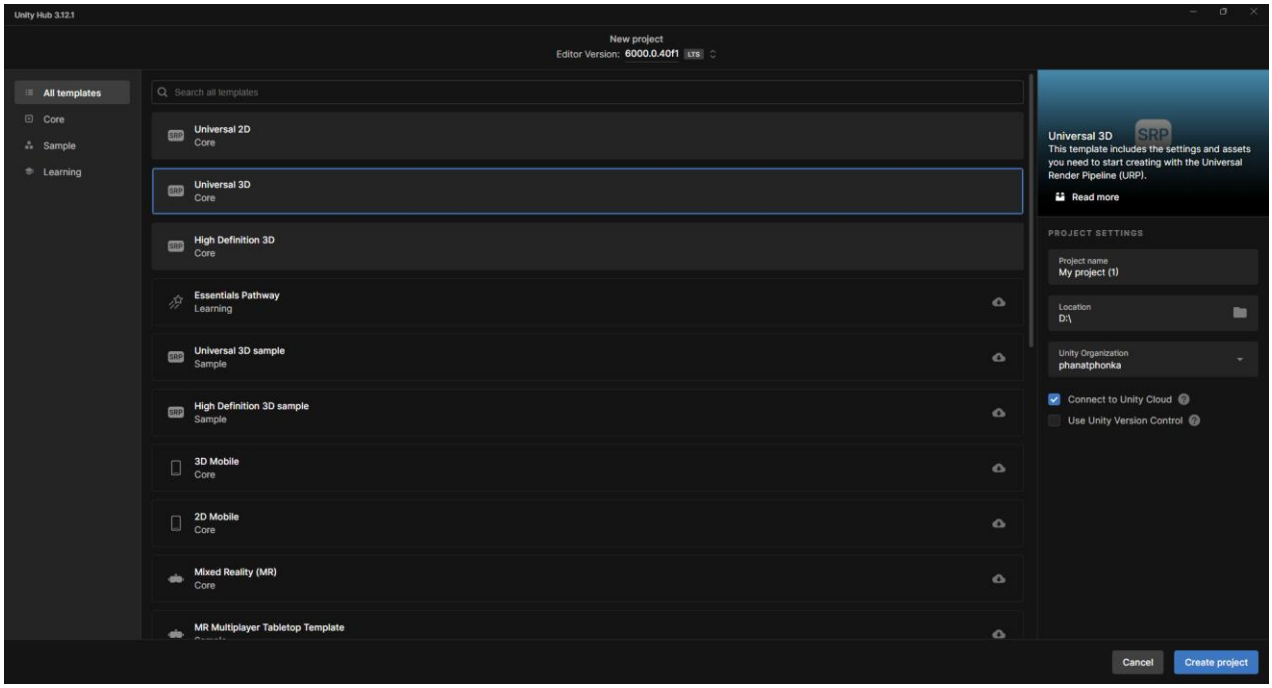


Рисунок 3.1 – Вікно створення нового проєкту

Рушій дозволяє обрати шаблон для майбутнього застосунку: 2D або 3D. Кожен з шаблонів включає у себе набір необхідних бібліотек та інструментів, необхідних для роботи. Також, Unity пропонує на вибір декілька рішень для рендеру об’єктів, серед яких є Built-in Render Pipeline, Universal Render Pipeline (URP) та High Definition Render Pipeline (HDRP). У межах цього проєкту було обрано рішення URP, розроблене командою Unity для забезпечення балансу між продуктивністю та візуальною якістю. Universal Render Pipeline підтримує багатоплатформність, що забезпечує ідентичну картинку на будь-якому пристрої та дозволяє створити спеціальні графічні профілі для кожної системи.

Після створення проєкту відкривається сам проєкт та його головна сцена (рис. 3.2).

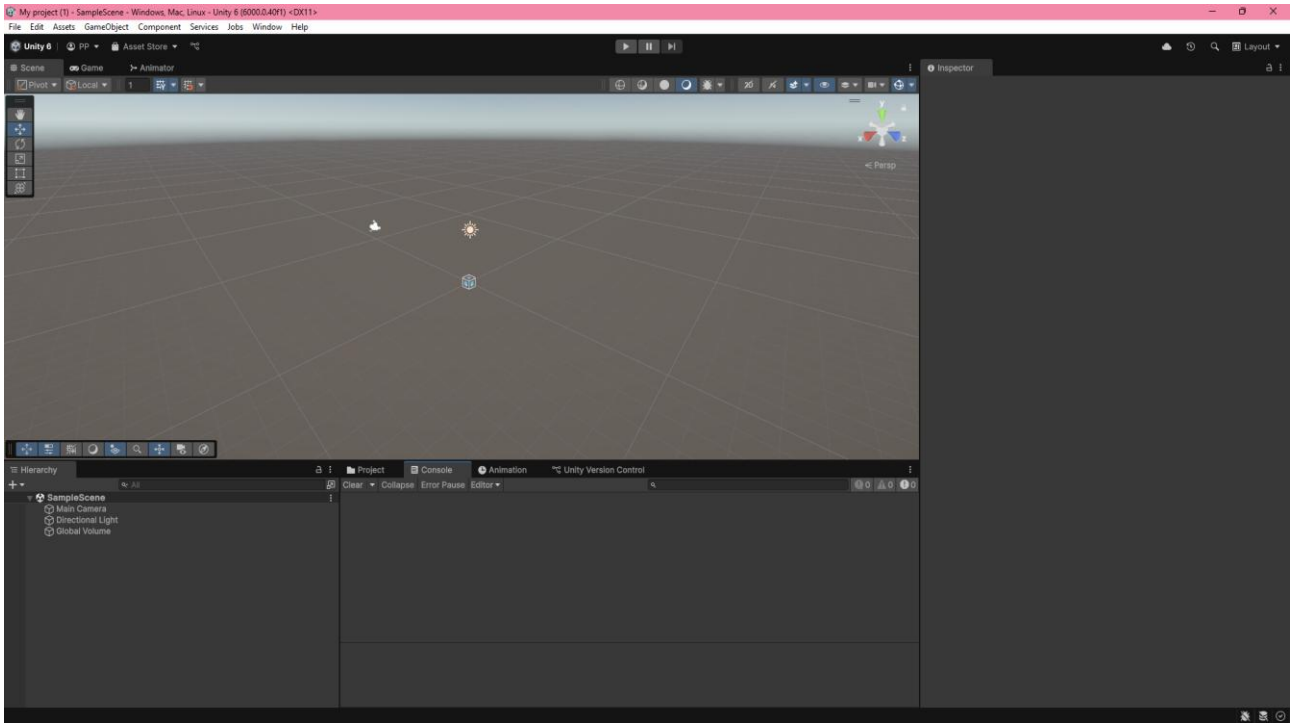


Рисунок 3.2 – Головне вікно рушія

Ця сцена включає в себе основні елементи для рендерингу – камера, джерело світла, а також об’єкт для постобробки (Global Volume), який відповідає за візуальні ефекти сцени. Камера відповідає за зір майбутнього гравця, джерело світла забезпечує базове освітлення сцени, а постобробка дозволяє застосовувати додаткові ефекти, такі як зміна кольорів, розмивання країв, та інші.

3.2 Рух гравця

Для реалізації прототипу було створено декілька 3D-об’єктів: капсула, яка буде відповідати за гравця, та площина, по якій гравець буде ходити (рис. 3.3).

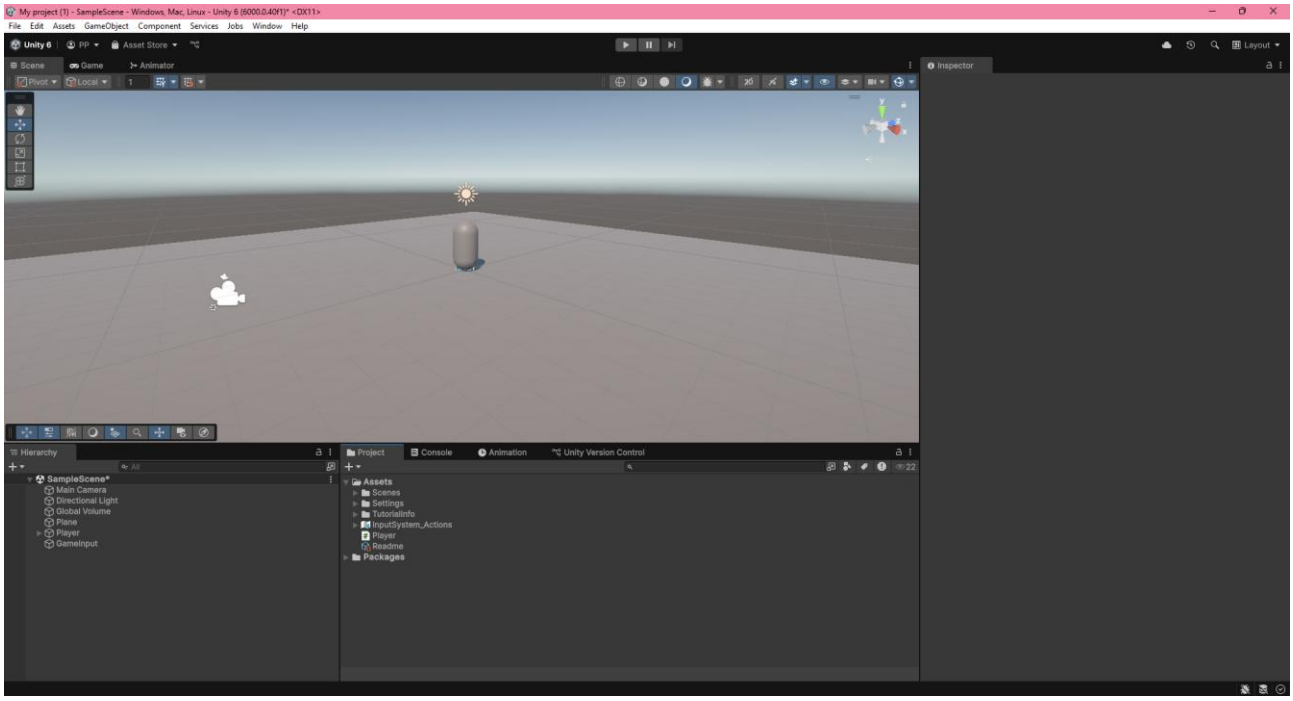


Рисунок 3.3 – Прототип гравця та землі

Сучасні версії ігрового рушія Unity дозволяють реалізувати управління за допомогою спеціального інструмента – Input System Package. Ця надбудова спрощує розробку різноманітних взаємодій через використання інтерфейсу та підтримки усіх популярних типів вводу, такий як клавіатура, мишка та геймпад. Input System Package дозволяє створювати карти дій для руху, взаємодій, та інших. Для кожної дії можливе налаштування типів управління та значень.

Таким чином, було створено карту дій для руху гравця, яка включає чотири основні напрямки: вперед, назад, вліво та вправо (рис. 3.4). Так як розроблювана гра не передбачає стрибки чи інше вертикальне переміщення, найбільш логічним є зчитування вектора напрямку руху у двомірному просторі, де вісь X відповідає за рух вліво та право, а Y – вперед або назад. Після натискання обраної клавіші ми зможемо зчитати значення вектору і, обробивши сигнал, здійснити рух гравця.

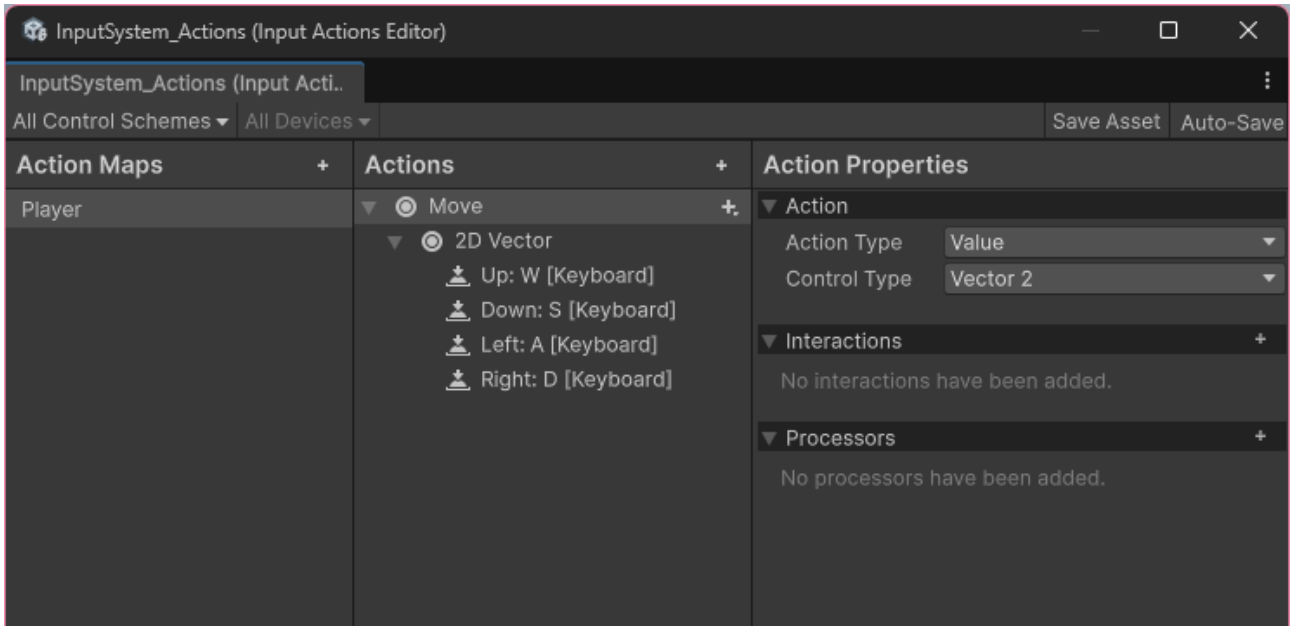


Рисунок 3.4 – Створена карта дій для руху гравця

Для використання створеної карти дій потрібно згенерувати клас. Це робиться простим натисканням кнопки з відповідною назвою у параметрах Input System Package.

Для організації та структурування був створений об’єкт GameInput та клас з такою ж назвою. У ньому ми будемо отримувати посилання на згенерований клас та повертати потрібні нам значення (рис. 3.5).

```
1   using UnityEngine;
2
    Скрипт Unity (1 ссылка на ресурсы) | Ссылок: 2
3   public class GameInput : MonoBehaviour
4   {
5       private InputSystem_Actions inputSystemActions;
6
        Ссылок: 3
7       public static GameInput Instance { get; private set; }
8
        Сообщение Unity | Ссылок: 0
9       private void Awake()
10      {
11          if (Instance != null)
12          {
13              Debug.LogError("More than one GameInput in the scene!");
14              return;
15          }
16          Instance = this;
17          inputSystemActions = new InputSystem_Actions();
18          inputSystemActions.Enable();
19      }
20
        Ссылок: 1
21      public Vector2 GetMovementNormalized()
22      {
23          Vector2 inputVector = inputSystemActions.Player.Move.ReadValue<Vector2>();
24          return inputVector.normalized;
25      }
26
27  }
```

Рисунок 3.5 – Клас GameInput

Найбільш логічним місцем для обробки сигналу та виконання необхідних обчислень перед рухом є клас Player (рис. 3.6).

```
private void Update()
{
    HandleMovement();
}

Ссылок: 1
private void HandleMovement()
{
    Vector2 inputVector = GameInput.Instance.GetMovementNormalized();
    Vector3 moveDir = new Vector3(inputVector.x, 0, inputVector.y);
    float moveDistance = movementSpeed * Time.deltaTime;
    float playerRadius = 0.7f;
    float playerHeight = 1.5f;

    bool canMove = !Physics.CapsuleCast(transform.position + Vector3.up * playerHeight / 3, transform.position + Vector3.up * playerHeight, playerRadius, moveDir, moveDistance);


    if (!canMove)
    {
        Vector3 moveDirX = new Vector3(moveDir.x, 0, 0);
        canMove = moveDir.x != 0 && !Physics.CapsuleCast(transform.position + Vector3.up * playerHeight / 3, transform.position + Vector3.up * playerHeight, playerRadius, moveDirX, moveDistance);
        if (canMove)
        {
            moveDir = moveDirX;
        }
        else
        {
            Vector3 moveDirZ = new Vector3(0, 0, moveDir.z);
            canMove = moveDir.z != 0 && !Physics.CapsuleCast(transform.position + Vector3.up * playerHeight / 3, transform.position + Vector3.up * playerHeight, playerRadius, moveDirZ, moveDistance);
            if (canMove)
            {
                moveDir = moveDirZ;
            }
        }
```

Рисунок 3.6 – Обробка вхідного сигналу класом Player

У наведеному вище коді наведена реалізація обробки руху гравця на основі вихідного сигнали. Після отримання вектору вводу ми перетворюємо 2D-вектор у 3D-простір, де вісь Y відповідає за вертикаль і лишається нульовою. Відстань руху залежить від швидкості персонажа та часу, що минув між кадрами, що забезпечує плавний і незалежний від кількості кадрів рух (див. рис. 3.7–3.8). Також, у наведеному коді передбачена перевірка на перешкоди – при наявності стіни або іншого об’єкта перед гравцем рух забороняється.

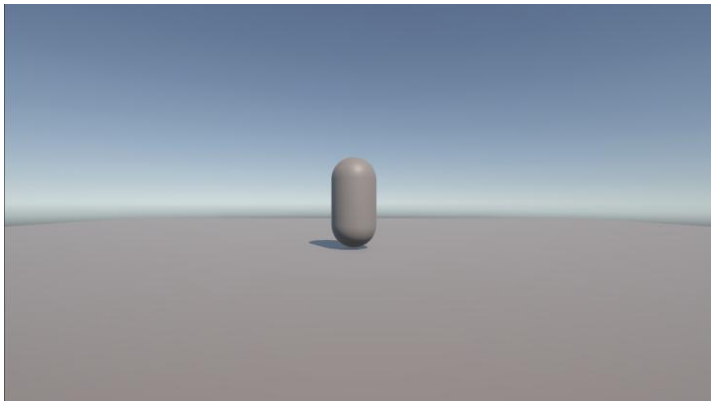


Рисунок 3.7 – Розташування гравця на початку гри

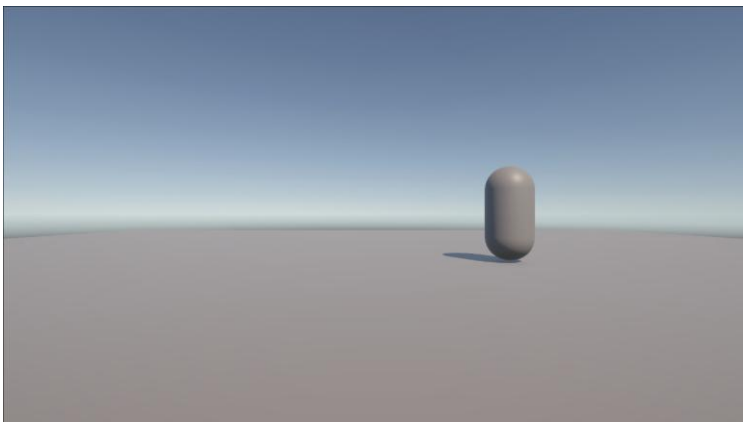


Рисунок 3.8 – Розташування гравця після виконання руху вправо

3.3 Поворот гравця

Для реалізації повороту у напрямку руху використовується функція Vector3.Slerp, яка дозволяє плавно змінювати напрямок погляду від поточного значення до нового напрямку руху (рис. 3.9).

```
float rotateSpeed = 10f;
transform.forward = Vector3.Slerp(transform.forward, moveDir, rotateSpeed * Time.deltaTime);
```

Рисунок 3.9 – Код реалізації повороту гравця

3.4 Взаємодія гравця

Створимо карти дій для передбачених на етапі проектування взаємодій (рис. 3.10). Вони включають в себе основну та альтернативну взаємодію. Для того, щоб виконати основну – потрібно натиснути клавішу E. Альтернативна ж взаємодія викликається натисканням клавіші Space. Типом управління у такому разі буде кнопка. Замість того, щоб постійно перевіряти, чи натиснута обрана клавіша, ми використаємо C# подію [17]. Вона буде викликати підписаний на неї метод кожен раз, як гравець натискає на клавішу взаємодії. Це дозволить зменшити кількість займаної пам’яті для ігрової логіки.

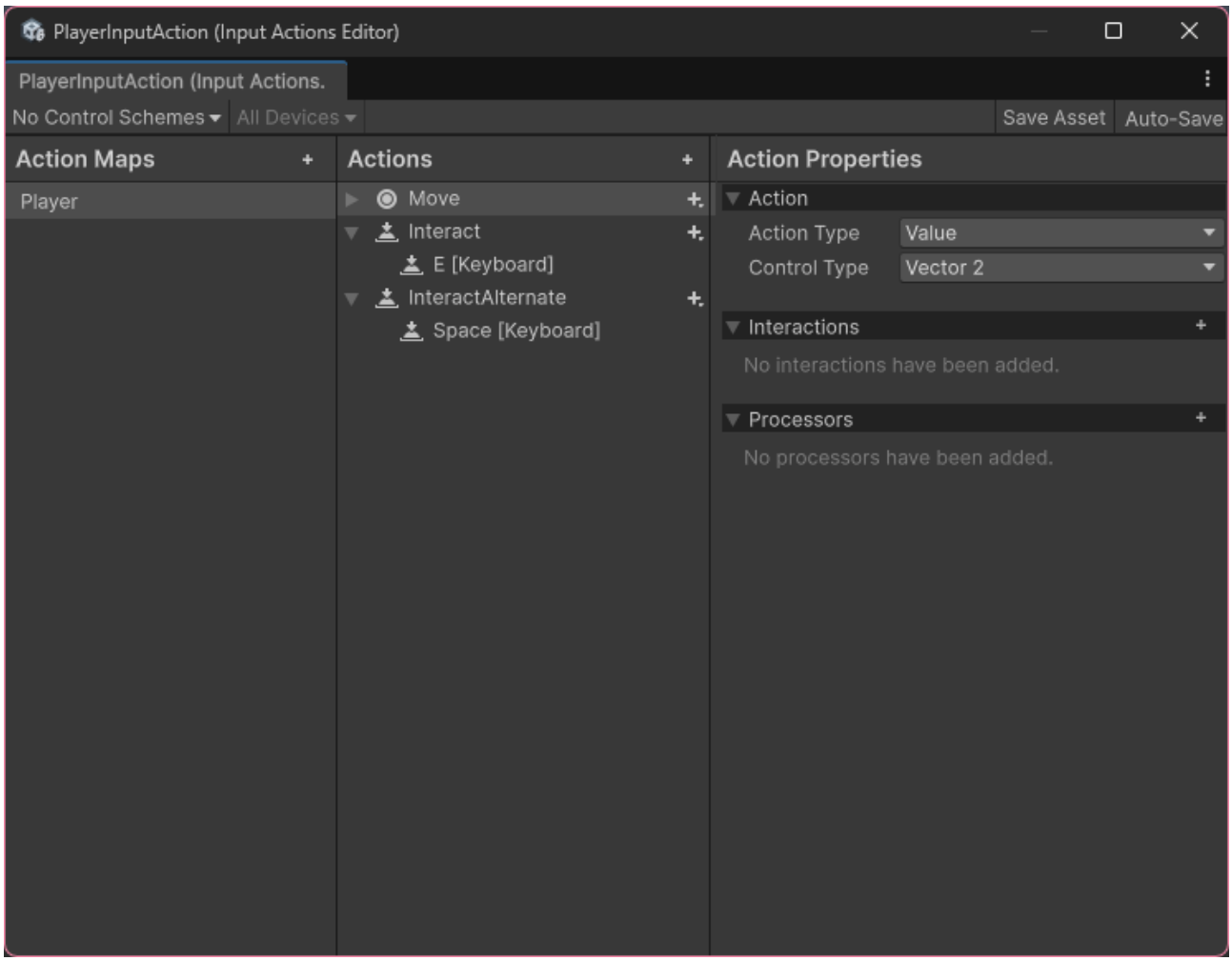


Рисунок 3.10 – Створені карти для основної та альтернативно взаємодії

Після успішного тестування прототипу змінимо капсулу на модель гравця (рис. 3.11).

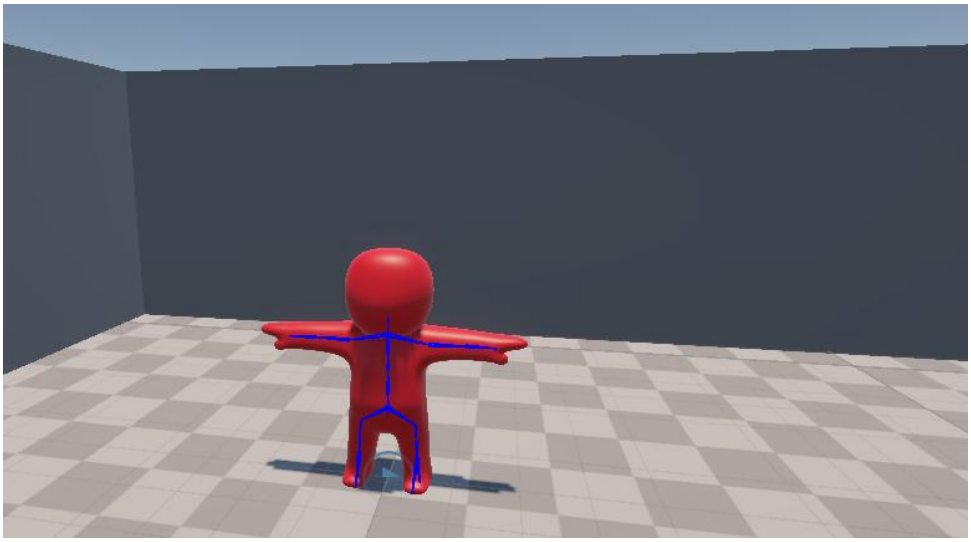


Рисунок 3.11 – Фінальна модель гравця

Гравцю потрібно знати, з яким предметом він взаємодіє. Для цього клас Player використовує промінь у направлені погляду гравця. Коли такий промінь зустрічається з об’єктом, котрому був присвоєний тег Storage, Player запам’ятовує посилання на цей об’єкт (рис. 3.12).

```
if (Physics.Raycast(transform.position, lastInteractDir, out RaycastHit raycastHit, interactDistance, layerMask))
{
    if(raycastHit.transform.TryGetComponent(out BaseStorage storage))
    {
        if(storage != selectedStorage)
        {
            SetSelectedStorage(storage);
        }
    }
    else
    {
        SetSelectedStorage(null);
    }
```

Рисунок 3.12 – Фрагмент коду з реалізацією вибору предмета

Спеціальний скрипт SelectedShelfVisual отримує посилання від гравця та візуально підсвічує обраний об’єкт (див. рис. 3.13–3.14).

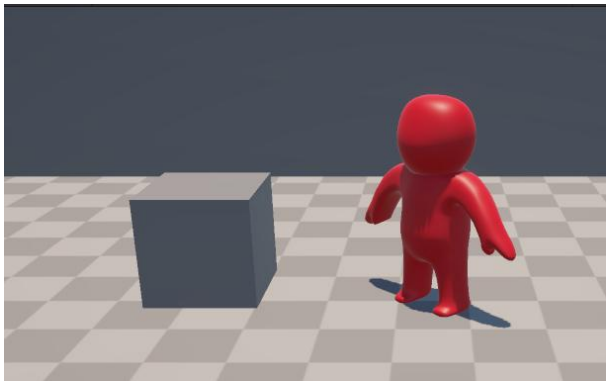


Рисунок 3.13 – Стан об’єкта поза радіусом взаємодії

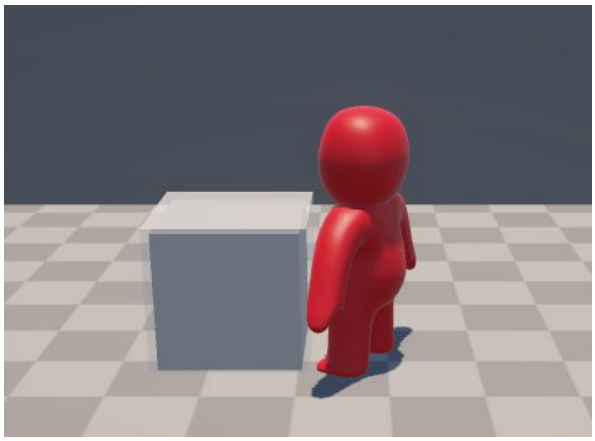


Рисунок 3.14 – Стан об’єкта, який знаходиться у зоні взаємодії

3.5 Предмети

У графічному редакторі Blender були змодельовані інструменти та їх складові (рис. 3.15). Ці інструменти стануть доступними до взаємодії об’єктами, де музичні інструменти – предмети, які будуть виходити зі строю, а їх частини – складовими, за допомогою яких поломки будуть усуватися.



Рисунок 3.15 – Моделі інструментів та їх частин

Окрему увагу потрібно звернути на спосіб організації даних у Unity. Ми не будемо реалізувати логіку поверх візуальних моделей даних об’єктів. Натомість ж, ми утворимо спеціальний контейнер для даних, який з’єднає логіку з візуальною частиною. Такий контейнер має назву ScriptableObject (рис. 3.16).

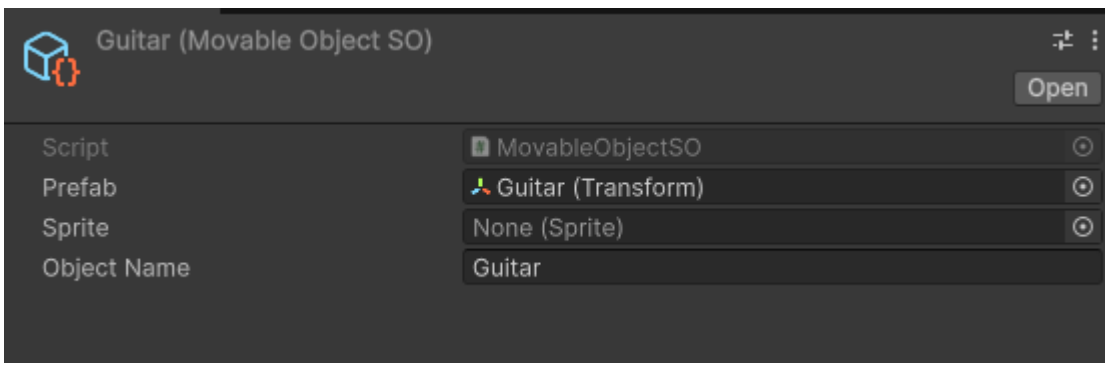


Рисунок 3.16 – ScriptableObject для гітари

Як можна побачити з наведеного вище рисунку, такий контейнер дозволяє об’єднати декілька різних типів даних та полегшує доступ до них.

Для реалізації логіки об’єктів був створений клас MovableObject. Він містить у собі посилання на свій ScriptableObject та батьківський об’єкт, до якого він прив’язаний. Основними методами даного класу є зміна батьківського об’єкта, та функції, які повертають посилання на поля класу MovableObject, такі як батьківський об’єкт та ScriptableObject. Також наявний метод DestroySelf, який видаляє поточний рухомий предмет. Усі ці методи знадобляться під час реалізації ігрової логіки взаємодії об’єктів.

Оскільки ігрова логіка для музичних інструментів відрізняється від логіки для складових частин, був створений окремий клас CompleteInstrument, який не тільки наслідує основні методи MovableObject, а й та реалізовує свої.

Основними задачами класу CompleteInstrument є зберігання частин, з яких він складається та методи для генерування поломок, списку потрібних складових для ремонту. Одним із найголовніших методів є TryAddCompleteInstrumentPart, який відповідає за виправлення поломок у випадку, якщо гравець задовольняє умові, тобто має при собі потрібний для ремонту предмет.

Для кожного інструмента та його частин був створений ігровий об’єкт з відповідним ім’ям (рис. 3.17).



Рисунок 3.17 – Ігрові об’єкти для кожного інструмента

Така система дозволяє налаштувати окремі посилання для кожного з об’єктів (рис. 3.18).

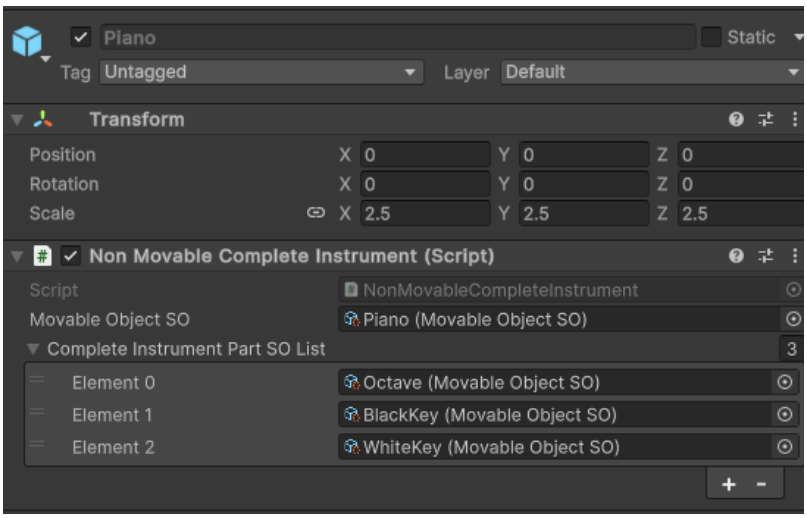


Рисунок 3.18 – Посилання на складові частини для піаніно

3.6 Об’єкти для зберігання предметів

За організацію усіх об’єктів, які будуть відповідати за зберігання, видачу та приймання предметів відповідає базовий клас – BaseStorage. Цей клас реалізує C#-інтерфейс IMovableObjectParent, який забезпечує універсальний підхід до взаємодії з ігровими об'єктами, що можуть бути контейнерами [18]. Основними можливостями класу BaseStorage є функції взаємодії з гравцем, встановлення дочірнього предмета, очищення предмета та перевірка і повернення поточного рухомого об’єкта, який зберігається у сховищі (рис. 3.19).

```
public class BaseStorage : MonoBehaviour , IMovableObjectParent
{
    [SerializeField] private Transform topPoint;

    private MovableObject movableObject;

    Ссылок 5
    public virtual void Interact(Player player)
    {
        Debug.Log("Interacting with : " + gameObject.name);
    }

    Ссылок 3
    public virtual void InteractAlternate(Player player)
    {
        Debug.Log("Interacting alternate with : " + gameObject.name);
    }

    Ссылок 2
    public Transform GetMovablePartFollowTransform()
    {
        return topPoint;
    }

    Ссылок 2
    public void SetMovableObject(MovableObject movableObject)
    {
        this.movableObject = movableObject;
    }

    Ссылок 7
    public MovableObject GetMovableObject()
    {
        return movableObject;
    }

    Ссылок 3
    public void ClearMovableObject()
    {
        movableObject = null;
    }
```

Рисунок 3.19 – Клас BaseStorage

Така система реалізує принципи спадкування та поліморфізму і дозволяє розширювати систему сховищ, створюючи нові дочірні класи, які наслідують BaseStorage і перевизначають потрібну логіку взаємодії [19].

3.6.1 Сховища

Для того, щоб зберігати запасні частини інструментів, потрібно створити спеціальні сховища (рис. 3.20). Їх основною задачею - видача передбачених частин музичних інструментів, які графічно зображені на моделі сховища.

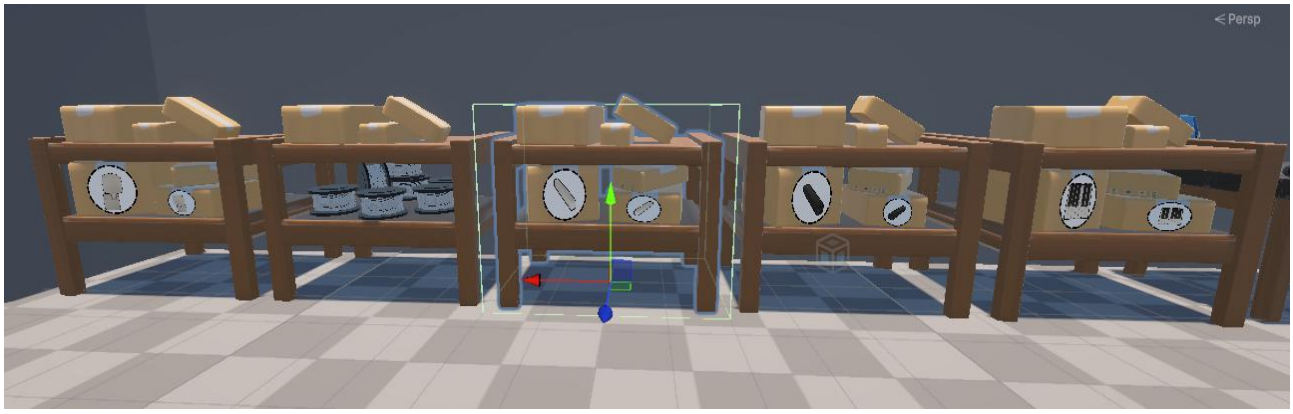


Рисунок 3.20 – Сховища для запчастин

Для контейнерів створено клас ShelfContainer. Він наслідує BaseStorage та містить посилання на об’єкт, який буде видавати. Метод Interact переписує зазначену в основному класі взаємодію (рис. 3.21).

```
public class ShelfContainer : BaseStorage
{
    [SerializeField] private MovableObjectSO instrumentPartSO;
    Ссылок: 2
    public override void Interact(Player player)
    {
        if (!player.HasMovableObject())
        {
            Transform instrumentPartTransform = Instantiate(instrumentPartSO.prefab);
            instrumentPartTransform.GetComponent<MovableObject>().SetMovableObjectParent(player);
        }
        else
        {
            Debug.Log("Player is carrying something");
        }

    }
}
```

Рисунок 3.21 – Клас ShelfContainer

Наведений фрагмент коду перевіряє, чи має гравець у руках деякий об’єкт, і, якщо такого об’єкта немає – видає йому предмет, який зберігається у даному контейнері (рис. 3.22).

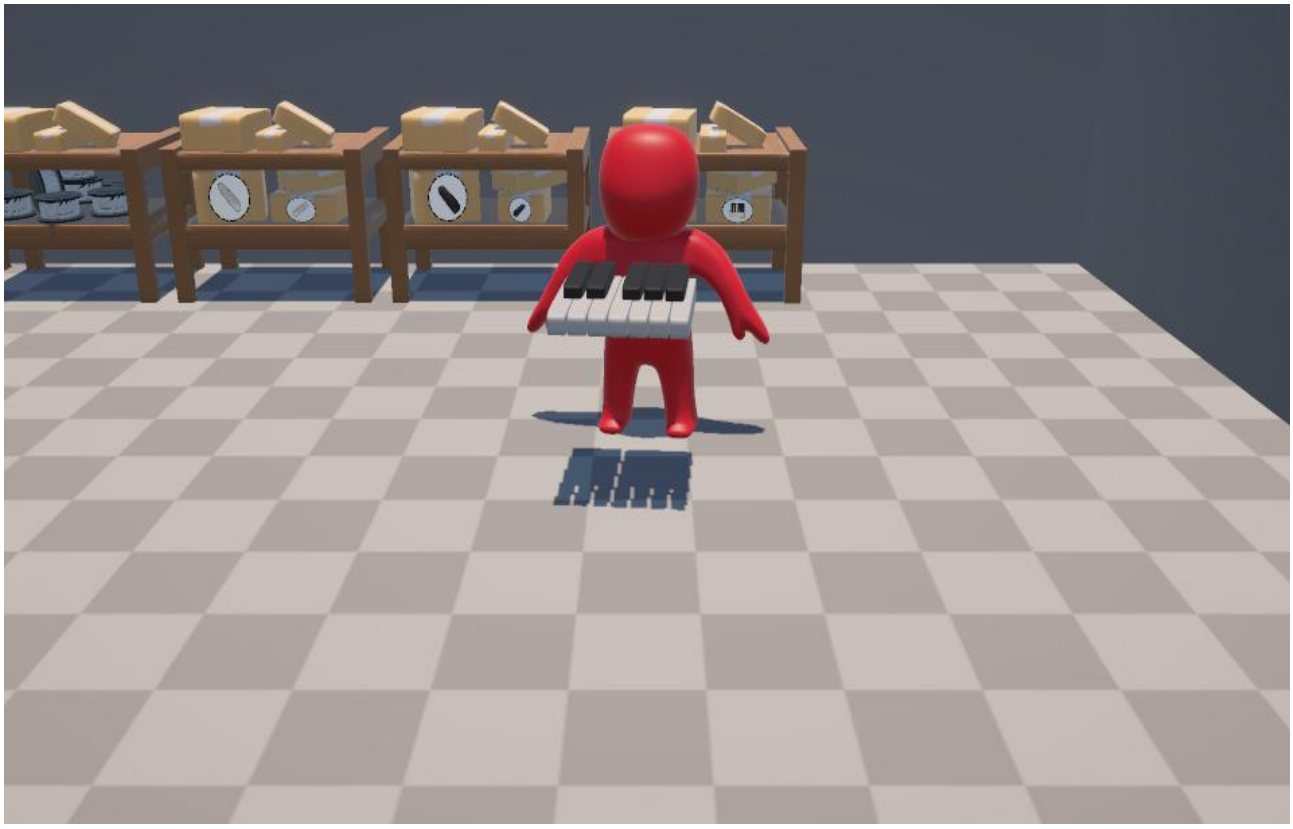


Рисунок 3.22 – Гравець отримав предмет зі сховища

3.6.2 Транзитні столи

Рухомі предмети можна не лише отримати, але й зберегти. Для цього було створений спеціальний стіл та клас ShelfTransit, який наслідує BaseStorage. Цей клас перевіряє наявність об’єктів у гравця та у самого себе. Якщо всі умови задоволені, гравець кладе предмет на стіл (рис. 3.23). Якщо ж предмет буде на столі, а гравець нічого не несе, він забирає предмет зі стола. Взаємодія реалізується в методі Interact, який визначає логіку передачі предмета залежно від того, хто з учасників має об'єкт. Така система зберігання та передачі предметів є інтуїтивною і не потребує додаткових дій від користувача.

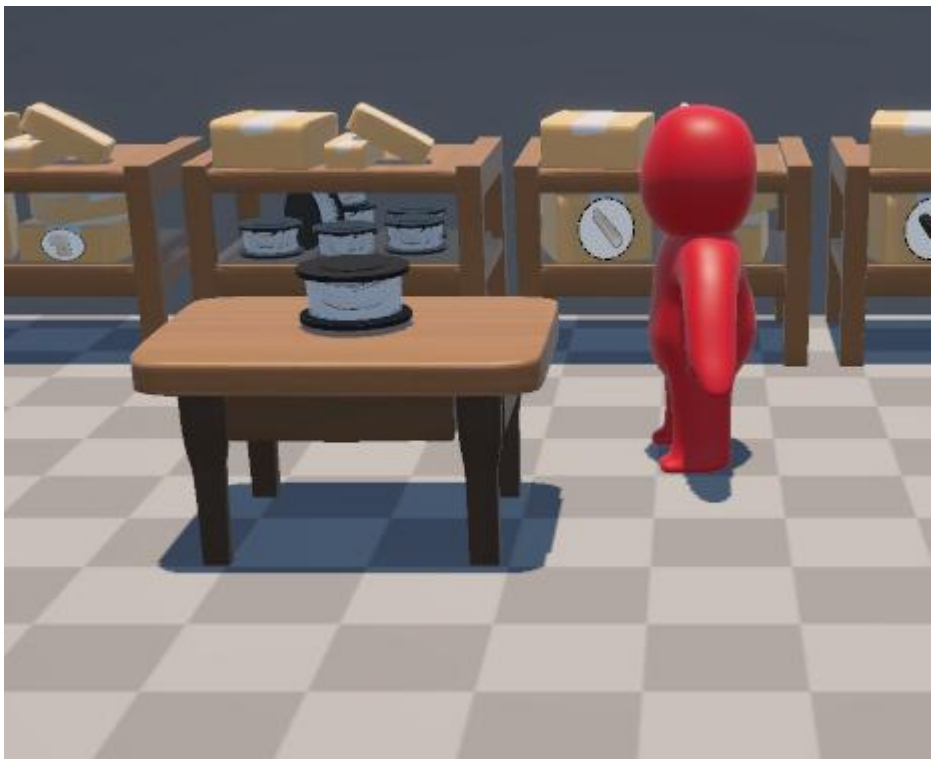


Рисунок 3.23 – Гравець поклав предмет на стіл

3.7 Музиканти

Для кожного з музичних інструментів було створено модель музиканта, який буде на ньому грати (рис. 3.24).



Рисунок 3.24 – Створені музиканти зі своїми інструментами

Кожному з музикантів було присвоєно новий клас – MusicianScript. Через те, що цей клас зберігає музичний інструмент, він також наслідує базовий клас BaseStorage. MusicianScript відповідає за взаємодію з музичним інструментом музиканта, а також обробку його поломки та ремонту. Коли відповідний інструмент зламаний, гравець може взаємодіяти з музикантом для початку ремонту. Взаємодія залежить від типу зламаного інструмента. Якщо це, наприклад, гітара – гравець отримає сам інструмент, який потребує ремонту. Важкі ж інструменти, такі як барабани, залишаються на місці, і гравець бачить список необхідних деталей, які необхідно принести.

У разі, коли гравець повертає музиканту повністю справний інструмент, MusicianScript перевіряє його відповідність та виконує заміну зламаного інструмента на новий. Для стаціонарних інструментів реалізовано альтернативна взаємодія, яка дозволяє поетапно відновлювати інструмент, додаючи окремі частини. Після завершення ремонту генерується подія, яка повідомляє про успішне відновлення інструмента.

Для візуальної ясності до кожного з музикантів була додана анімація гри, яка замінюється на іншу у разі, якщо інструмент вийшов зі строю (рис. 3.25)

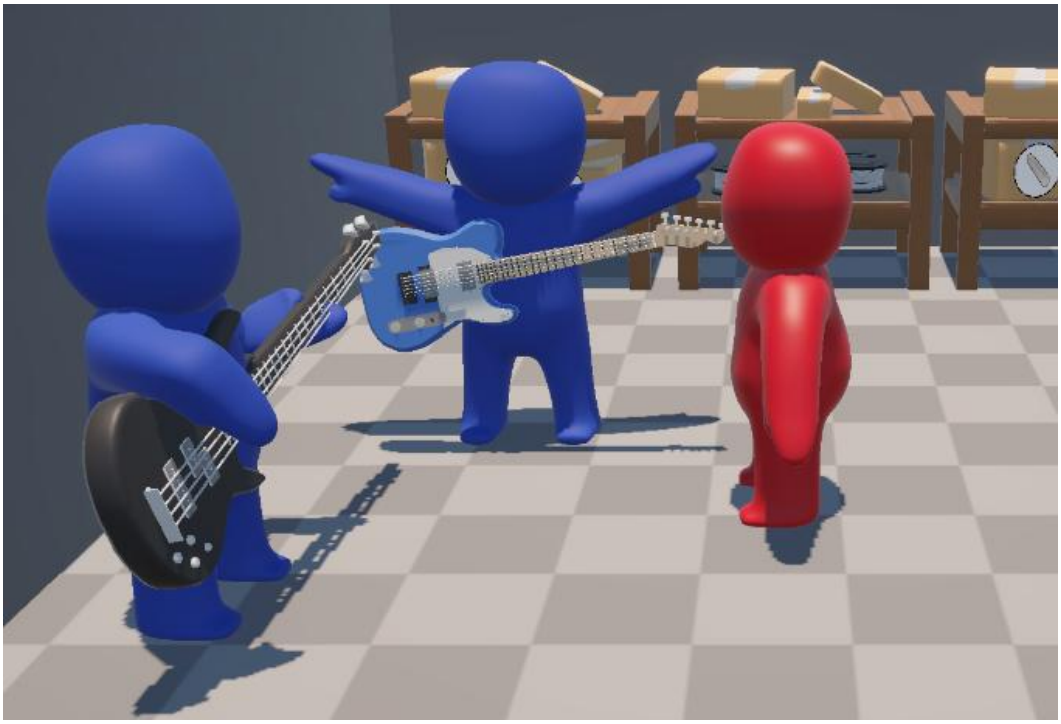


Рисунок 3.25 – Різні анімації музикантів у залежності від стану

3.8 InstrumentManager

Для управління усіма музикантами та їх поломками використовується клас InstrumentManager. Він слідкує за глобальним таймером, після завершення якого випадково обирає одного з музикантів та змінює стан його інструмента на зламаний. Якщо обраний музикант вже має зламаний інструмент, система робить до трьох спроб знайти іншого шляхом випадкового вибору (рис. 3.26).

```
private void breakInstrument()
{
    MusicianScript brokenMusicianScript = musicianScriptArray[UnityEngine.Random.Range(0, musicianScriptArray.Length)];

    if (!brokenMusicianScriptList.Contains(brokenMusicianScript) && !brokenMusicianScript.isBroken)
    {
        brokenMusicianScript.isBroken = true;
        brokenMusicianScriptList.Add(brokenMusicianScript);
        OnTimerActivated?.Invoke(this, EventArgs.Empty);
        brokenTries = 0;
    }
    else
    {
        brokenTries++;

        if (brokenTries >= 3)
        {
            foreach (MusicianScript musicianScript in musicianScriptArray)
            {
                if (!brokenMusicianScriptList.Contains(musicianScript) && !musicianScript.isBroken)
                {
                    musicianScript.isBroken = true;
                    brokenMusicianScriptList.Add(musicianScript);
                    OnTimerActivated?.Invoke(this, EventArgs.Empty);
                    Debug.LogWarning("Forced break after 3 failed tries.");
                    break;
                }
            }

            brokenTries = 0;
        }
    }
}
```

Рисунок 3.26 – Фрагмент програмної реалізації поломок музикантів

У разі невдачі, поломка примусово присвоюється першому доступному музиканту. Коли інструмент відремонтовано, скрипт прибирає музиканта зі списку зламаних. Керування музикантами у централізованому скрипті дозволяє гарантувати періодичні поломки і уникнути проблем з передачею інформації до кожного музиканта окремо.

На рисунку 3.27 зображено вікно InstrumentManager з доступними для налаштування параметрами часу та обраними музикантами, які знаходяться у спеціальному масиві.

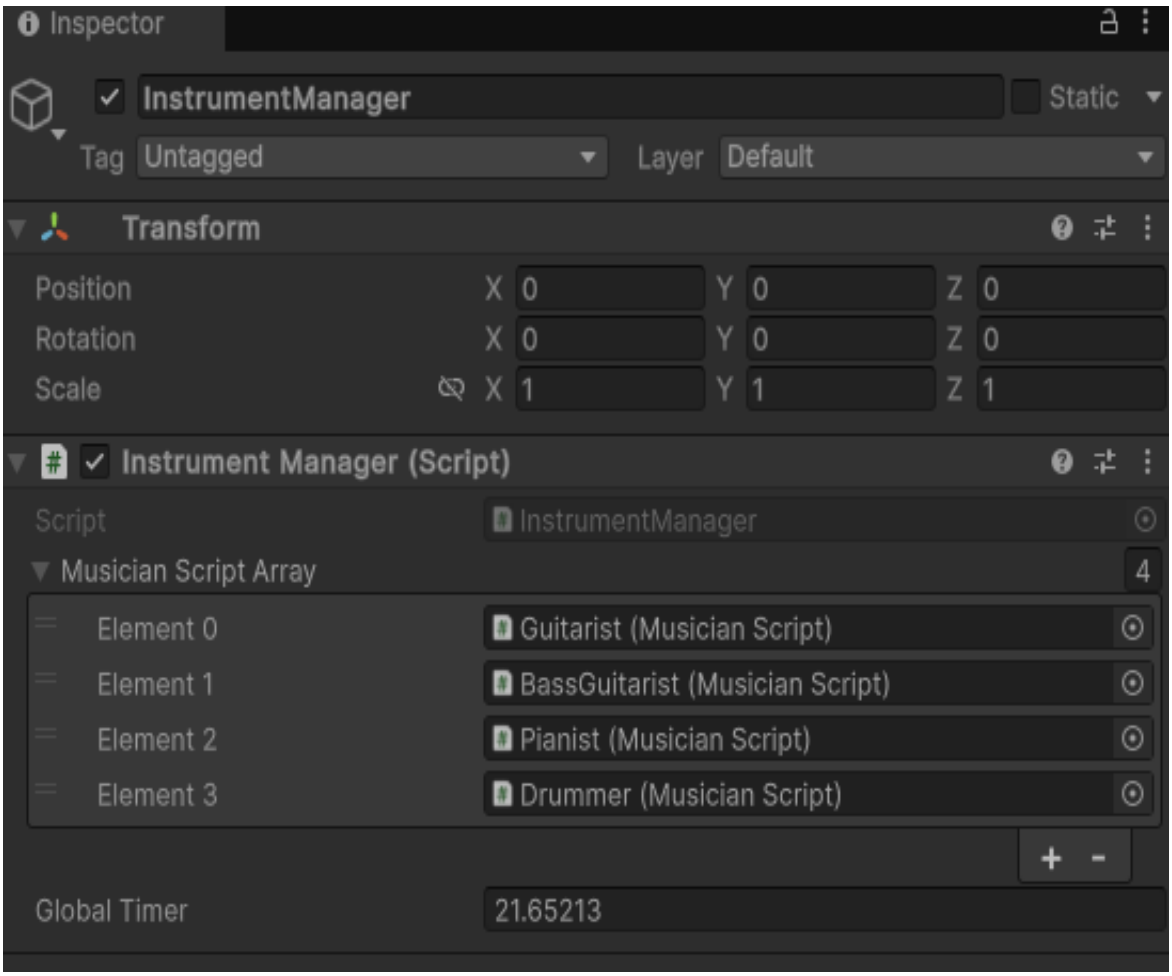


Рисунок 3.27 – Вікно InstrumentManager у редакторі рушія

3.9 Ремонт інструментів

Виправлення поломок переносних інструментів виконується за допомогою спеціального об’єкта – стола для ремонту. Стіл для ремонту зберігає список усіх доступних для ремонту інструментів, що робить неможливим додавання на стіл їх складових частин, які ніколи не будуть потребувати ремонту. Його клас CraftTable наслідує BaseStorage та переписує базовий метод Interact. При взаємодії стіл перевіряє коректність об’єкта, який гравець несе у руках. Якщо об’єкт є допустимим, він розміщується на стіл. Після цього система викликає подію, яка візуально відображає частини, що потребують заміни (рис. 3.28). Якщо гравець не тримає предметів у руках, а на столі вже лежить інструмент, він може його забрати.



Рисунок 3.28 – CraftTable відображає необхідні для ремонту запчастини

На рисунку 3.29 зображена логіка альтернативної взаємодії зі столом для ремонту. Якщо на столі вже розміщено інструмент і гравець підходить із відповідною частиною, система перевіряє, чи ця частина підходить для ремонту саме цього інструмента. У разі відповідності запчастина автоматично додається до інструмента, а об'єкт у руках гравця зникає. Після цього CraftTable обновлює візуальне відображення необхідних запчастин.

```
public override void InteractAlternate(Player player)
{
    if (HasMovableObject())
    {
        if (player.HasMovableObject())
        {
            if (GetMovableObject() is CompleteInstrument)
            {
                CompleteInstrument completeInstrument = GetMovableObject() as CompleteInstrument;
                if (completeInstrument.TryAddCompleteInstrumentPart(player.GetMovableObject().GetMovableObjectSO()))
                {
                    OnCompleteInstrumentPartAdded?.Invoke(this, new CompleteInstrumentPartActionEventArgs { brokenCompleteInstrumentPartSOList = completeInstrument
                    player.GetMovableObject().DestroySelf();
                }
            }
        }
    }
}
```

Рисунок 3.29 – Реалізація альтернативної взаємодії з CraftTable

3.10 Музика

Для відеогри було написано музичне супроводження у FL Studio. Для правильної роботи логіки аудіо потрібно розділити на 4 доріжки, де кожна буде відповідати за свій інструмент (рис. 3.30).



Рисунок 3.30 – Розділений на різні доріжки аудіофайл

Кожна доріжка з’єднана зі своїм музикантом за допомогою компонента AudioSource. Таким чином за кожен аудіофайл буде відповідати свій MusicianScript (рис. 3.31).

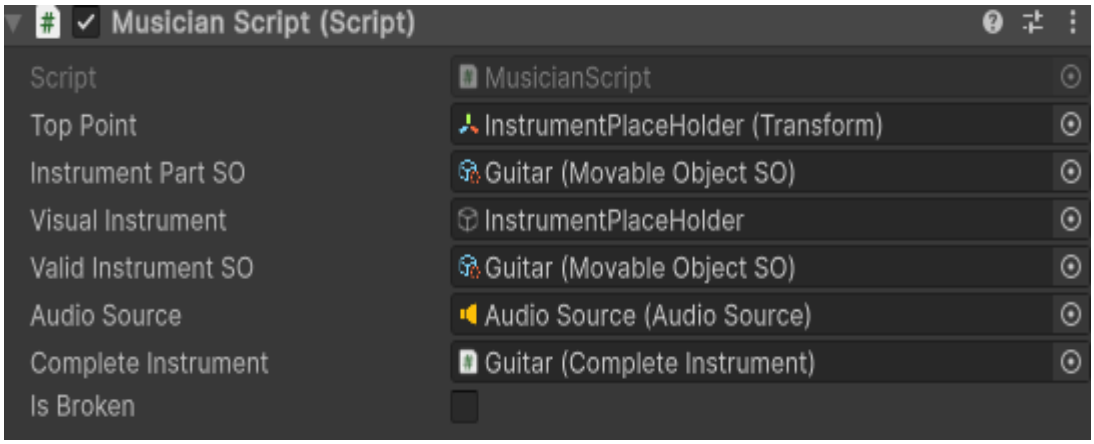


Рисунок 3.31 – Посилання на аудіодоріжку усередині класу музиканта

Оскільки у класі MusicianScript вже передбачені стани, що відповідають за зміну анімацій, ми можемо використати їх також для керування музикою. Замість того, щоб вимикати аудіо повністю, система плавно змінює гучність музики до нуля. Для того, щоб така зміна не заважала основному процесу гри та

не блокувала іншу логіку на час виконання, у Unity передбачено використання Coroutine. Вона дозволяє виконувати асинхронні дії поза основним потоком.

Після того, як у MusicianScript змінюється стан, система викликає методи FadeInAudio або FadeOutAudio, які відповідають за поступове зменшення та збільшення гучності звукової доріжки (рис. 3.32).

```
                    OnInstrumentFixed?.Invoke(this, new OnInstrumentFixedEventArgs
                    {
                        musicianScript = this
                    }); //event

                    Debug.Log(this + " is fixed");
                    StartCoroutine(FadeInAudio());
                }
            }
        }
    }
}

Ссылок: 2
private IEnumerator FadeInAudio()
{
    float fadeDuration = 1f;
    float targetVolume = .5f;

    while(audioSource.volume < targetVolume)
    {
        audioSource.volume += targetVolume * Time.deltaTime / fadeDuration;
        yield return null;
    }
}
Ссылок: 1
private IEnumerator FadeOutAudio()
{
    float fadeDuration = 1f;
    float startVolume = audioSource.volume;

    while (audioSource.volume > 0f)
    {
        audioSource.volume -= startVolume * Time.deltaTime / fadeDuration;
        yield return null;
    }
}
```

Рисунок 3.32 – Виклик методів FadeAudio після зміни стану

3.11 Інтерфейс

Було розроблено систему виводу візуальних підказок про зламані інструменти. У той момент, коли глобальний таймер добігає кінця і музичний інструмент виходить зі строю, викликається спеціальна подія. Вона відображає та оновлює список зламаних інструменти та їх назву на екрані (рис. 3.33).

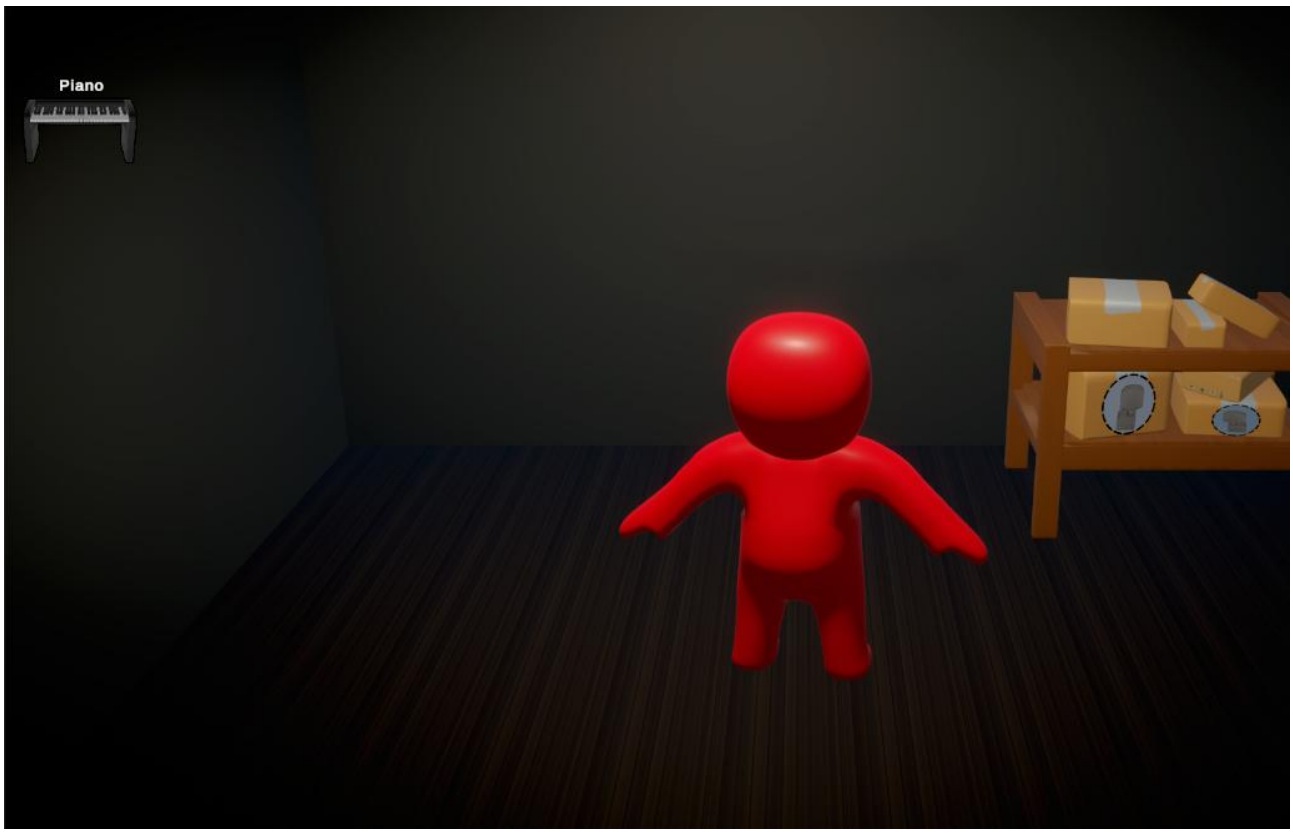


Рисунок 3.33 – Візуальне відображення зламаних інструментів

Також, для стаціонарних інструментів була реалізована своя система виводу зображень, схожа на ту, що використовується у CraftTable (рис. 3.34).

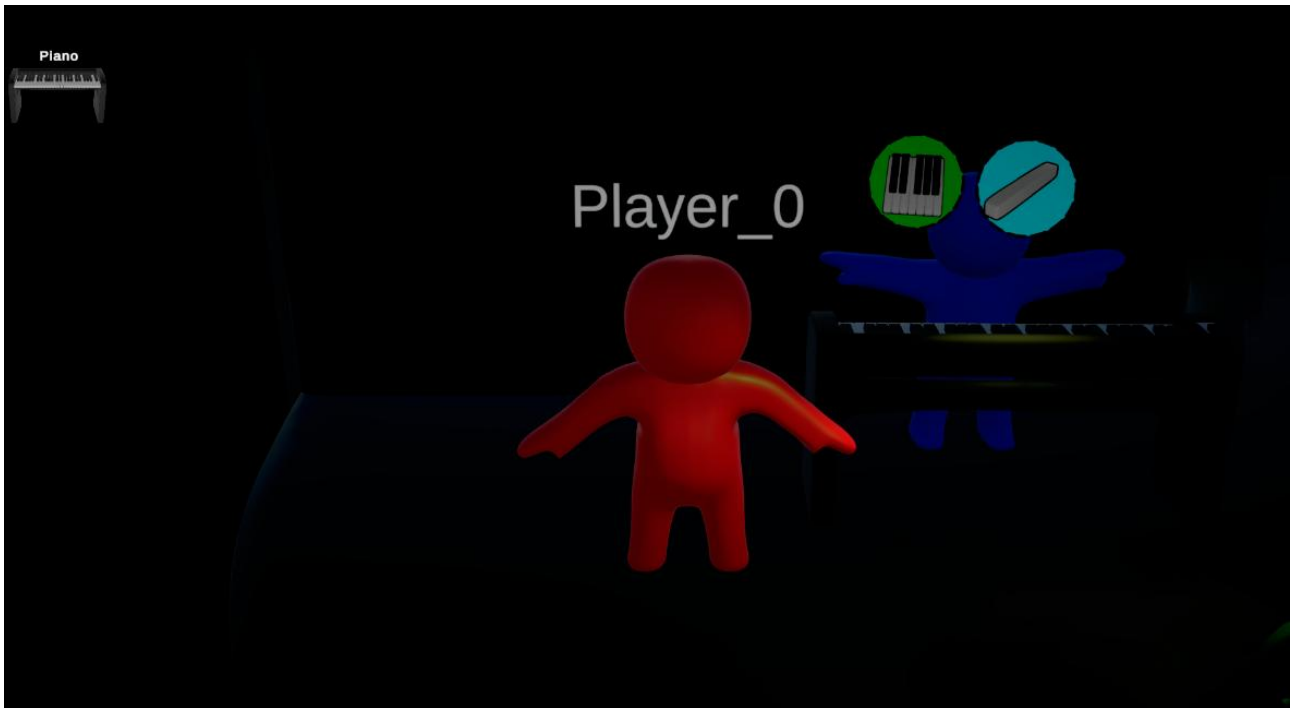


Рисунок 3.34 – Вивід необхідних для ремонту запчастин

3.12 Мережева логіка

Після розробки всієї логіки гри починається процес додавання мережевого з’єднання та синхронізації подій ігрового світу. Для виконання синхронізації ми повинні додати спеціальний компонент NetworkObject, який сигналізує рушію, що ми бажаємо передавати поточний об’єкт по мережі, а не виконувати його локально.

3.12.1 NetworkManager

Використовуючи бібліотеку NetCode ми можемо звести на мінімум потребу в переробці логіки. Для роботи бібліотеки потрібно створити компонент NetworkManager, який буде відповідати за підключення гравців (рис. 3.35).

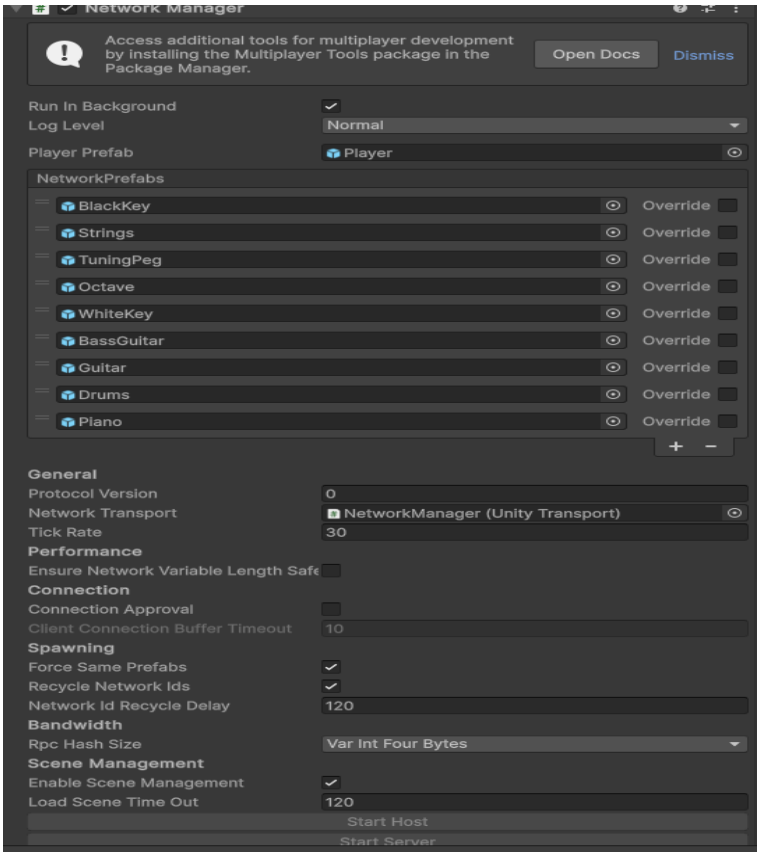


Рисунок 3.35 – Компонент NetworkManager

Усередині NetworkManager потрібно вказати на об’єкт гравця, а також обрати усі предмети, які будуть генеруватися сервером і передаватися до клієнтів. Такими предметами є музичні інструменти та запчастини до них.

3.12.2 Меню підключення

Для спрощення процесу підключення було створене спеціальне меню, яке викликається на початку гри. Воно включає у себе необхідні для з’єднання дані, такі як IP-адреса, порт підключення, ім’я гравця, а також кнопки для створення і підключення до сервера (рис. 3.36). Ці дані передаються до NetworkManager при натисканні однієї з кнопок підключення, що забезпечує простий спосіб підключення гравців.

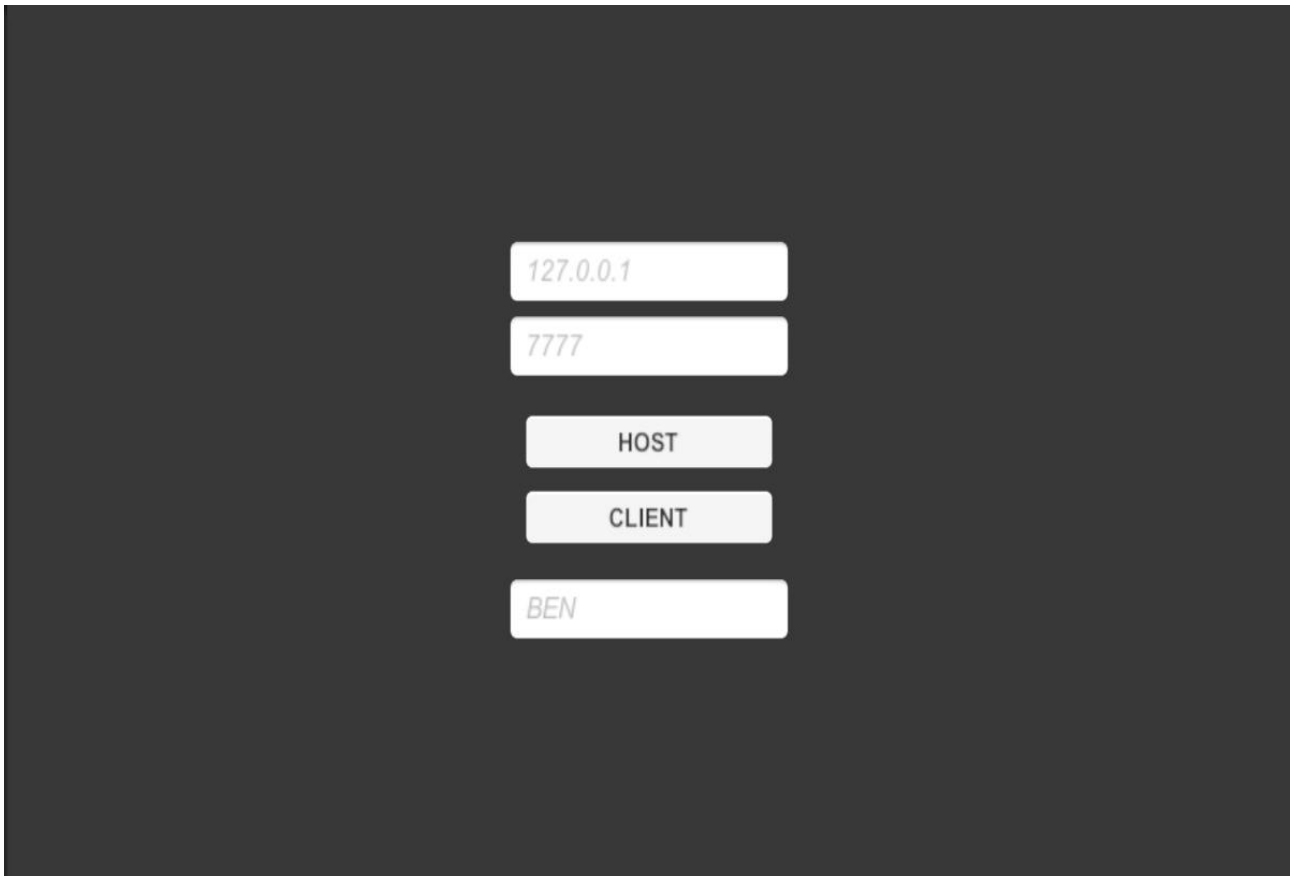


Рисунок 3.36 – Меню підключення до гри

3.12.3 Синхронізація подій

Синхронізація подій у грі реалізується через розподіл ролей сервера і клієнтів, де сервер контролює стан гри. Клієнти відправляють запити на сервер за допомогою Remote Procedure Calls – спеціальних викликів методів, які виконуються на сервері або на інших клієнтах [20]. Сервер оброблює ці запити та оновлює загальний стан гри відправляючи його до клієнтів за допомогою Server Remote Procedure Calls, що забезпечує однакову картину світу у всіх гравців.

На рисунку 3.37 зображена логіка синхронізації поломки та ремонту музичних інструментів.

```
Ссылок: 1
private void breakInstrument()
{
    int breakInstrumentIndex = UnityEngine.Random.Range(0, musicianScriptArray.Length);
    SpawnNewBrokenMusicianClientRpc(breakInstrumentIndex);
}

[ClientRpc]
Ссылок: 1
private void SpawnNewBrokenMusicianClientRpc(int breakInstrumentIndex)
{
    MusicianScript brokenMusicianScript = musicianScriptArray[breakInstrumentIndex];

    if (!brokenMusicianScriptList.Contains(brokenMusicianScript) && !brokenMusicianScript.isBroken)
    {
        brokenMusicianScript.isBroken = true;
        if (!brokenMusicianScriptList.Contains(brokenMusicianScript))
        {
            brokenMusicianScriptList.Add(brokenMusicianScript);
            OnTimerActivated?.Invoke(this, EventArgs.Empty);
            brokenTries = 0;
        }
    }
    else
    {
        brokenTries++;

        if (brokenTries >= 3)
        {
            foreach (MusicianScript musicianScript in musicianScriptArray)
            {
                if (!brokenMusicianScriptList.Contains(musicianScript) && !musicianScript.isBroken)
                {
                    musicianScript.isBroken = true;
                    brokenMusicianScriptList.Add(musicianScript);
                    OnTimerActivated?.Invoke(this, EventArgs.Empty);
                    Debug.LogWarning("Forced break after 3 failed tries.");
                    break;
                }
            }

            brokenTries = 0;
```

Рисунок 3.37 – Фрагмент коду синхронізації музичних інструментів

Як можна побачити з наведеного фрагменту коду, логіка, яка була реалізована раніше для класу InstrumentManager тепер використовує ClientRpc.

Це означає, що кожен раз коли музичний інструмент виходить зі строю, сервер оновлює стан інструмента та сповіщає всіх клієнтів про зміну.

За такою ж системою була синхронізована уся необхідна ігрова логіка. Це включає у себе рух та взаємодію гравців, підняття та збереження предметів, анімації, ремонт, ігрові події, інтерфейс та звук.

Окремо потрібно виділити процес синхронізації створення та знищення ігрових об’єктів. Компонент NetCode вимагає, щоб події такого типу виконувалися виключно сервером. Для виконання цих умов був створений об’єкт GameMultiplayer і клас з такою ж назвою. Він містить список усіх інструментів та запчастин, з якими гравці будуть взаємодіяти. Основною задачею цього класу є делегування серверу відповідальності за створення та знищення об’єктів. Тепер клієнти лише ініціюють запити, а фактичне виконання дій відбувається на сервері за допомогою методу RPC (рис. 3.38).

```
Ссылок: 1
public void DestroyObject(MovableObject movableObject)
{
    DestroyObjectServerRpc(movableObject.NetworkObject);
}

[ServerRpc(RequireOwnership = false)]
Ссылок: 1
private void DestroyObjectServerRpc(NetworkObjectReference movableObjectNetworkObjectReference)
{
    movableObjectNetworkObjectReference.TryGet(out NetworkObject movableObjectNetworkObject);
    MovableObject movableObject = movableObjectNetworkObject.GetComponent<MovableObject>();

    ClearMovableObjectOnParentClientRpc(movableObjectNetworkObjectReference);

    movableObject.DestroySelf();
}
```

Рисунок 3.38 – Методи знищення об’єктів у класі GameMultiplayer

3.12.4 Ігровий рахунок

Для реалізації ігрового рахунку був створений клас ScoreManager. Він відповідає за відображення та зміну нарахованих балів. Після початку гри ScoreManager встановлює стартовий рахунок у обсязі 1000 балів (рис. 3.39).

Слідкуючи за ігровими подіями, скрипт буде віднімати по 10 балів за кожного непрацюючого музиканта з періодичністю в 2.5 секунди (рис. 3.40).



Рисунок 3.39 – Відображення актуального рахунку на екрані гравців

```
Ссылок: 1
private IEnumerator CheckBrokenInstruments()
{
    while (isGameRunning)
    {
        var list = InstrumentManager.Instance.GetBrokenMusicianList();

        if (list.Count > 0)
        {
            startScore -= 10 * list.Count;
            scoreText.text = startScore.ToString();

            if (startScore <= 0)
            {
                ShowGameOverClientRpc();
                yield break; // Stop the coroutine
            }
        }

        yield return new WaitForSeconds(2.5f); // wait 2.5 second before next check
    }
}
```

Рисунок 3.40 – Перевірка стану музикантів та зменшення рахунку

У випадку, якщо ігровий рахунок знизиться до нуля, клас ScoreManager викликає метод ShowGameOverClientRpc, який призупиняє ігровий процес для всіх клієнтів. Реалізується блокування подальшої взаємодії шляхом паузи

ігрового часу. Окрім цього вимикається звуковий супровід, що забезпечує повне завмирання гри. Після цього на екран виводиться спеціальне полотно з повідомленням про завершення гри (рис. 3.41).



Рисунок 3.41 – Екран програшу

3.13 Висновок за третім розділом

У третьому розділі роботи був розроблений та описаний процес створення гри у Unity. Основні етапи включали створення проєкту, реалізація руху та системи взаємодії гравця, логіка предметів та музичних інструментів, створення системи сховищ та видачі об’єктів. Також була створена логіка поломки та ремонту інструментів. Розроблена візуальна частина – користувацький інтерфейс, який включає у себе іконки, зображення та меню. Була проведена робота з аудіофайлами та поступовою зміною їх гучності, що створює плавний ефект переходу між станами музикантів. Після завершення основної логіки була проведена робота над синхронізацією станів клієнтів та сервера. За допомогою рушія Unity та бібліотеки NetCode була розроблена система створення сервера та підключення до нього. Додатково був реалізований ігровий рахунок та кінцеве меню гри, яке викликається після того, як рахунок стає нульовим.

ВИСНОВКИ

У даній кваліфікаційній роботі було розроблено мережеву гру з використанням ігрового рушія Unity. Метою роботи було створення багатокористувацької кооперативної гри, у якій гравці виступають у ролі техніків, що забезпечують безперебійний музичний концерт.

На етапі аналізу предметної області було досліджено сучасні тенденції ігрової індустрії, зокрема популярність багатокористувацьких ігор, архітектуру мережевих з’єднань та існуючі аналогічні проекти. Було описано вибір технологій для розробки, серед яких ігровий рушій Unity, бібліотека Netcode for GameObjects, графічний редактор Blender та аудіоінструменти FL Studio і Adobe Audition.

У розділі проєктування було визначено основні компоненти системи, такі як гравець, музиканти, інструменти, сховища, стіл для ремонту та інтерфейс. За допомогою діаграм класів та варіантів використання було спроектовано логіку взаємодії між об’єктами та гравцями. Також було розглянуто мережеву архітектуру, включаючи клієнт-серверну модель та синхронізацію станів.

Програмна реалізація включала створення ігрової сцени, моделювання персонажів та об’єктів, розробку системи руху та взаємодії гравця, логіку предметів, сховищ та ремонту інструментів. Було реалізовано музичний супровід, інтерфейс користувача та систему ігрового рахунку. Важливим етапом стала інтеграція мережевого коду, що забезпечило синхронізацію подій між клієнтами та сервером.

У результаті було створено функціональну мережеву гру, у якій два гравці можуть спільно виправляти поломки музичних інструментів, підтримуючи високий ігровий рахунок. Проєкт демонструє ефективне використання Unity для розробки багатокористувацьких ігор, а також потенціал для подальшого вдосконалення, такого як додавання нових рівнів, інструментів та механік.

Робота підтвердила актуальність обраної теми та доцільність використання сучасних інструментів для створення інтерактивних розважальних додатків. Отриманий досвід може бути корисним для подальших досліджень у галузі розробки ігор та мережевих технологій.

ПЕРЕЛІК ПОСИЛАНЬ

1. Video game. Wikipedia. URL : http:// en.wikipedia.org/wiki/Video_game (дата звернення : 10.01.2025).

2. What is Online Games. IGI Global. URL : https://www.arm.com/glossary/ gaming-engines (дата звернення : 12.01.2025).

3. Single-player : What Does It Mean in Gaming?. Onlyfarms. URL : https://onlyfarms.gg/wiki/general/single-player-meaning-in-gaming (дата звернення : 12.01.2025).

4. Unity – Manual : Network Clients and Servers. Unity. URL : https://docs.unity3d.com/530/Documentation/Manual/UNetClientServer.html (дата звернення : 13.02.2025).

5. Format Wars : Online vs. Lan. Medium. URL : https://espoworld.medium.com/format-wars-online-vs-lan-51ef80b53508 (дата звернення : 15.02.2025).

6. Modern Game Networking Models. Rune.ai. URL : https://developers.rune.ai/blog/modern-game-networking-models (дата звернення : 17.02.2025).

7. Asynchronous Multiplayer. Lark. URL : https://www.larksuite.com/ en_us/topics/gaming-glossary/asynchronous-multiplayer (дата звернення : 17.02.2025).

8. Client-Server Architecture Explained. Medium. URL : https://medium.com/nerd-for-tech/client-server-architecture-explained-with-examples-diagrams-and-real-world-applications-407e9e04e2d1 (дата звернення : 21.02.2025).

9. What is Client-Server Network? Zenarmor. URL : https://www.zenarmor.com/docs/network-basics/what-is-client-server-network (дата звернення : 21.02.2025).

10. Buttfield-Addison P., Manning J., Nugent T. Unity Game Development Cookbook : Essentials for Every Game. San Francisco, 2019. 405 с.

11. What is a Gaming or Game Engine? Arm. URL : https://www.arm.com/glossary/gaming-engines (дата звернення : 23.02.2025).

12. Unity чи Unreal Engine – плюси та мінуси. Gamedev.Dou. URL : https://gamedev.dou.ua/articles/unity-or-unreal-engine/ (дата звернення : 23.02.2025).

13. Unity Documentation. Unity. URL : https://docs.unity.com/ (дата звернення : 27.02.2025).

14. Blender Documentation. Blender. URL : https://docs.blender.org/ (дата звернення : 28.02.2025).

15. Інструкція, як будувати UML-діаграми. Dou. URL : https://dou.ua/forums/topic/40575/ (дата звернення : 15.03.2025).

16. Singleton. Refactoring.Guru. URL : https://refactoring.guru/design-patterns/singleton (дата звернення : 15.03.2025).

17. Hocking J. Unity in Action, Third Edition: Multiplatform game development in C#. New York, 2022. 416 с.

18. C# Guide – .NET manager language. Microsoft. URL : https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення : 21.04.2025).

19. C# Inheritance. W3schools. URL : https://www.w3schools.com/cs/cs_inheritance.php (дата звернення : 21.04.2025).

20. Remote Procedure Call (RPC). TechTarget. URL : https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC (дата звернення : 27.04.2025).