МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами

Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри

__ ____проф. Приходько С.Б.

«___» _____ 2025 р.

КВАЛІФІКАЦІЙНАРОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: Розробка багатокористувацької гри на Unity з використанням Photon Fusion

Виконав: студент групи 4151

_____Дроздов Д.С

(підпис, ПІБ)

Керівник роботи:

Завідуючий кафедри ПЗАС д.т.н.,проф.

(посада, науковий ступень вчене звання)

_____ _____Приходько С.Б.

(підпис, ПІБ)

Миколаїв – 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами
Кафедра програмного забезпечення автоматизованих систем
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма «Інженерія програмного забезпечення»

«ЗАТВЕРДЖУЮ»
Гарант освітньої програми
_____ доц. Макарова Л.М.
(підпис)
«__» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студенту: Дроздову Данилу Сергійовичу
(Прізвище, ім'я, по батькові)
1. Тема роботи: Розробка багатокористувацької гри на Unity з використанням Photon Fusion
Керівник роботи завідуючий кафедри ПЗАС, д.т.н. Приходько С.Б. _____ _
Затверджені наказом ректора №153 від 08.04.2025 р. _____
2. Термін подання роботи: 02.06.2025 р. _____
3. Вихідні дані по роботі: _____
_____

4. Перелік питань, що належать до розробки (найменування розділів)
Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Перелік умовних позначень, символів, одиниць та термінів (при необхідності); Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).

5. Перелік презентаційних матеріалів 1. Титульний аркуш; 2. Актуальність теми; 3. Мета розробки; 4.Аналіз вимог. Діаграма варіантів використання; 5. Архітектура проєкту ; 6. Засоби розробки; 7. Ескізний проєкт. Діаграми діяльності(діаграма діяльності для варіанту використання «Створити ігрову сесію»); 8. Ескізний проєкт. Діаграми діяльності(діаграма діяльності для варіанту використання «Почати гру»); 9. Технічний проєкт. Діаграма класів; 10. Технічний проєкт. Діаграми послідовності(Діаграма послідовності для прецеденту «Підключення до сесії»); 11. Технічний проєкт. Діаграми послідовності(Діаграма послідовності для прецеденту «Створення сесії»); 12 - 13.Робочий проєкт; 14. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 08.04.2025 р.

КАЛЕНДАРНИЙ ПЛАН

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	31.03.2025	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	07.04.2025	ПП*
3.	Аналіз вимог до ПЗ	14.04.2025	ПП*
4.	Розробка архітектури ПЗ	21.04.2025	
5.	Розробка проєкту ПЗ	05.05.2025	
6.	Реалізація та тестування ПЗ	12.05.2025	
7.	Підготовка розділу Результати розробки ПЗ	16.05.2025	
8.	Підготовка розділу з охорони праці	19.05.2025	ПП*
9.	Підготовка розділу ВИСНОВКИ	23.05.2025	
10.	Оформлення списку використаних джерел та додатків	26.05.2025	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	02.06.2025	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	06.06.2025	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	09.06.2025	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	16.06.2025	

* - за результатами переддипломної практики (ПП), яка була з 03.02.2025 до 23.03.2025 р.

Студент _____ Дроздов Д.С.
(підпис) (ПІБ)

Керівник роботи _____ Приходько С.Б.
(підпис) (ПІБ)

АНОТАЦІЯ

Кваліфікаційна робота присвячена вирішенню практичної задачі інженерії програмного забезпечення, а саме розробці багатокористувацької гри за допомогою ігрового двигуна Unity та мережевого фреймворку Photon Fusion.

В кваліфікаційній роботі була розроблена гра, яка має 2 міні-ігри. Гра має можливість вміщувати до 4 гравців в одній сесії. Кожна міні-гра має унікальні механіки та навколишнє середовище.

Об’єктом роботи для даної кваліфікаційної роботи є процес розробки мережевої багатокористувацької гри на Unity з використанням Photon Fusion. Робота включає в себе: опис та аналіз предметної галузі мережевої багатокористувацької гри на Unity, обґрунтування необхідності розробки програмного забезпечення, постановку задачі, проєкт, результати тестування, результати розробки, розділ з охорони праці та додатки.

В роботі використовувались різні архітектурні принципи, фреймворки та бібліотеки для досягнення кінцевого результату. Була розроблена сильна мережева архітектура, яка дозволила легко масштабувати проєкт та створювати нові міні-ігри. Проєкт вийшов оптимізованим і швидким у виконанні, що дозволило збирати гру на мобільних пристроях. Оптимізація міні-ігор дозволяє виконувати важкі ігрові дії та візуалізувати гарні сцени. Тестування всіх мережев\их компонентів було вдалим, повний ігровий цикл працює повноцінно без помилок.

Висновком роботи є успішне інтегрування фреймворку Photon Fusion з ігровим двигуном Unity. Основні архітектурні принципи та фреймворки дозволяють створити високоефективний та легко розширювальний проєкт/код.

Кваліфікаційна робота викладена на 107 сторінках друкованого тексту, містить 42 рисунки, 4 таблиці, список використаних джерел з 10 найменувань та 5 додатків.

ABSTRACT

The qualification work is devoted to solving a practical software engineering problem, namely the development of a multiplayer game using the Unity game engine and the Photon Fusion networking framework.

In the qualification work, a game was developed that has 2 mini-games. The game has the ability to accommodate up to 4 players in one session. Each mini-game is unique in terms of mechanics and environment.

The object of research for this qualification work is the process of developing a networked multiplayer game on Unity using Photon Fusion. A practical task was analysed and existing analogues and methods for developing networked multiplayer games were analysed.

Various architectural principles, frameworks, and libraries were used to achieve the final result. A strong network architecture was developed, which made it easy to scale the project and create new mini-games. The project turned out to be optimised and fast in execution, which allowed the game to be launched on mobile devices. The optimisation of mini-games allows you to perform heavy game actions and visualise beautiful scenes. The testing of all network components was successful, and the full game cycle works properly without errors.

The work includes: description and analysis of the subject area of a networked multiplayer game on Unity, justification of the need for software development, problem statement, design, test results, development results, labour protection section and applications.

The conclusion of the work is the successful integration of the Photon Fusion framework with the Unity game engine. The basic architectural principles and frameworks allow creating a highly efficient and easily extensible project/code.

The qualification work is presented on 107 pages of printed text, contains 42 figures, 4 tables, a list of 10 references and 5 appendices.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ І ТЕРМІНІВ

А

Ассети – загальна назва до файлів, текстур, звуків, моделей.

Б

Бінд залежностей – створення прив’язки конкретного типу об’єкта до іншого типу, або абстракції.

К

Колайдер – фізична фігура, яка надає об’єкту можливість взаємодіяти з фізичним світом ігрового двигуна.

Р

Рендеринг – процес візуалізації пристроєм картинки програми на екрані.

Ф

Фреймворк – збірка, структура бібліотек, які повністю/або частково пропонують свою архітектуру проєкту.

Х

Хост – керуючий сесією гри, той хто створює сервера, кімнати в багатокористувацьких іграх.

ВСТУП

Сучасний ігровий ринок демонструє стрімкий розвиток багатокористувацьких ігор, які забезпечують гравцям можливість спільної взаємодії у віртуальному середовищі. Висока популярність таких проєктів зумовлюється як зростаючою доступністю мережевих технологій, так і бажанням гравців брати участь у соціальних та кооперативних ігрових активностях.

Unity є одним із найпопулярніших ігрових двигунів завдяки своїй багатофункціональності, кросплатформеності та простоті у використанні. В реалізації багатокористувацьких ігор важливу роль відіграють спеціалізовані мережеві фреймворки, такі як Photon Fusion, що надають широкий спектр інструментів для створення стабільних, масштабованих та ефективних мережевих ігрових проєктів.

Використання Photon Fusion дозволяє вирішувати завдання синхронізації гравців у режимі реального часу, підтримки низької затримки та забезпечення стабільності з'єднання, що робить його перспективним інструментом для створення багатокористувацьких ігор. Таким чином, дослідження методів розробки багатокористувацьких ігор з використанням Unity та Photon Fusion є актуальним у контексті сучасних вимог до ігрових продуктів.

Метою кваліфікаційної роботи є розв’язання практичної задачі інженерії програмного забезпечення, а саме розробка багатокористувацької гри з використанням ігрового двигуна Unity та фреймворку Photon Fusion, яка забезпечує стабільну мережеву взаємодію між користувачами, високий рівень продуктивності та якісний ігровий досвід. Для досягнення поставленої мети необхідно вирішити наступні задачі**:**

- Аналіз практичної задачі для розробки багатокористувацької гри.
- Аналіз існуючих підходів до розробки багатокористувацьких ігор на базі Unity.

- Розробка технічного дизайну та архітектури ігрового проєкту, який включає механізми мережевої взаємодії.
- Реалізація багатокористувацької гри.
- Тестування гри на відповідність вимогам стабільності, продуктивності та зручності.

Об’єктом для даної кваліфікаційної роботи є процес розробки мережевої багатокористувацької гри на Unity з використанням Photon Fusion.

Практична значущість полягає у створенні прототипу багатокористувацької гри, який може бути використаний як основа для комерційних проєктів, навчальних матеріалів або вдосконалення існуючих рішень у галузі розробки багатокористувацьких ігор.

1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ПОСТАНОВКА ЗАДАЧІ НА РОЗРОБКУ МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY

1.1 Аналіз практичної задачі

Розробка багатокористувацьких ігор є складною задачею програмного забезпечення, що включає обробку даних, синхронізацію станів гравців та оптимізацію мережевого трафіку. Кваліфікаційна робота присвячена створенню багатокористувацької гри на Unity з використанням Photon Fusion, що є сучасним фреймворком від компанії Photon для реалізації високопродуктивних мережевих рішень. Задача допоможе зрозуміти плюси і мінуси вибраного мережевого фреймворку для двигуна Unity. Розглянемо такі основні багатокористувацькі мережеві фреймворки для Unity:

- **Photon PUN** – один з мережевих фреймворків компанії Photon. Він є полегшенною і в той час старою версією Photon Fusion. Добре підійде для ознайомлення з багатокористувацькістю та легкими проєктами. Але не є гарним варіантом для конкурентоспроможних проєктів, які хочуть створити міцну та розвинену багатокористувацьку архітектуру.
- **Photon Quantum** – мережевий фреймворк компанії Photon. Є найсильнішою, високоефективною версією багатокористувацьких фреймворків від Photon. Він використовує інші архітектурні шаблони проєктування і зроблений з упором на фізику. Підходить для сильних і великих проєктів, які розраховують отримати оптимізований, високоефективний багатокористувацький продукт. Вимагає більших знань, ніж інші версії Photon-а, не підійде для новачків. Також плата за кожного юзера і підтримка такого проєкту буде на порядок дорожче.
- **Photon Fusion** – ще один фреймворк компанії Photon. Він є новою і більш сильною версією старого Photon PUN. Не такий сильний, як Quantum, але

ідеально підходить для малих, середніх проєктів з невеликим бюджетом і часом на створення гри. Є кращим вибором для більшості варіантів.

- **Mirror** – open source безкоштовний фреймворк. Є також ефективним і сильним багатокористувацьким інструментом. Але більше підходить для малих проєктів, або для розробки без необхідності в сторонніх серверах. Photon Fusion краще підходить для більших проєктів, де потрібна висока масштабованість і продуктивність. Mirror має менше функцій, які могли б допомогти в розробці і керуванні проєктом.
- **Netcode -** це офіційний фреймворк від Unity для створення багатокористувацьких ігор. Бідний на фунціонал, має менше можливостей в порівнянні з Photon і Mirror. Підходить для невеликих проєктів, які хочуть максимально взаємодіяти з Unity екосистемою.

1.2 Опис існуючих аналогів багатокористувацьких ігор, створених на Unity

Гра VRChat (Логотип гри представлений на рисунку 1.1)
Мережевий фреймворк: Photon PUN

Рисунок 1.1 – Логотип гри VRChat [1]

VRChat – це багатокористувацька кросплатформна гра (PC і VR), де гравець може спілкуватись, взаємодіяти з іншими людьми, гарно проводити час і відпочивати.

Гра Stumble Guys (Логотип гри представлений на рисунку 1.2)
Мережевий фреймворк: Photon Quantum

Рисунок 1.2 – Логотип гри Stumble Guys [2]

Stumble Guys – це аналог Fall Guys, але який доступний як на комп’ютері, як і на телефоні. Гравець може бігати з іншими гравцями (до 32 в матчі) та проходити перешкоди, які з’являються на його шляху.

Гра Liar's Bar (Логотип гри представлений на рисунку 1.3)
Мережевий фреймворк: Mirror

Рисунок 1.3 – Логотип гри Liar's Bar [3]

Liar's Bar – це весела та цікава гра до 4 гравців, яка включає в себе настільні ігри: карти, рулетку, кидання кубика. Хто програє міні-гру – той крутить барабан і стріляє. Грайте за різних тварин, в різних локаціях.

Гра Lethal Company (Логотип гри представлений на рисунку 1.4)
Мережевий фреймворк: Unity Netcode

Рисунок 1.4 – Логотип гри Lethal Company [4]

Lethal Company – гра де ваша задача вкрасти якомога більше деталей з іноземної планети, яка населена монстрами. Збирайте деталі і відносіть на корабель, потім здавайте і виконуйте завдання.

Таким чином є потреба в створенні власної гри на Photon Fusion. Так як в порівнянні з іншими конкурентами, він є одним з найкращих варіантів для створення невеликої багатокористувацької гри на Unity. З трохи вищим порогом ніж в Photon Pun, ми будемо мати кращий фунціонал і результат. Наша гра не буде мати великої архітектури і цілі обробляти багато фізичних компонентів і об’єктів, тому Photon Quantum нам теж не потрібен. Unity Netcode не потрібний в міру того, що не достатньо розвинений. Також, нам для тесту і робочого функціювання гри потрібний невеликий робочий сервер, який буде синхронізувати гру. На відміну від Mirror, Photon дає стартову частинку серверу (на 20 людей), що дасть змогу запустити і протестувати гру.

Reference

Гра, яка буде натхненням і прикладом для цієї роботи, це – Pummel Party (Логотип гри представлений на рисунку 1.5).

Рисунок 1.5 – Логотип гри Pummel Party [5]

Pummel Party побудована на основі міні-монополії, де гравець з друзями (до 8 людей) можете зіграти і провести гарно час. Кожен раунд гравець граєте міні-гру і заробляєте очки, хто більше набрав – той переміг і отримує кращі призи для монополії. Після міні-гри гравці кидають кубик і ходять. Щоб перемогти гравцю потрібно пройти карту і знайти кубки. Той, хто знайде 3 кубка, – перемагає. В цей час друзі гравця можуть заважати йому і вибивати різні речі, які він знаходитиме після міні-гри.

В даній грі не буде монополії, так як для цього потрібно більше часу. Будуть міні-ігри. Але, маючи навіть 2 міні-гри, можна вдосталь повеселитись з друзями і гарно провести час. Міні-ігри будуть мати різні цікаві локації і механіки.

1.3 Постановка задачі на створення багатокористувацької гри

На основі аналізу задачі проєкту та аналогів фреймворків, які були зазначені в пункті 1.2, була визначена необхідність створення гри з використанням Photon Fusion, що забезпечує ефективну мережеву взаємодію, високу продуктивність та безкоштовний доступ до частинки серверу для підключення гравців.

Практичною задачею є створення високоефективної багатокористувацької гри на Unity до 4 гравців. Потрібно розробити багатокористувацьку гру на Unity з використанням Photon Fusion, яка задовольняє наступним вимогам:

- Легко масштабована, оптимізована архітектура проєкта і головної ігрової сцени.
- Оптимізоване використання мережевих ресурсів.
- Надійна синхронізація між гравцями в усіх можливих випадках.
- Ігрове лобі та декілька ігрових рівнів.
- Цікаві і захоплюючі міні-ігри.
- Зручне керування гравця.
- Зрозумілий користувацький інтерфейс.

Вимоги до складу виконуваних функцій

Гра повинна мати можливість виконання наступних функцій:

- Підключення до сесії: дозволяє гравцям під’єднатися до існуючої ігрової сесії.
- Створення нової сесії: дозволяє гравцю створити нову ігрову сесію.
- Налаштування сесії: дозволяє гравцю налаштувати новостворену ігрову сесію під власні потреби.

- Початок гри: дозволяє почати гру для всіх гравців, які знаходяться в спільній ігровій сесії.
- Синхронізація станів: дозволяє серверу синхронізовувати дані між гравцями.
- Надання списку активних сесій: надає можливість системі отримувати інформацію про активні сесії на сервері.

Вимоги до інформаційної та програмної сумісності

- Програма повинна працювати на всіх версіях Android починаючи з 27 версії.

Вимоги до складу та параметрів технічних засобів

- RAM: 2 GB
- До 150 МБ пам’яті.

Деталізація постановки задачі на розробку гри на Unity з використанням фреймворку Photon Fusion знаходиться в технічному завданні, яке представлене у додатку А.

2 СТВОРЕННЯ МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY

2.1 Аналіз вимог до багатокористувацької гри на Unity

У цьому підрозділі було виконано аналіз вимог до мережевої багатокористувацької гри на Unity з використанням фреймворку Photon Fusion за допомогою побудови моделі варіантів використання та розробки їх специфікацій.

2.1.1 Побудова моделі варіантів використання мережевої багатокористувацької гри на Unity

Модель варіантів використання є ключовим етапом у проєктуванні багатокористувацької гри, оскільки вона визначає взаємодію між користувачами та системою. У контексті розробки гри на Unity з використанням Photon Fusion, **основними акторами** є:

- Гравець – користувач, який взаємодіє з грою, здійснюючи вхід, приєднання до сесій, керування персонажем та участь у грі.
- Сервер гри – обробляє логіку гри, синхронізує стан між клієнтами та забезпечує цілісність гри.

Основні варіанти використання включають:

- Підключення до сесії: дозволяє гравцям під’єднатися до існуючої ігрової сесії.
- Створення нової сесії: дозволяє гравцю створити нову ігрову сесію.
- Налаштування сесії: дозволяє гравцю налаштувати новостворену ігрову сесію під власні потреби.

- Початок гри: дозволяє почати гру для всіх гравців, які знаходяться в спільній ігровій сесії.
- Синхронізація станів: дозволяє серверу синхронізовувати дані між гравцями.
- Надання списку активних сесій: надає можливість системі отримувати інформацію про активні сесії на сервері.

Ця модель дозволяє чітко визначити функціональні вимоги до системи та забезпечити ефективну взаємодію між її компонентами. Рисунок 2.1 демонструє діаграму варіантів використання багатокористувацької мережевої гри на Unity.

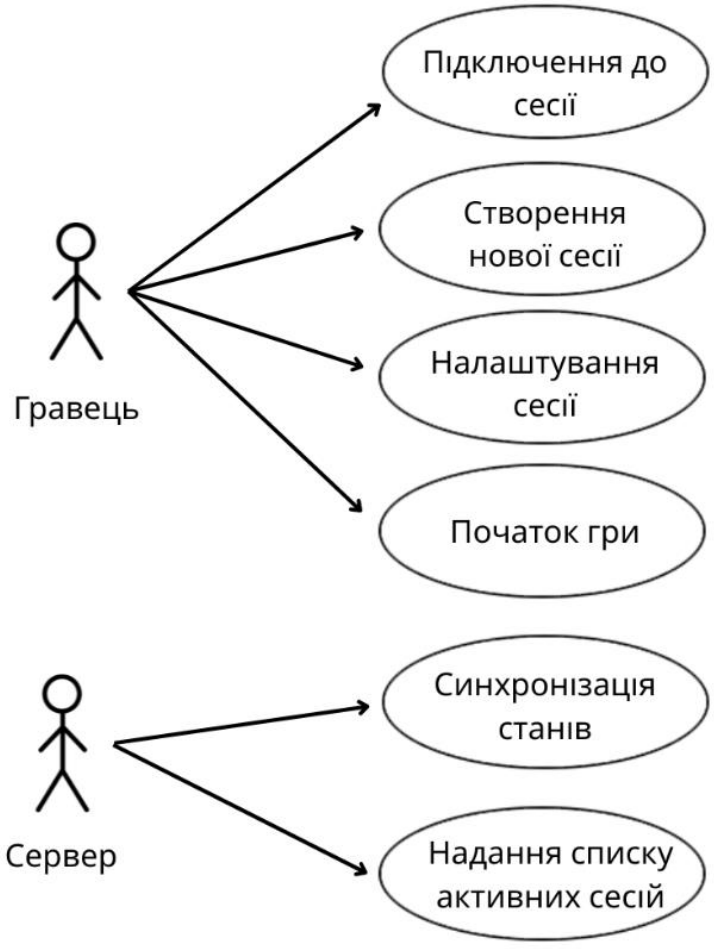

Рисунок 2.1 – Діаграма варіантів використання

Наступним кроком буде розробка специфікацій використання.

2.1.2 Специфікації варіантів використання мережевої багатокористувацької гри на Unity

Назва: Підключення до сесії

Актори: Гравець

Передумови:

1) Гравець має підключення до мережі.
2) Існують доступні ігрові сесії для приєднання.

Основний потік подій:

1) Гравець вписує назву ігрової сесії.
2) Гравець вибирає опцію приєднання до сесії.
3) Система перевіряє чи є така кімната.
4) Система перевіряє наявність вільних місць у сесії.
5) У разі наявності місця, гравець приєднується до сесії.
6) Система ініціалізує стан гри для нового гравця та синхронізує його з іншими учасниками.

Альтернативні потоки:

- Якщо сесія заповнена, система повідомляє гравцю та пропонує вибрати іншу сесію або створити нову.

Післяумови:

- Гравець успішно приєднаний до обраної сесії та готовий до участі в грі.

Назва: Створення нової сесії

Актори: Гравець

Передумови:

1) Гравець має підключення до мережі.
2) Немає схожої по назві ігрової сесії.

Основний потік подій:

1) Гравець вписує назву ігрової сесії.
2) Гравець вибирає опцію створення ігрової сесії.

3) Система перевіряє чи є така кімната.
4) Система створює ігрову кімнату.
5) Система надає гравцю статус хоста.

Альтернативні потоки:

- Якщо назва такої кімнати вже існує, гравець отримає попередження в інтерфейсі.

Післяумови:

- Гравець успішно створює ігрову сесію.

Назва: Налаштування сесії

Актори: Гравець

Передумови:

1) Наявність підключення до мережі.

Основний потік подій:

1) Гравець вписує назву сесії.
2) Гравець вибирає опцію створення сесії.
3) Система налаштовує сесію під потреби гравця.

Післяумови:

- Гравець отримує налаштовану сесію, яка є готовою для гри.

Назва: Початок гри

Актори: Гравець

Передумови:

1) Гравець має статус хоста і підключений до сесії з одним і більше гравцями.

Основний потік подій:

1) Гравець вибирає опцію початку гри.
2) Система починає анімацію Fade in на екранах гравців.
3) Система починає завантаження ігрової сцени на пристроях гравців.

4) Система закінчує завантаження ігрової сцени.
5) Система підключає гравців на ігрову сцену.

Альтернативні потоки:

- Якщо хост залишиться один в кімнаті, гра не зможе початись.

Післяумови:

- Гравці успішно підключаються на ігрову сцену.

Назва: Синхронізація станів

Актори: Сервер

Передумови:

1) Наявність підключення до мережі.

Основний потік подій:

1) Оновлення локальних даних у гравця.
2) Система розпізнає зміни і відправляє оновлені дані на сервер.
3) Сервер фіксує зміни і розсилає їх до всіх підключених гравців.
4) Система приймає нові зміни іншого гравця.

Післяумови:

- Дані успішно синхронізовані.

Назва: Надання списку активних сесій

Актори: Сервер

Передумови:

2) Наявність підключення до мережі.

Основний потік подій:

4) Система запрошує список активних сесій з певними фільтрами.
5) Сервер отримує запит, фільтрує активні сесії згідно налаштувань та надає системі дані.
6) Система приймає список.

Післяумови:

- Система отримує відфільтрований список активних сесій.

2.2 Розробка архітектури мережевої багатокористувацької гри на Unity

Архітектура програми – це головна конструкція, скелет на якій тримаються механіки, елементи програми. Гарна архітектура – ключ до легкої масштабованості, розвитку і високоефективності програми. Без неї великі проєкти закрились би в перші роки після відкриття і ми б не побачили таких гігантів як Google, Microsoft і т.д., сервісами яких ми так любимо користуватись.

В іграх так само потрібно продумувати і використовувати розумну, сильну архітектуру. Особливо в багатокористувацьких іграх, де окрім інтерфейсу, керування, базових механік гри, потрібно мати взаємодію з сервером і стабільну роботу мережевих компонентів.

Дана гра буде мати такі архітектурні принципи:

- DI (Dependency Injection) – шаблон проєктування програмного забезпечення, що передбачає надання зовнішньої залежності програмному компоненту, використовуючи «інверсію керування» (англ. Inversion of control, IoC) для розв'язання (отримання) залежностей [6].
- SOLID – це п'ять основних принципів об'єктно–орієнтованого програмування: принцип єдиної відповідальності, принцип «відкритий–закритий», принцип підстановки Ліскова, принцип розділення інтерфейсів та принцип інверсії залежностей.
- Реактивне програмування – це парадигма програмування, побудована на потоках даних і змін. Це означає, що у мовах програмування має бути можливість легко виразити статичні чи динамічні потоки даних, а реалізована модель виконання буде автоматично розсилати зміни через потік даних [7].

- MVP – шаблон проєктування, похідний від MVC, що відділяє візуальне відображення (інтерфейс) та поведінку обробки подій у різні класи, а саме: Модель (Model), Представлення (*View*), Пред'явник (*Presenter*).

Для великих і сильних проєктів використовують більше серйозних паттернів, але для гри середнього рівня вистачить і цього перечислення. У цьому перечисленні прописані правила, як ввести архітектуру, але для DI і реактивного програмування нам потрібні додаткові бібліотеки і фреймворки, щоб виконувати ці правила.

Для DI ми візьмемо безкоштовний opensource фреймворк Zenject. Він пропонує велику кількість корисних методів, можливостей по керуванню залежностями в проєкті. Потребує невеликих налаштувань в проєкті. Zenject буде сам впроваджувати залежності для коду, який їх потребує.

Під реактивне програмування була вибрана opensource бібліотека UniRx, яка є достатньо сильним і розвиненим інструментом. Вона дозволить відстежувати зміни даних і робити потрібні дії при оновлені. Корисна бібліотека, коли потрібно зробити швидку і багаторівневу фільтрацію даних.

Додаткові головні бібліотеки для роботи:

- UniTask – opensource бібліотека, яка допоможе оптимізувати асинхронне програмування C# і Unity. Також надає можливість створення паралельної задачі для розвантаження гри в пікових моментах.
- DoTween – бібліотека яка спеціалізується на створенні анімацій, відкладених викликів з коду, без анімування об’єктів через редактор.
- Addressables – Unity бібліотека, яка оптимізує використання пам’яті виконавчого пристрою. Вона надає можливість завантажувати в пам’ять потрібні на даний момент текстури, рисунки, дані для гри і вивантажувати коли вони не потрібні. Без неї, при запуску гри всі дані завантажуються в пам’ять одразу.

2.2.1 Розробка архітектурного ядра

Архітектурне ядро – це фундамент всієї архітектури гри. Ядро включає в себе головні сервіси, класи, функції, що керують основними процесами проєкту/гри та є опорою для внутрішнього ядра проєкту/гри.

Можна виділити такі головні сервіси інфраструктури:

- AssetsLoader – клас, який завантажує в пам’ять дані, текстури, модельки. І є фасадом/обгорткою бібліотеки Addressables описаною вище в розділі 2.2.
- ObjectsFactory – клас, який створює з потрібними налаштуваннями об’єкти на сцені окремо від мережі (якщо не потрібна синхронізація конкретно для цього об’єкта).
- Network **(**NetworkConnector, NetworkObjectProvider, NetworkSceneLoader, т.д) – мережеві класи.
- Zenject Bindings – класи, які впроваджують залежності, потреби іншим класам в грі.
- GameDataSaver – сервіс для збереження даних гравця. В даній грі цей сервіс зберігає нікнейм гравця, який той вводить при заході в гру.
- Application Input – система вводу гравця. Динамічно підтримує ввід гравця, з телефона, або комп’ютера.
- Update System – система кадрових оновлень. Більш оптимізована і зручна, ніж стандартна система від Unity.

Нижче, на рисунку 2.2, можна побачити дерево залежностей і реалізацію сервісів ObjectsFactory і AssetsLoader. Якщо потрібно створити об’єкти на сцені – використовуємо ObjectsFactory, якщо завантажити в пам’ять текстурку – AssetsLoader. На рисунку 2.2 видно, що AssetsLoader є обгорткою з додатковою логікою над базовою бібліотекою Unity – Addressables. AssetsLoader реалізує інтерфейс IAssetsLoader, який є в залежностях ObjectsFactory. Zenject (фреймворк залежностей) автоматично надасть класу ObjectsFactory його залежності

(IAssetsLoader, DIContainer) при створенні. DIContainer дозволяє фабриці об’єктів при створені нових об’єктів надавати їм залежності, без прямої участі від високорівневих Zenject класів.

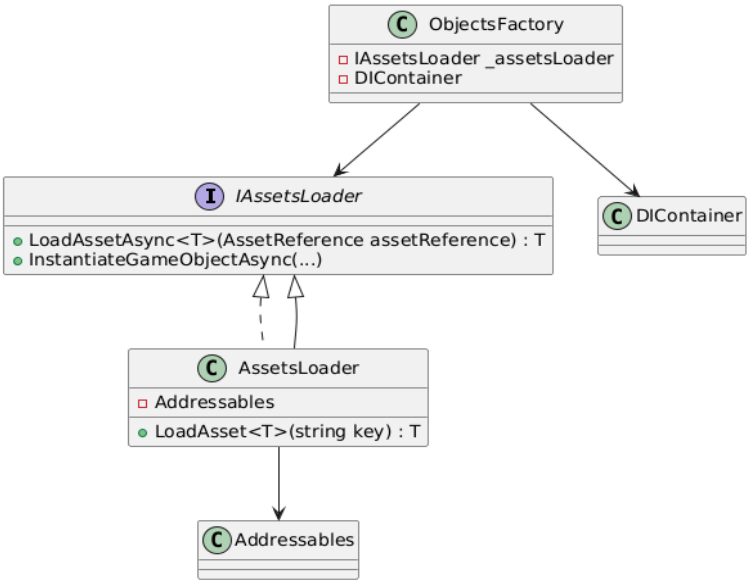

Рисунок 2.2 – Ієрархічне дерево сервісу Objects Factory

Далі розберемо сервіс надання вводу гравця – ApplicationInputProvider, переглянемо рисунок 2.3.

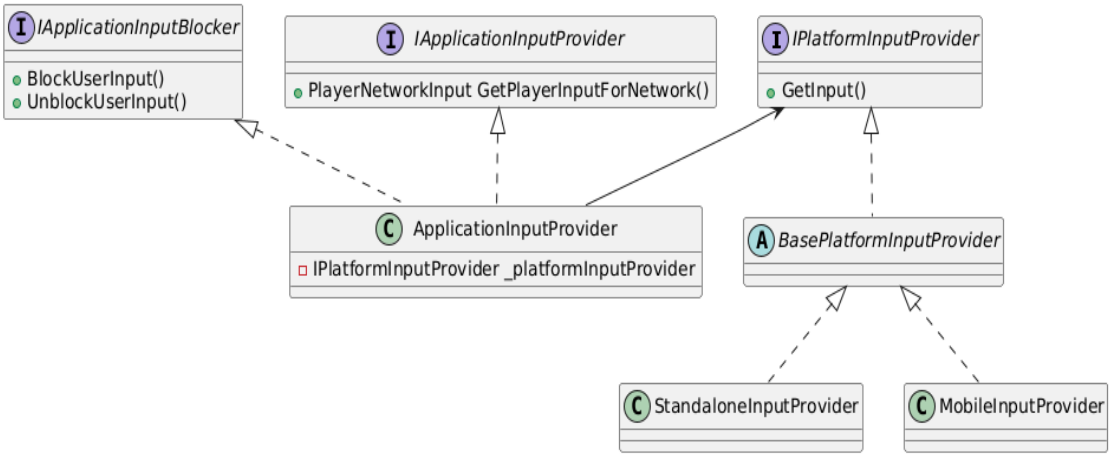

Рисунок 2.3 – Ієрархічне дерево сервісу ApplicationInputProvider

На рисунку 2.3 можна побачити сервіс ApplicationInputProvider, який має залежність на IPlatformInputProvider – інтерфейс класу любої платформи (телефон,

комп’ютер і т.д), що дає можливість тестувати гру на комп’ютері. І в той час мати керування через телефон з однієї реалізації. Реалізація класів StandaloneInputProvider і MobileInputProvider підтверджує факт легкості можливих змін і розширення цієї системи.

Перейдемо до мережевих головних сервісів на рисунку 2.4.

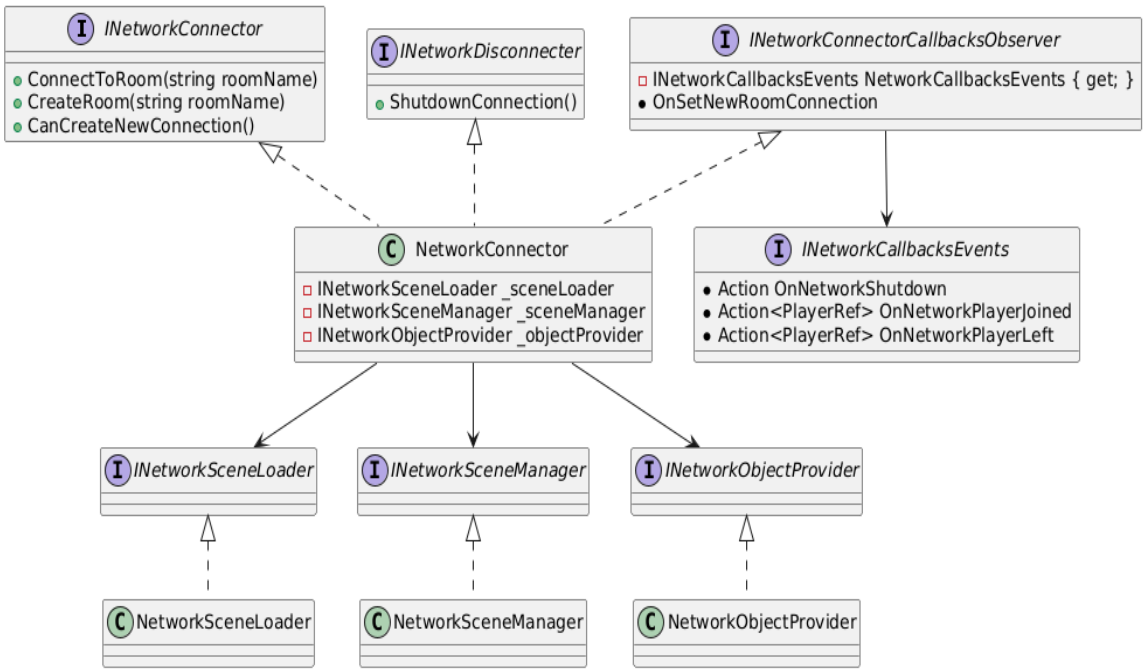

Рисунок 2.4 – Ієрархічне дерево мережевих сервісів підключення

На даному рисунку показані залежності класу NetworkConnector, який може створити підключення, підключитись до лобі, відключитись. Він передає підключенню мережеві класи завантаження сцен (NetworkSceneLoader + NetworkSceneManager), завантажувач мережевих об’єктів (NetworkObjectProvider). Рисунок не має повну павутину залежностей із-за обмежень в розмірі, але показує основні компоненти мережевої оболонки.

2.2.2 Розробка архітектури ігрової головної сцени

В розділі 2.2.1 було описане головне архітектурне ядро, яке дає змогу: зайти в лобі, створити лобі, підключитись до гри, отримати ввід гравця, створити об’єкт

на сцені. У цьому розділі буде описана архітектура ігрової сцени для взаємодії з гравцями та міні-іграми. Також є лобі сцена, але вона майже повторює архітектуру головної ігрової сцени, тому розглядати її не будемо.

Розглянемо компоненти підключення гравців і налаштування початку міні-гри на рисунку 2.5.

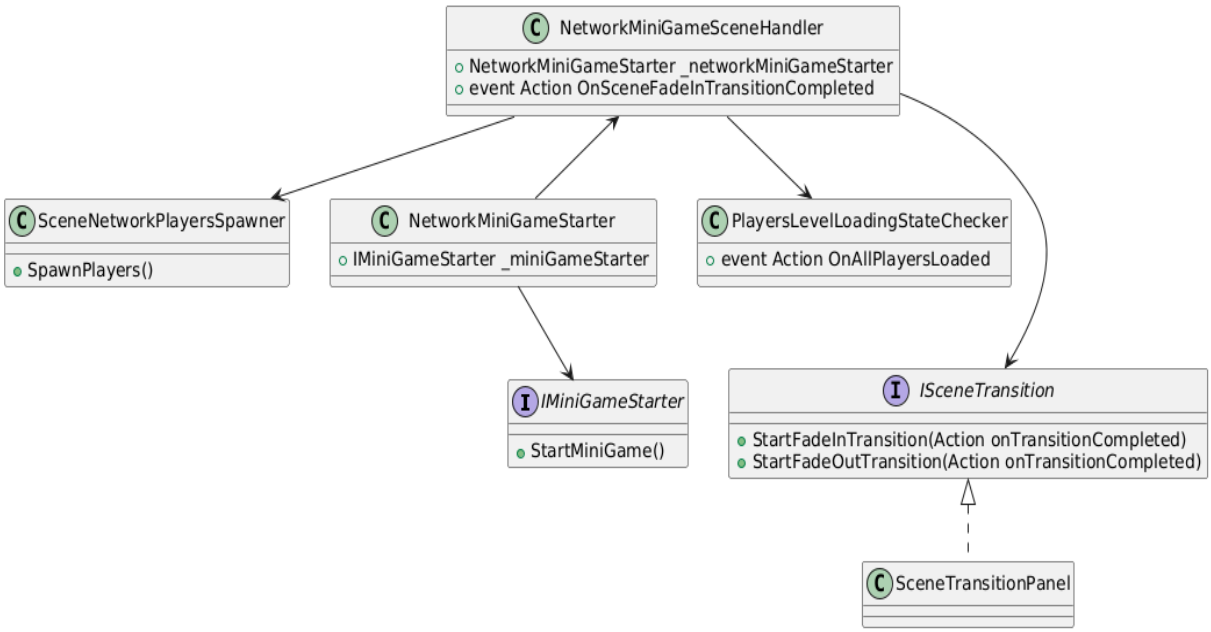

Рисунок 2.5 – Ієрархічне дерево мережевого обробника міні-гри

На цьому рисунку є головний мережевий обробник ігрової сцени – NetworkMiniGameHandler. Він чекає команди від PlayersLevelLoadingStateChecker, як тільки всі гравці завантажаться, обробник запустить анімацію початку гри через SceneTransitionPanel, створить гравців за допомогою SceneNetworkPlayersSpawner і викликає у міні-гри початок із зворотнім відліком. Таким чином, можна створювати окремо міні-ігри не змінюючи основну архітектуру головної сцени, що відповідає принципам SOLID. Компоненти міні-ігор просто реалізують інтерфейс IMiniGameStarter і підставляють любу механіку.

2.3 Проєкт мережевої багатокористувацької гри на Unity

В ході проєктування мережевої багатокористувацької гри на Unity було розроблено ескізний, технічний та робочий проєкти.

2.3.1 Ескізний проєкт мережевої багатокористувацької гри на Unity

Створення ескізного проєкту дозволить зрозуміти з чого почати та які існують основні цілі для досягнення мети проєкту. В таблиці 2.1 наведені дані цілі.

Таблиця 2.1 – Ескізні мережеві цілі проєкту.

Назва	Опис
Створення сесії	Гравець може створити сесію із унікальним кодом
Приєднання до сесії	Гравець вводить назву і приєднується до існуючої сесії
Початок гри	Хост може запустити гру, коли готові гравці
Налаштування гри	Хост може налаштовувати базово гру, після чого система доповнює ці налаштування.
Синхронізація даних	Всі мережеві дані гри синхронізуються коректно між сервером та гравцями.
Відображення лідерборду	Показ рахунку всіх гравців після закінчення раунду

В таблиці 2.2 можна побачити всі сцени, які потрібні для зручної та повноцінної роботи гри.

Таблиця 2.2 – Ігрові сцени проєкту

Назва сцени	Опис
Bootstrap	Сцена передзавантаження гри
Головне меню	Сцена де гравець може почати гру
Лобі-очікування	Сцена де гравці очікують на початок міні-гри
Ігрова сцена міні-ігор	Сцена міні-гри

Таблиця 2.3 показує, які характеристики потрібно назначити ігровій кімнаті/сесії для надійного функціонування геймпплею.

Таблиця 2.3 – Налаштування ігрової сесії

Налаштування	Опис
Кількість гравців	До 4 гравців
Наявність паролю	Є пароль
Можливість підключитись під час гри	Немає
Чи передається хост, якщо він покине гру	Ні

Місця, де в грі буде інтерфейс:

- Головне меню.
- Панелька рахунку гравців, яка є в лобі очікування та на міні-іграх.
- Панелька налаштувань.
- Інтерфейсні елементи під час гри в міні-гру.

Діаграма діяльності для варіанту використання «Створити ігрову сесію» представлена на рисунку 2.6.

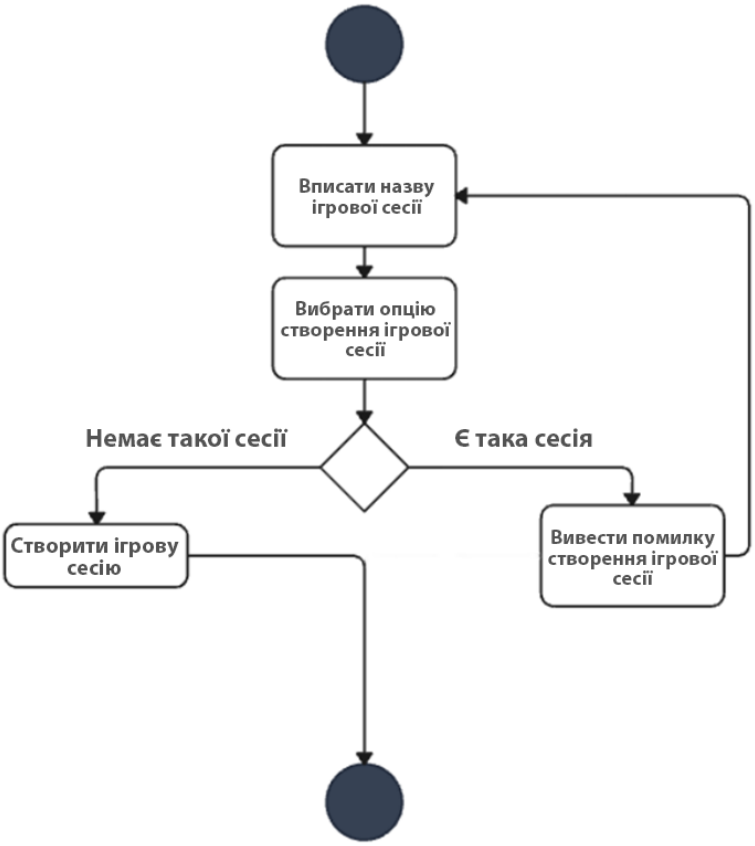

Рисунок 2.6 – Діаграма діяльності для варіанту використання «Створити ігрову сесію»

Діаграма діяльності для варіанту використання «Почати гру» представлена на рисунку 2.7.

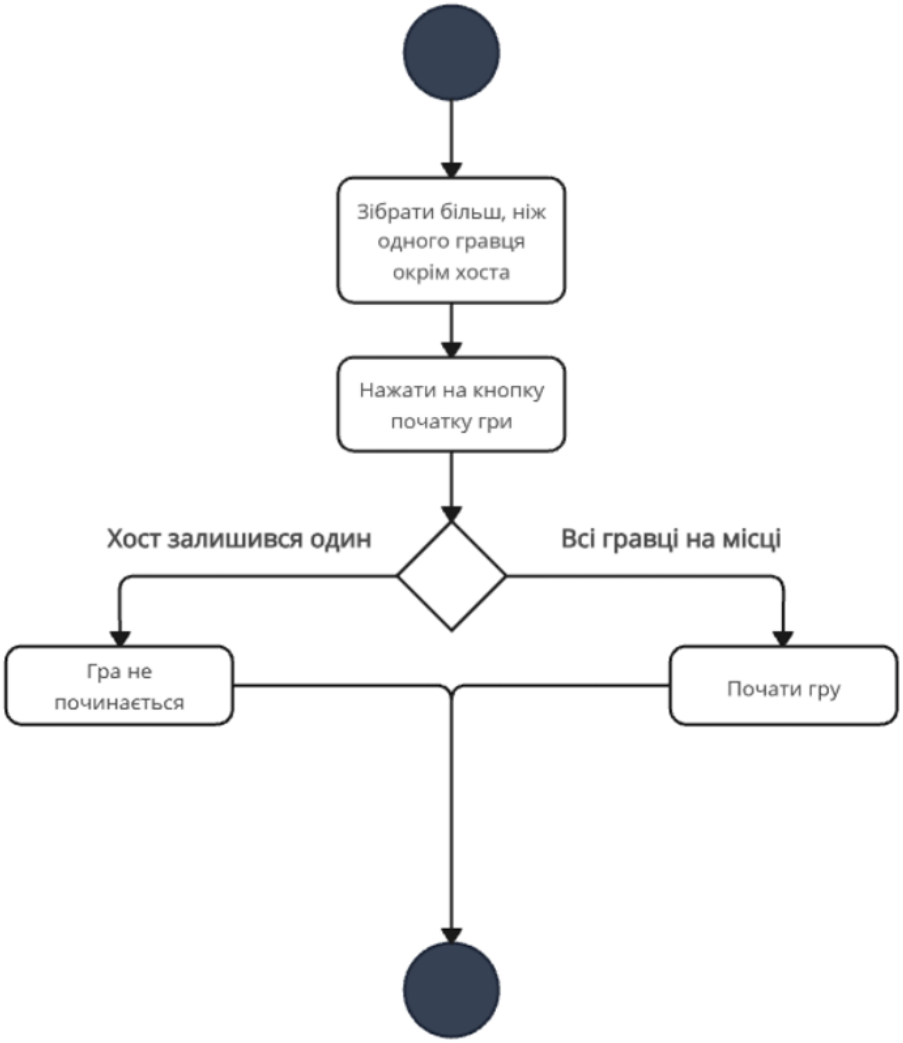

Рисунок 2.7 – Діаграма діяльності для варіанту використання «Почати гру»

Наступним етапом розробки ескізного проєкту мережевої багатокористувацької гри на Unity являється розробка інтерфейсу. Нижче, на рисунках 2.8 – 2.9, представлені ескізи інтерфейсу.

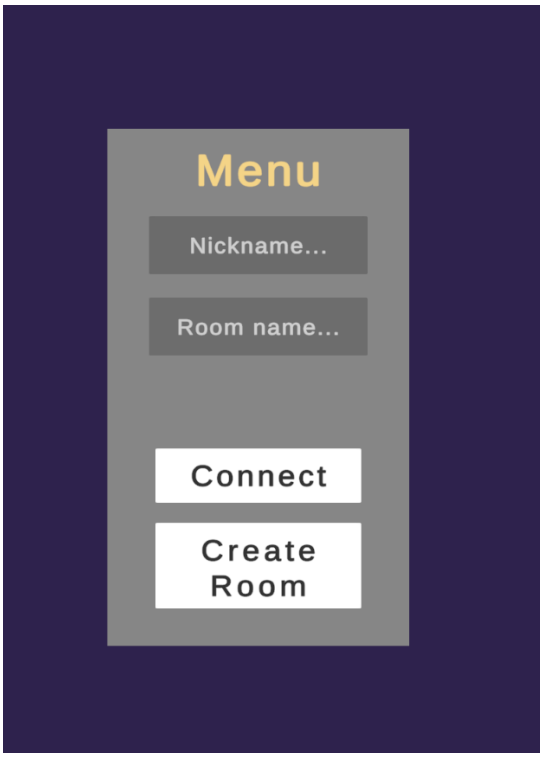

Рисунок 2.8 – Ескіз інтерфейсу ігрового меню

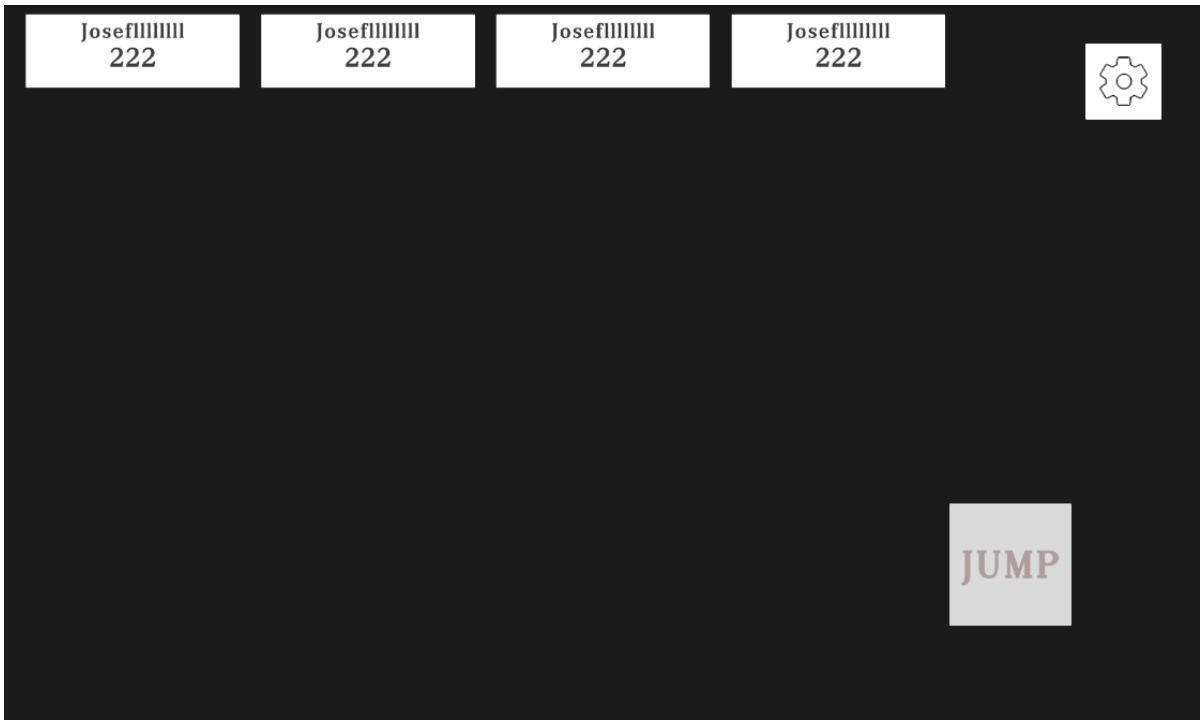

Рисунок 2.9 – Ескіз інтерфейсу під час гри

Наступним кроком буде створення технічного проєкту.

2.3.2 Технічний проєкт мережевої багатокористувацької гри на Unity

Провівши аналіз ескізного проєкту, було розроблено технічний проєкт багатокористувацької мережевої гри на Unity з використанням Photon Fusion. Діаграма основних класів і інтерфейсів представлена на рисунку 2.10.

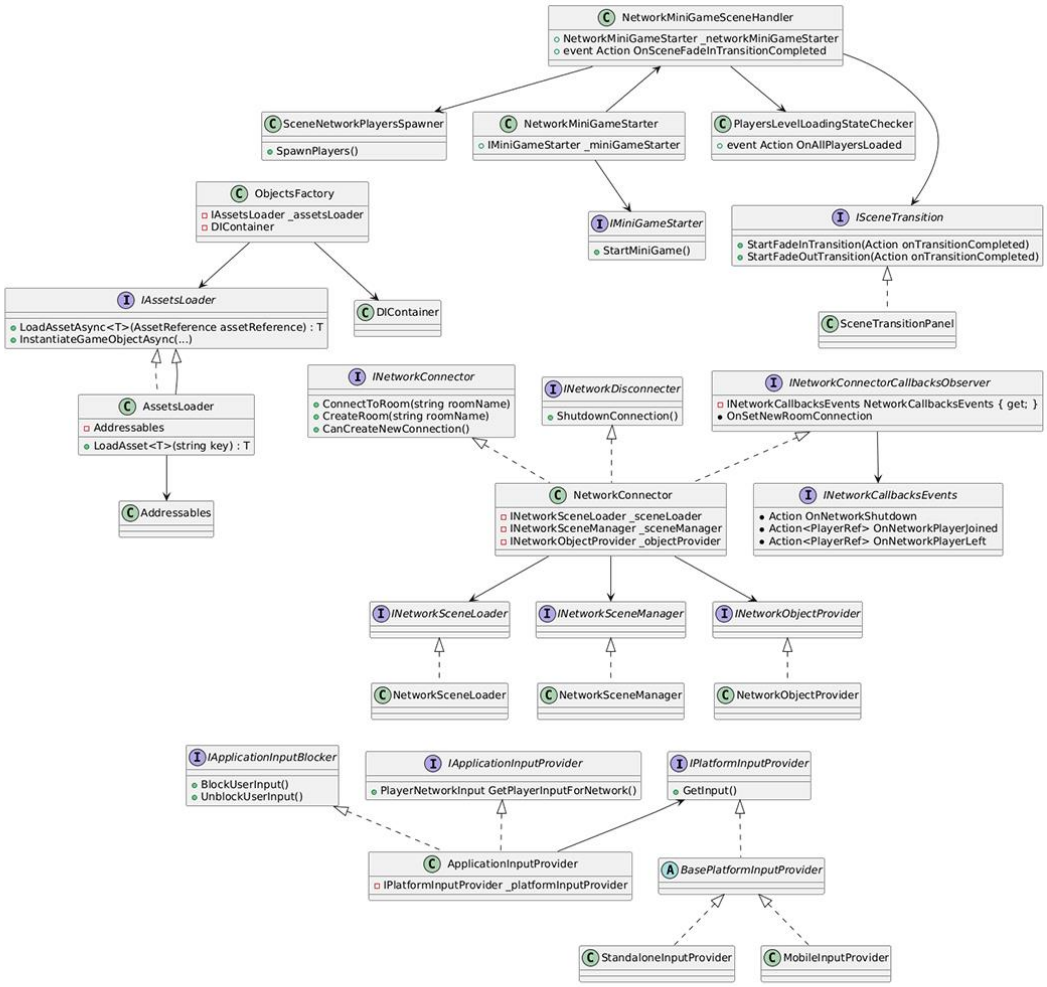

Рисунок 2.10 – Діаграма класів і інтерфейсів

Специфікації класів для багатокористувацької гри на Unity приведено нижче:

Клас: NetworkConnector

Назва: NetworkConnector

Короткий опис: Клас є сервісом створення/розриву підключень.

Поля:

- _sceneLoader – призначене для зберігання екземпляру мережевого класу завантаження сцен.
- _sceneManager – призначене для зберігання екземпляру мережевого класу керування сценами.
- _objectProvider – призначене для зберігання екземпляру мережевого класу керування об’єктами сцени.

Методи:

- ConnectToRoom – метод підключення до існуючої кімнати.
- CreateRoom – метод створення нової кімнати.
- ShutdownConnection – метод розриву підключення.
- CreateConnection – основний метод створення підключення, який використовується методами ConnectToRoom та CreateRoom.

Клас: NetworkMiniGameStarter

Назва: NetworkMiniGameStarter

Короткий опис: клас, який починає міні-гру.

Поля:

- applicationInputBlocker – сервіс блокування керування гравців, поки не почнеться гра.
- miniGameStarter – інтерфейс, який запускає міні-гру для будь-якого класу, який його реалізує.
- countdownTimer – таймер зворотнього виклику, який вказує скільки часу залишилось до початку міні-гри.

Методи:

- StartMiniGame – метод старту міні-гри.
- RPC_StartMiniGame – метод старту міні-гри у всіх мережевих гравців

- OnCountdownEnded – колбек метод, який викликається при закінченні таймеру і розблоковує керування для гравця.

Клас: NetworkSceneLoader

Назва: NetworkSceneLoader

Короткий опис: клас мережевого завантаження ігрової сцени.

Поля:

- _sceneSettingsChanger – сервіс зміни налаштувань сцени.

Методи:

- LoadScene – метод мережевого завантаження ігрової сцени для всіх гравців.
- GetScenePathKey – метод отримання шляху сцени по її ключу.

Клас: NetworkObjectProvider

Назва: NetworkObjectProvider

Короткий опис: клас, який перевизначає базову поведінку створення мережевих об’єктів.

Поля:

- _dependenciesInjector – екземпляр класу, який вводить залежності до інший класів.

Методи:

- AcquirePrefabInstance – перевизначений базовий метод, який додатково вводить новим мережевим об’єктам залежності до їх класів.

Клас: NetworkMiniGameStarter

Назва: NetworkMiniGameStarter

Короткий опис: клас, який починає міні-гру.

Поля:

- applicationInputBlocker – сервіс блокування керування гравців, поки не почнеться гра.
- miniGameStarter – інтерфейс, який запускає міні-гру для будь-якого класу, який його реалізує.
- countdownTimer – таймер зворотнього виклику, який вказує скільки часу залишилось до початку міні-гри.

Методи:

- StartMiniGame – метод старту міні-гри.
- RPC_StartMiniGame – метод старту міні-гри у всіх мережевих гравців
- OnCountdownEnded – колбек метод, який викликається при закінченні таймеру і розблоковує керування для гравця.

Клас: PlayerMovement

Назва: PlayerMovement

Короткий опис: клас, який відповідає за переміщення гравця.

Поля:

- _characterController – низькорівнева реалізація стану персонажа, яка йде з фреймворком PhotonFusion.
- _lastJumpTime – останній час стрибка гравця.
- _teleportPos – позиція для телепорту персонажа.

Методи:

- Teleport – метод телепортації на іншу позицію.
- Move – метод руху.
- Jump – метод стрибка.
- SetMoveVectorsRelativeToCamera – метод, який встановлює вектор руху персонажа відносно повороту камери.

Діаграма послідовності для прецеденту «Підключення до сесії» представлена на рисунку 2.11.

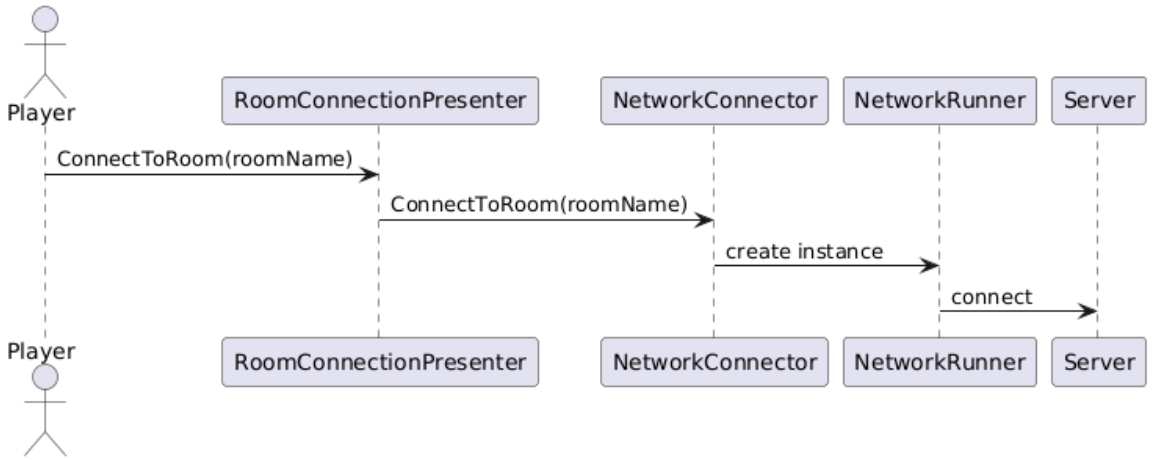

Рисунок 2.11 – Діаграма послідовності для прецеденту «Підключення до сесії»

Діаграма послідовності для прецеденту «Створення сесії» представлена на рисунку 2.12.

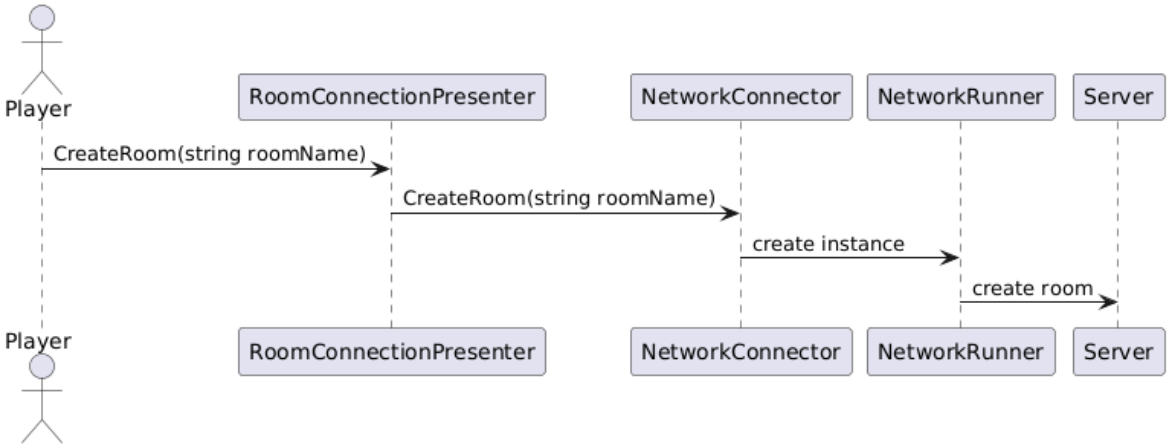

Рисунок 2.12 – Діаграма послідовності для прецеденту «Створення сесії»

Після створення ескізу і технічного проєкту, наступним етапом буде розробка проєкту.

2.3.3 Робочий проєкт мережевої багатокористувацької гри на Unity

Реалізація

Обґрунтування вибору засобів розробки багатокористувацької гри на Unity

Unity є одним із найпопулярніших ігрових двигунів завдяки своїй багатофункціональності, кросплатформеності та простоті у використанні. В реалізації багатокористувацьких ігор важливу роль відіграють спеціалізовані мережеві фреймворки, такі як Photon Fusion, що надають широкий спектр інструментів для створення стабільних, масштабованих та ефективних мережевих ігрових проєктів.

Використання Photon Fusion дозволяє вирішувати завдання синхронізації гравців у режимі реального часу, підтримки низької затримки та забезпечення стабільності з'єднання, що робить його перспективним інструментом для створення багатокористувацьких ігор. Таким чином, дослідження методів розробки багатокористувацьких ігор з використанням Unity та Photon Fusion є актуальним у контексті сучасних вимог до ігрових продуктів.

Підготовка

Почнемо з підготовки проєкту і його сервісів до повної розробки. Створюватись гра буде на 2022.3.54f1 версії Unity. Після створення проєкту, потрібно додати бібліотеки, фреймворки для роботи, а також власний базовий шаблон.

Імпортуємо такі бібліотеки і фреймворки:

- Zenject (реалізація DI принципу).
- UniTask (краща робота потоків і асинхронність).
- UniRx (реактивне програмування).

- Photon Fusion (мережевий фреймворк).
- ParallelSync (open source бібліотека, яка допоможе створювати тестові вікна редактора, що імітують мережевих гравців).

Власно написаний шаблон включає в себе:

- Базову ігрову сцену, яка буде слугувати шаблоном для інших сцен.
- Базову реалізацію прив'язок залежностей фреймворка Zenject.
- Реалізацію сервісу AssetsLoader (описаний в розділі 2.2.1, код в додатку Д (**1)**).
- Реалізацію сервісу ObjectsFactory (описаний в розділі 2.2.1, код в додатку Д (**2)**).
- Реалізацію сервісу ApplicationInputProvider (описаний в розділі 2.2.1, код в додатку Д (**3)**).
- Реалізацію сервісу GameDataSaver (описаний в розділі 2.2.1, код в додатку Д (**4)**).
- Реалізацію сервісу UpdateSystem (описаний в розділі 2.2.1).

Створимо сцену передзавантаження сцени меню підключення. Вона буде займатись завантаженням сцени меню підключення з пам'яті, щоб менше навантажувати телефон і не залишати сцену в пам'яті. В майбутньому, коли гравець покидатиме меню – його сцена буде вивантажуватись з пам'яті. Також, важливо зробити базові налаштування в самому проєкті і прив'язати ключ кабінету Photon до проєкту, щоб мережевий фреймворк Photon Fusion працював правильно.

Створення меню підключення

Меню підключення – це початкова сцена гри з інтерфейсом. Де гравець зможе:

- Вписати свій нікнейм (ім'я).
- Створити ігрову кімнату.
- Підключитись до існуючої кімнати.
- За потреби покинути кімнату, або вигнати учасників (якщо він є лідером кімнати).

- Змінити свій одяг (скін).
- Почати гру (якщо він є лідером кімнати).

Імпортуємо безкоштовні ассети модельок гравців і інтерфейсу. Створюємо 4 подіума для гравців. Далі робимо основну панельку меню, інтерфейс якої буде змінюватись в залежності від стану, але фон і рамка залишається такою самою.

На етап “Не підключений” додамо два поля. Один для вводу нікнейму, який зберігатиме строку нікнейму в GameDataSaver. В наступні заходи в гру гравця, поле вставлятиме нікнейм автоматично з збережених даних. Друге поле буде слугувати під назву ігрової сесії, або для підключення в існуючу, або для створення. Також додамо дві кнопки на “Підключитись” та “Створити кімнату”. Результат на рисунку 2.13.

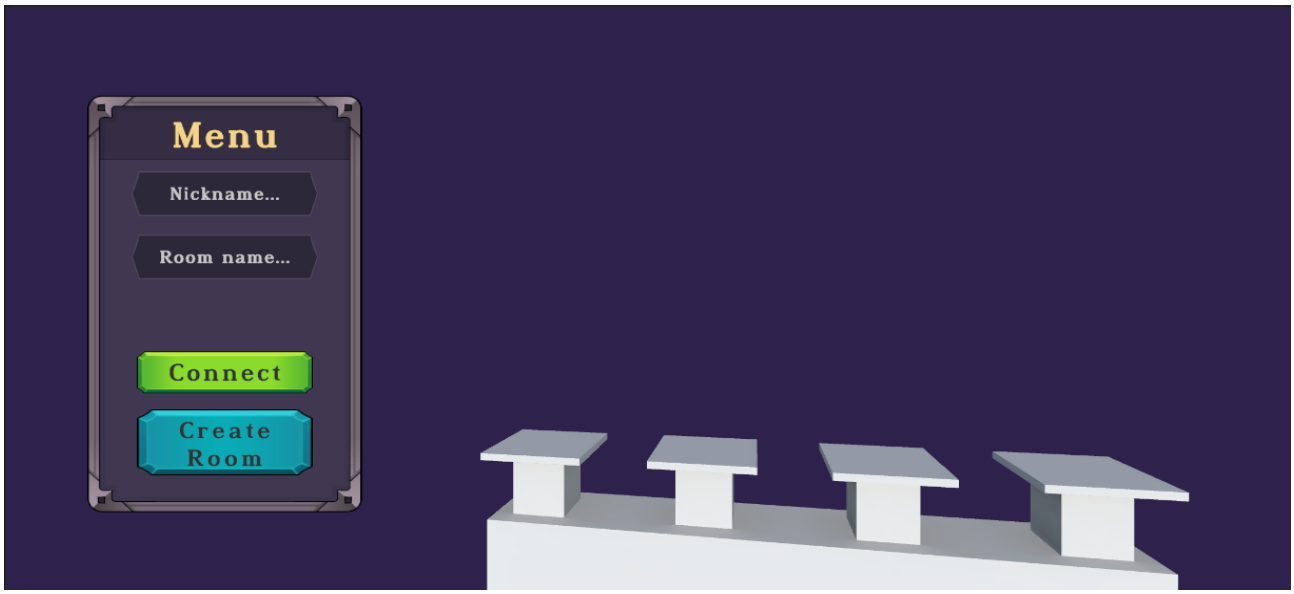

Рисунок 2.13 – Меню підключення, етап “Не підключений”

На етап “Підключений”, текст “Menu” змінемо на текст назви кімнати. Приберемо поля вводу нікнейму і назви кімнати. Натомість, додамо два тексти, перший – це кількість гравців в кімнаті в вигляді “Players n/4” (де n – кількість гравців), другий – нікнейм гравця. Також замінемо кнопки на “Почати гру” та “Покинути кімнату”. 4 п’ядестала, які ми поставили на сцені, мають точки

прив’язки для спавна гравців. Таким чином, кожен новий гравець спавниться на своєму п’ядесталі.

Нижче наведений код спавна гравців на п’ядесталах при підключенні до кімнати і деспавна – якщо хтось виходить з кімнати.

```
public void SpawnPlayer(NetworkRunner runner, PlayerRef playerRef)
{
    Pedestal pedestal =
_playersPodium.GetPedestalByNumber(runner.ActivePlayers.Count());
    NetworkObject player = runner.Spawn(_networkPodiumPlayerPrb,
pedestal.PlayerSpawnPos, pedestal.PlayerSpawnRotation, playerRef);
    _playersPodium.ReservePedestalForPlayer(pedestal.Number, player);
    _playersDict.Add(playerRef, player);
}

public void DespawnPlayer(NetworkRunner runner, PlayerRef playerRef)
{
    NetworkObject player = _playersDict[playerRef];
    _playersPodium.FreePedestalFromPlayer(playerRef);
    _playersDict.Remove(playerRef);
    runner.Despawn(player);
}
```

В методі SpawnPlayer ми отримуємо наступний вільний п’ядестал для гравця і спавнимо його, потім записуємо його в словник. У випадку, якщо будь-який гравець покидає кімнату, викликається DespawnPlayer, який видаляє об’єкт гравця і звільняє п’ядестал. Якщо всі слоти зайняті і кімната стає повністю забита, і наприклад, 2-й гравець покидає кімнату, тоді 3-й гравець займає позицію 2-го, а 4-й гравець позицію 3-го п’ядесталу. Тобто сортируються знову, заповнюючи пусті місця. У випадку виходу лідера лобі з кімнати, вся кімната повністю розпускається.

В інтерфейсі залишилось додати кнопки кіку (вигону) гравця під модельками. Та панельку зміни одягу (скіна) гравця, зміна якого приймається тільки на модельці гравця. Одяг синхронізується між гравцями, тому на всіх пристроях у гравців буде однаковий одяг, який вони вибрали. Додамо стрілочки і циклічну прокрутку, якщо

гравець дійде до кінця списку одягу, список почне давати одяг з самого початку. Нижче маленька частина коду зміни одягу персонажа:

```
private async UniTask<Transform> SpawnNewModelSkin(byte id)
{
    AssetReference skinRef = _playersSkinsStorage.GetSkinReferenceById(id);
    GameObject modelSkin = await _objectsFactory.Create(skinRef, new
Vector3(0,1000,0));
    Transform modelSkinTrm = modelSkin.transform;
    modelSkinTrm.parent = transform;
    modelSkinTrm.localPosition = Vector3.zero;
    modelSkinTrm.localRotation = Quaternion.identity;
    return modelSkinTrm;
}
```

На цьому фрагменті коду ми отримуємо посилання на ассет одягу. Далі через _objectsFactory (ObjectsFactory описаний в 2.2.1) передаємо посилання на ассет. В наступних рядках ми переміщуємо, повертаємо цей об’єкт одягу і фактично “одягаємо” нашого гравця в новий скін. Так як всередині ObjectsFactory є AssetLoader, то при створені нового скіна – старий скін очищається з пам’яті, при умові, що інші гравці не одягнули його. Зручно і ефективно.

Ось як виглядає фінальна картинка меню підключення, переглянемо рисунок 2.14:

Рисунок 2.14 – Меню підключення, етап “Підключений”

Наступним етапом, хотілось би додати трішки життя всій картинці. Додамо випадкові цікаві анімації до гравців, які кожні n секунд будуть програвати анімацію (щастя, гри на гітарі і т.д.). Для цього потрібно скачати безкоштовні анімації, додати їх в проєкт і налаштувати під модельки. В аніматорі Unity назначимо всі анімації і створюємо переходи з умовами, номерами цих анімацій. Далі потрібно створити скрипт з синхронізацією анімацій. Синхронізовані дані будуть виглядати так:

```
[Networked, OnChangedRender(nameof(OnIdleAnimNumberChanged))]
private int _idleAnimNumber { get; set; }

[Networked, OnChangedRender(nameof(OnEmotionReactionAnimNumberChanged))]
private int _emotionReactionAnimNumber { get; set; }
```

де _idleAnimNumber – це номер анімації в стані спокою. А _emotionReactionAnimNumber – номер анімації реакції. Коли по таймеру у гравця потрібно програвати анімацію, ставиться додаткове випадкове число на реакцію, яке синхронізується на сервері між усіма гравцями. Після чого і у інших гравців починає програватись анімація. Слово Networked в коді вище означає синхронізацію змінної, а OnChangedRender отримує метод, який викликається при зміні цієї змінної.

Останнім етапом треба підключити створення кімнат, підключення і взагалі мережеву частину до меню. Архітектура описаного мережевого підключення була прописана та представлена в розділі 2.2.1, рисунок 2.4. Повний код мережевого підключення гри можна побачити в додатку Д (**5**). Даний код буде використовуватись як головний сервіс мережевого підключення. Більше ні один з класів не повторює його роботу.

Також зв’яжемо механіку підключення до наших кнопок, тут використовуючи принцип MVP (описаний в розділі 2.2), ми створюємо скрипти для нашого інтерфейсу, розділяючи роботу з візуальною і логічною частиною. На прикладі коду нижче показана частина класу, яка відповідає за візуальне оновлення.

```
public void UpdatePlayersCountInRoom(int playersInRoom, int maxPlayersRoomSize)
{
    _playersCountInRoomText.text = $"Players: {playersInRoom}/{maxPlayersRoomSize}";
}

public void SetNewRoomSessionSettings(string roomName, string nickname, bool isHost)
{
    _startGameButton.gameObject.SetActive(isHost);

    _roomNameText.text = roomName;
    _nicknameText.text = nickname;
}

private void SubscribeButtons()
{
    _startGameButton.onClick.AddListener(_presenter.StartGame);
    _leaveRoomButton.onClick.AddListener(_presenter.LeaveRoom);
}
```

Як видно, методи класу RoomInfoPanelView оновлюють тільки візуальну частину. А метод SubscribeButtons підписує кнопки інтерфейсу методами логічної частини класу RoomInfoPanelPresenter, частина якого показана нижче.

```
public void StartGame()
{
    if (_networkRunner.SessionInfo.PlayerCount == 1)
    {
        _errorDisplay.Display("Need more than one player");
        return;
    }

    _networkRunner.SessionInfo.IsOpen = false;
    _networkSceneLoader.LoadScene(_networkRunner, _lobbySettingsData);
}

public void LeaveRoom()
{
    _networkRunner.Shutdown();
}

private void UpdatePlayersInSession()
{
    _view.UpdatePlayersCountInRoom(_networkRunner.ActivePlayers.Count(),
_networkRunner.SessionInfo.MaxPlayers);
}
```

Як видно, в методі LeaveRoom для підключень і відключень використовується клас NetworkConnector з додатку Д (**5**). Принцип MVP надає можливість легкого керування візуальної і логічної частин інтерфейсу.

Створення лобі очікування та ігрового персонажа

Сцена лобі очікування потрібна для підготовки усіх гравців до міні-гри і можливого відпочинку між міні-іграми.

Площа лобі повинна бути середньою по розмірам і поміщатись в камеру з FOV (Field Of View) значенням – 60. Щоб на телефонах персонажі не виглядали занадто маленькими і одночасно великими. Далі потрібно розставити дерева, кущі і траву, щоб вони наполовину закривали фон сцени. Результат можна побачити на рисунку 2.15.

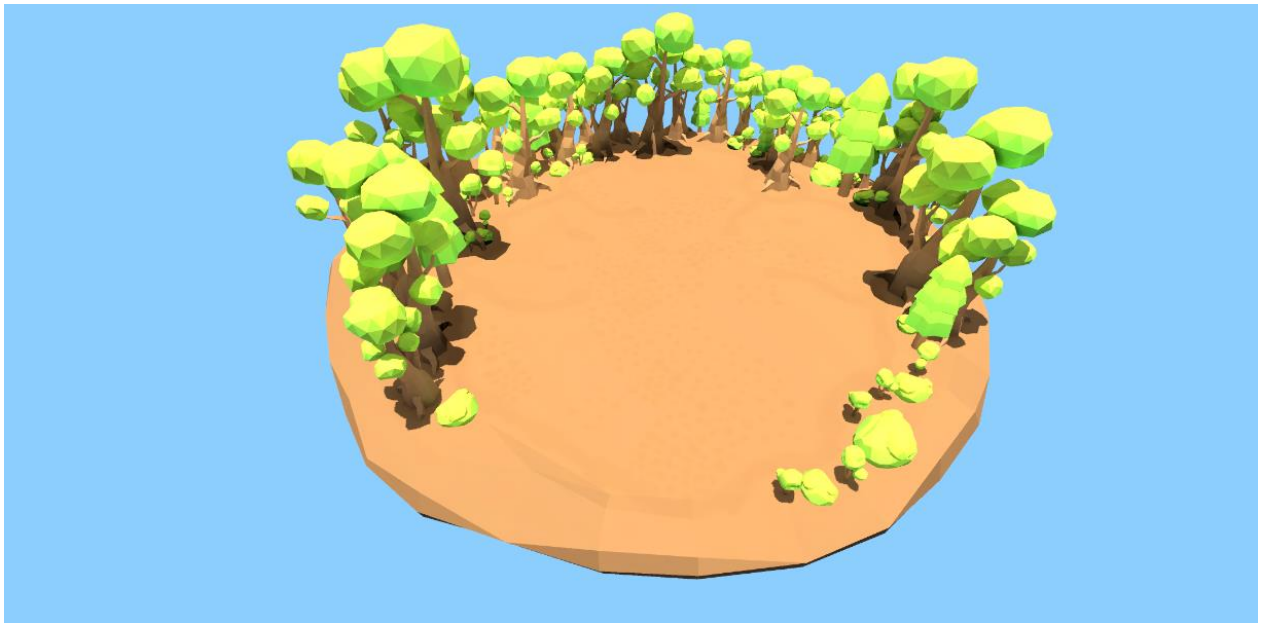

Рисунок 2.15 – Лобі очікування (проміжний етап)

Наступним етапом треба додати деталей до сцени. Камінці, паркан, міні-ринок. А також зону трігеру (портал) для переходу на міні-гру, стоячи в якому, всі гравці можуть почати наступну гру.

До речі, при освітлені гри в реальному часі, Unity буде сильно навантажувати центральний та графічний процесори пристрою. Тому, для оптимізації потрібно “запекти” освітлення сцени. Для цього у Unity є функція Lighting Baking. Зробивши такі налаштування запікання, як на рисунку 2.16, ми зможемо дешево рендерити сцену не переймаючись за навантаження освітлення на пристрій.

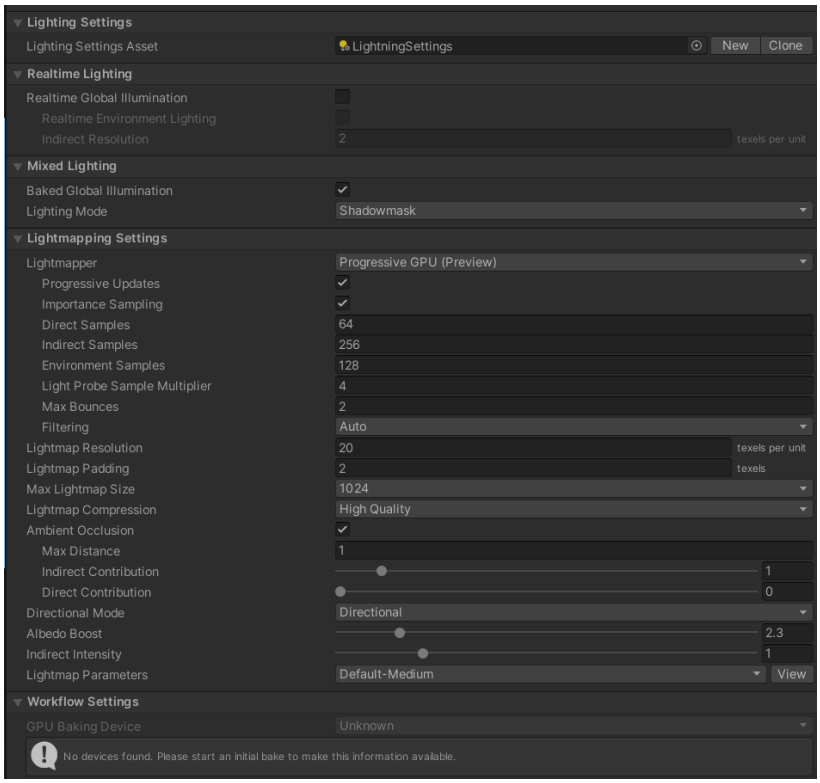

Рисунок 2.16 – Налаштування Light Baking

Кінцевий результат сцени на рисунку 2.17.

Рисунок 2.17 – Лобі очікування

В грі та лобі повинна бути можливість руху для гравців. Тому потрібно зробити об’єкт гравця та навісити на нього мережеві компоненти, включно з механікою руху. PlayerMovement (скрипт руху) складається з таких основних методів:

```
public override void FixedUpdateNetwork()
{
    if (_ifNeedTeleport)
    {
        _characterController.Teleport(_teleportPos, _teleportRotation);
        _ifNeedTeleport = false;
    }

    if (GetInput(out PlayerNetworkInput input))
    {
        Move(input.MovementVector);

        if (input.IsJumpPressed && Time.time - _lastJumpTime >= _jumpBlockTimeDelay &&
_characterController.Grounded)
        {
            Jump();
        }
    }
}

private void Move(Vector2 movementInputVector)
{
    Vector3 movementVector = (_forwardMoveVectorRelativeToCamera *
movementInputVector.y + _rightMoveVectorRelativeToCamera *
movementInputVector.x).normalized;
    _characterController.Move(movementVector);
}

private void Jump()
{
    _characterController.Jump();
    _playerAnimations.AnimateJump();
    _lastJumpTime = Time.time;
}
```

На цьому фрагменті коду є метод FixedUpdateNetwork, який представляє тік (мережевий кадр) і робить синхронізацію стану на сервері. Всередині нього компонент отримує ввід гравця і на його основі робить відповідні дії: рухає, якщо був рух (метод Move), або стрибає, якщо був стрибок (метод Jump).

Після налаштувань, основні компоненти і скрипти на гравці виглядають так, як на рисунку 2.18.

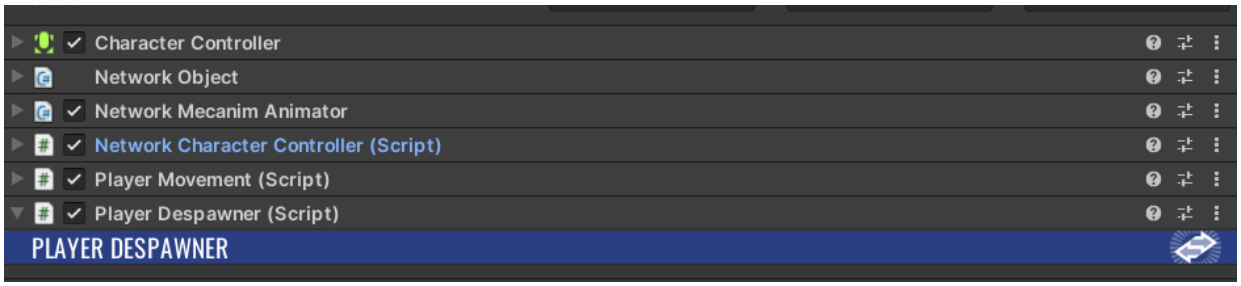

Рисунок 2.18 – Компоненти гравця

Вони означають:

- CharacterController – колайдер (оболонка) для взаємодії з фізичним світом.
- NetworkObject – мережевий компонент об’єкту, для синхронізації з сервером.
- NetworkMecanimAnimator – мережевий компонент для синхронізації анімацій.
- NetworkCharacterController – низькорівневий компонент переміщення персонажу.
- PlayerMovement – компонент керування персонажем, отримує ввід і передає дані NetworkCharacterController–у.
- PlayerDespawner – відключає мережевий об’єкт гравця при виході з кімнати.

Наступним етапом буде створення зони входу в портал, який переключить сцену лобі на випадкову міні-гру. Але тут була знайдена невелика проблема. Зазвичай, щоб зрозуміти в Unity чи є щось/хтось в зоні трігера, колайдера – використовується TriggerSystem, яка вказує коли об’єкт входить і виходить із зони. Але з мережевим керуванням ця система перестала працювати правильно. Вона декілька разів вказувала, що гравець виходить/входить із зони в зону. Але він стояв на місці. Було прийнято рішення замінити цю механіку на “динамічний тригер”, або OverlapSphere, з ним можна динамічно створювати тригер і визначати чи є гравці в зоні. Код в додатку Д (**6**). Коли всі гравці зайдуть в зону – вмикається зворотній

відлік на всіх пристроях і по його закінченню починається гра. Якщо хтось вийде з зони до закінчення таймеру, таймер скидається і зона буде чекати всіх гравців знову.

Процес розробки основних компонентів лобі та міні-ігор

У цьому розділі буде йти розробка основних мережевих і локальних компонентів гри, які використовуються в міні-іграх, а де-які також в сцені лобі очікування. Результатом компонування основної сцени будуть рисунок 2.19 і рисунок 2.20.

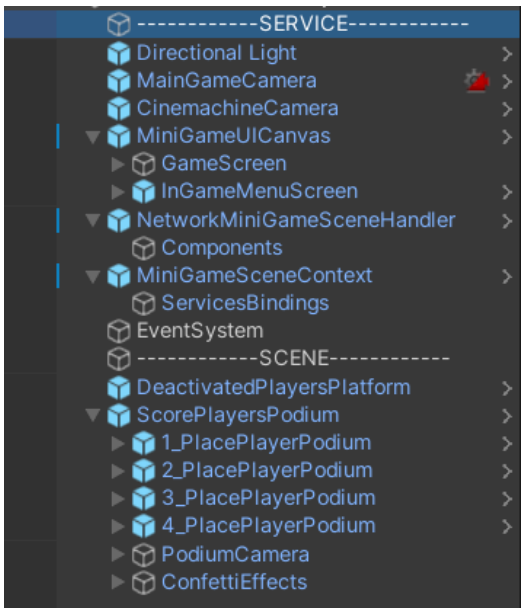

Рисунок 2.19 – Ієрархія основної сцени

Рисунок 2.20 – Готовий вид з камери на пусту сцену

Готові міні-ігри будуть створюватись під час виконання гри на основній сцені, тому на рисунку 2.20 тільки фоновий голубий колір і елементи інтерфейсу.

Камера

Камера для лобі очікування і міні-ігор буде використовуватись з однаковими налаштуваннями. Але, до цього не було сказано, як рухається камера та які є проблеми з керуванням персонажа по сцені.

Щоб камера слідувала за гравцем, мала активні точки спостерігання (границі від яких камера починає слідкувати за об'єктом) та ще сотні різних налаштувань, використовують Unity пакет – Cinemachine. Його потрібно встановити в редакторі, якщо не був встановлений автоматично. Він не є фреймворком, а просто доповненням Unity з сильним функціоналом до камер. Налаштування на компоненті виглядають приблизно так (скорочена версія), як на рисунку 2.21:

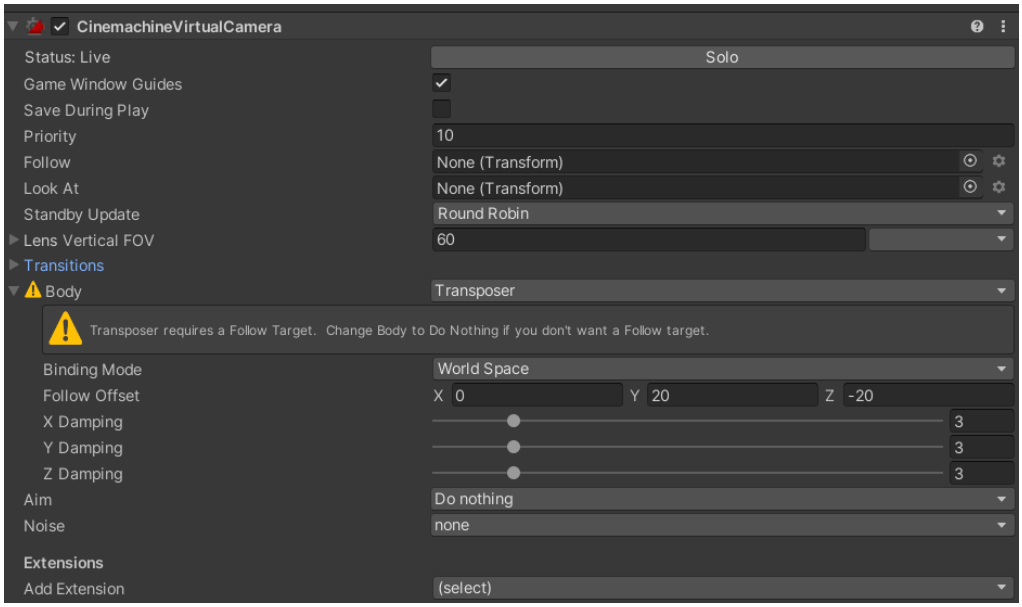

Рисунок 2.21 – Cinemachine компонент камери

На рисунку 2.21 зображені налаштування, які вказують, що потрібно слідкувати за гравцем в світових координатах, бути на відстані від гравця в координатах (x:0, y:20, z:–20), мати FOV (Field Of View) 60, з значенням плавності 3 по всім осям, без повороту в сторону об'єкту за яким слідкуємо, а тільки

переміщення. І ще з десятком дрібних налаштувань, які закриті в випадаючому списку.

Щодо проблеми, яка існує з камерою і керуванням гравця. При створені гравця, камера прив'язується до нього. Якщо керувати гравцем вперед-назад, то в залежності від повороту камери і як вона прив'яжеться, гравець, натискаючи вперед, може ходити назад чи вліво, або вправо. Тому, потрібно для вводу вектору руху мати множник на напрямки виду камери. Тобто, якщо камера повернута на x вісь від глобальної осі координат, персонаж повинен рухатись по x вектору при ходьбі вперед. Або якщо повернута на y вісь, то персонаж при ходьбі вперед йде по y вектору. Приклад коду нижче:

```
private void Move(Vector2 movementInputVector)
{
    Vector3 movementVector = (_forwardMoveVectorRelativeToCamera *
movementInputVector.y + _rightMoveVectorRelativeToCamera *
movementInputVector.x).normalized;
    _characterController.Move(movementVector);
}

private void SetMoveVectorsRelativeToCamera()
{
    Transform cameraTrm = Camera.main.transform;

    Vector3 forward = cameraTrm.forward;
    Vector3 right = cameraTrm.right;

    forward.y = 0;
    right.y = 0;

    _forwardMoveVectorRelativeToCamera = forward.normalized;
    _rightMoveVectorRelativeToCamera= right.normalized;
}
```

В даному фрагменті коду метод SetMoveVectorsRelativeToCamera встановлює передній і правий вектор погляду камери. Тому, при русі гравця ці значення потрібно перемножити.

Інтерфейс

Інтерфейс гри розбивається на дві частини: ігрову та меню. Ігрова – активна під час гри, меню – коли гравцю потрібні налаштування, або вийти з гри.

Ігрова частина має такі елементи:

- Панелька вводу. Джойстик та кнопка стрибка (для телефона).
- Панелька очків кожного гравця.
- Таймери. Звичайний та час гри в міні-гру.
- Кнопка активації меню.

В меню входить:

- Основна панелька інтерфейсу.
- Кнопка “Повернутись в гру”.
- Кнопка “Покинути гру”.

Панелька вводу передає дані в ApplicationInputProvider, якщо поточна платформа є телефоном. Таймери потрібні для підготовки гравців до старту міні гри, розуміння коли закінчиться міні гра, в кінці перед поверненням в лобі.

Панелька очків на початку гри активує міні частини для кожного гравця, де вписані їх імена і кількість очків, набраних в поточний момент. Виглядає це так:

```
private void ActivatePlayerScores(Func<PlayerRef, int> scoreReturnerStorageMethod =
null)
{
    IEnumerable<PlayerRef> activePlayersInSession =
_activeRunnerInfoProvider.ActiveSessionPlayers;

    int playersPanelsIndex = 0;

    foreach (PlayerRef playerRef in activePlayersInSession)
    {
        int playerScore = scoreReturnerStorageMethod != null ?
scoreReturnerStorageMethod(playerRef) : 0;
        ActivatePlayerScoreMiniPanel(playerRef, playerScore, playersPanelsIndex);
        playersPanelsIndex++;
    }
}

private void ActivatePlayerScoreMiniPanel(PlayerRef playerRef, int score, int
activatedIndex)
{
    PlayerScorePanelPresenter panel = _playersScores[activatedIndex];
    string playerNickname = _playersNicknamesStorage.GetPlayerNickname(playerRef);
    panel.Activate(playerNickname, score);
    _playersPanelsDict.Add(playerRef, panel);
}
```

Метод ActivatePlayerScores отримує список активних гравців в кімнаті і проходиться циклом по методу ActivatePlayerScoreMiniPanel, передаючи

індентифікатори гравців, їх кількість очків на даний момент і т.д. Кінцевий результат такий, як на рисунку 2.22:

Рисунок 2.22 – Панелька поточних очків гравців

В меню не так багато кодової логіки. Кнопка “Повернутись в гру” просто закриває панельку з деякими діями. А кнопка “Покинути гру” викликає метод ShutdownConnection у вже обговореному класі NetworkConnector (розділ 2.2.1).

Zenject. Бінди залежностей

Вже було описано, що у цьому проєкті використовується фреймворк Zenject і він є основною складовою для принципу DI та впровадження залежностей. Але не було показано як це робиться, і сказано, як це працює. Подивимось на рисунок 2.23.

```
Frequently called  0+7 usages  Danil Drozdov  More...
public override void InstallBindings()
{
    // Infrastructure Services
    BindApplicationInputProvider();
    BindDependenciesInjector();
    BindActiveSceneComparer();
    BindGameDataSaver();
    BindObjectsFactory();
    BindAssetsLoader();
    BindObjectsPool();

    // Network
    BindNetworkSceneLoader();
    BindNetworkConnector();
    BindNetworkRunnerInfoProvider();

    // Core
    BindPlayersScoresStorage();
    BindPlayersNicknamesStorage();
    BindPlayersSkinsStorage();
    BindGameLevelsRandomizer();

    // UI
    CreateAndBindSystemUIDependencies();
}
```

Рисунок 2.23 – Методи створення залежностей в контейнері Zenject

На рисунку 2.23 показані методи з проєктного головного класу залежностей. Ще є сценові класи створення залежностей, там список трішки менший. Кожен метод розбивається приблизно на таку схему:

```
Container
    .Bind<IType>()
    .To<Type>()
    .AsSingle()
    .NonLazy();
```

На коді вище створюється залежність в Zenject. Код впроваджує в так званий Container залежність, де ми призначаємо типу (частіше інтерфейсу) IType об’єкт типу Type.

Читається так:

- Bind<IType>() – призначити типу IType.
- To<Type>() – новостворений об’єкт типу Type.
- AsSingle() – як єдиний створений екземпляр.
- NonLazy() – негайно.

І так в кожному методі. Десь можна додавати додаткові функції, етапи для більш гнучкого створення залежностей в контейнері. Щоб отримати з іншого кінця коду залежність, нам потрібно поставити атрибут [Inject] біля поля, методу, якщо це клас MonoBehaviour (від Unity). Або в конструктор Zenject, який сам надає залежності без атрибута, якщо це звичайний клас. Приклад впровадження залежностей в метод класу на рисунку 2.24:

```
[Inject]
Danil Drozdov
private void Construct(IPlayersScoresStorage playersScoresStorage, IPlayersMiniGameScoresStorage miniGameScoresStorage,
    IPlayersNicknamesStorage playersNicknamesStorage, INetworkActiveRunnerInfoProvider activeRunnerInfoProvider,
    INetworkConnectorCallbacksObserver networkConnectorCallbacksObserver, IActiveSceneComparer activeSceneComparer)
{
    _playersScoresStorage = playersScoresStorage;
    _playersNicknamesStorage = playersNicknamesStorage;
    _activeRunnerInfoProvider = activeRunnerInfoProvider;
    _activeSceneComparer = activeSceneComparer;
    _networkCallbacksEvents = networkConnectorCallbacksObserver.NetworkCallbacksEvents;
}
```

Рисунок 2.24 – Inject метод

Як видно з рисунку вище, Zenject автоматично надає інтерфейси, які нам потрібні, і далі ми можемо працювати з ними як заманеться. І не потрібно прокидувати посилання на ці класи по всьому коду, достатньо лише закинути їх в Zenject в одному місці.

Мережева логіка

Надійна архітектура мережі – ключ до успіху багатокористувацького проєкту. Розіб'ємо процес заходу гравців на рівень, їх синхронізацію аж до старту самої міні-гри. Всіма етапами керує клас NetworkSceneHandler, частково або повністю. Розділимо передзавантаження гравців на етапи:

- Очікування підключення всіх гравців до сцени.
- Спавн гравців.
- Ввімкнення переходу від екрана завантаження до гри.
- Ініціалізація гравця.

За очікування підключення гравців до сцени відповідає клас PlayersLevelLoadingStateChecker. Клас і опис можна побачити в додатку Д (**7**). Спавн об'єктів гравців відбувається одразу, як тільки хост заходить на сцену, навіть не чекаючи всіх гравців. Коли всі зайдуть, їм автоматично Photon видає керування над об'єктами. Що виглядає приблизно так:

```
public void SpawnPlayers()
{
    IEnumerable<PlayerRef> players = Runner.ActivePlayers;

    foreach (PlayerRef playerRef in players)
    {
        NetworkObject spawnedPlayer = SpawnPlayer(playerRef);
        _activePlayersObjectsList.Add(spawnedPlayer);
    }
}

private NetworkObject SpawnPlayer(PlayerRef playerRef)
{
    Transform spawnPoint = GetRandomPlayerSpawnPoint().transform;
    return Runner.Spawn(_playerPrb, spawnPoint.position, spawnPoint.rotation, playerRef);
}
```

де SpawnPlayers проходиться циклом і викликає основний метод спавну гравця – SpawnPlayer.

Після того, як NetworkSceneHandler отримує інформацію від івенту, що всі гравці завантажились, він викликає ініціалізацію компонентів та вмикає перехід з екрану завантаження до гри. В основному, в ініціалізації шукається об’єкт гравця, яким керують, і до нього прив’язується камера, щоб вона постійно дивились на локального гравця, а не на інших.

З екраном переходу дещо цікавіше. Перехід використовує Fade In/Out ефект. Тобто, круговий ефект появи/зникнення сцени. Такого функціоналу в Unity немає. Тому будемо писати шейдер (код для відеопроцесора). Шейдер на мові GLSL – OpenGL Shading Language. Також, щоб ним керувати, потрібен один клас на C#. Шейдер з описом знаходиться в додатку Д (**8**). Зовнішній код, який керує шейдером, є клас SceneTransitionPanel, який оновлює коли потрібно дані шейдера цим кодом:

```
private Material _imageMat;
private Action _onTransitionCompleted;
private float _targetCircleRadius;
private float _circleRadius;
private bool _ifNeedDeactivatePanel;

private static readonly int _radius = Shader.PropertyToID("_Radius");

public void OnSystemUpdate(float deltaTime)
{
   _circleRadius = Mathf.MoveTowards(_circleRadius, _targetCircleRadius, deltaTime * _transitionSpeed);
   UpdateMaterialData();

   if (_circleRadius == _targetCircleRadius)
   {
      if (_ifNeedDeactivatePanel)
      {
         _image.gameObject.SetActive(false);
         _ifNeedDeactivatePanel = false;
      }

      _onTransitionCompleted?.Invoke();
      this.StopUpdate();
   }
}

private void UpdateMaterialData()
{
   _imageMat.SetFloat(_radius, _circleRadius);
}
```

OnSystemUpdate метод (з бібліотеки UpdateSystems) оновлює кожен кадр дані шейдера і робить анімації появлення/зникнення сцени. Дивитись рисунок 2.25.

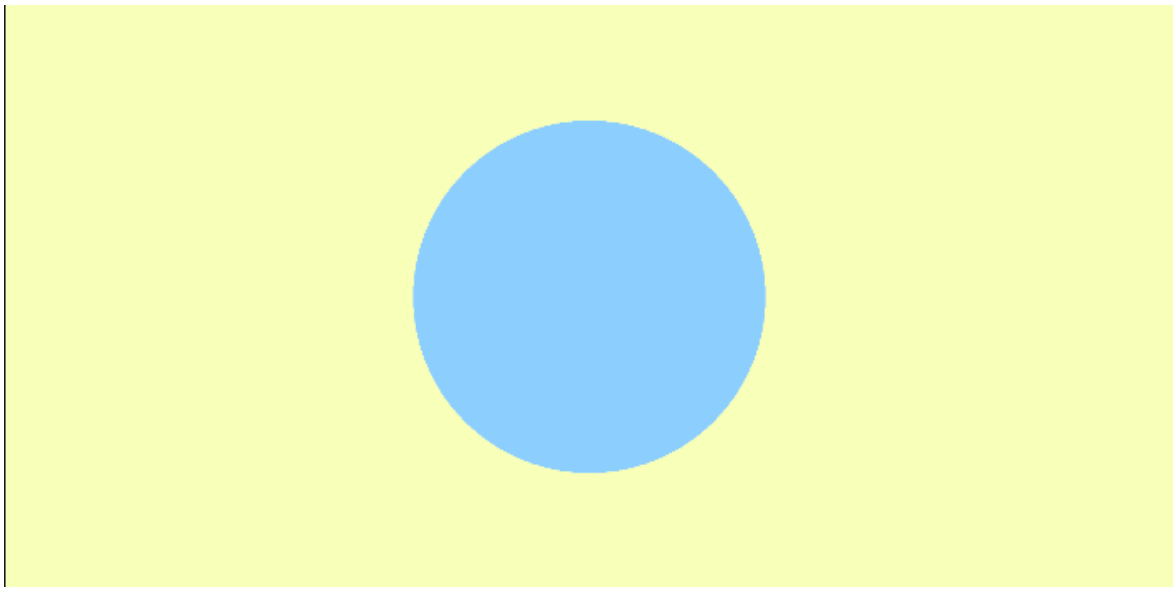

Рисунок 2.25 – Екран переходу на сцені

Платформа вимкнених персонажів та подіум результатів

Після того, як гравець програє в міні-грі, його персонаж стає не потрібним на сцені. Щоб запобігти не потрібним мережевим і фізичним проблемам, знадобиться платформа деактивованих персонажів. І замість того, щоб вимикати їх, ми будемо телепортувати неактивних гравців на цю платформу. Але тут все не так просто. Через те, що позиція персонажів у нас синхронізується з сервером, просто перемістити персонажа, вказавши вектор позиції, у нас не вийде. Для цього, нам потрібно зберегти нову позицію гравця на майбутній синхронізований тік. Код виглядає так:

```
{
    if (_ifNeedTeleport)
    {
        _characterController.Teleport(_teleportPos, _teleportRotation);
        _ifNeedTeleport = false;
    }
}

public void Teleport(Vector3 teleportPos, Quaternion? rotation = null)
{
    _ifNeedTeleport = true;
    _teleportPos = teleportPos;
    _characterController.Velocity = Vector3.zero;
    _teleportRotation = rotation ?? Quaternion.identity;
}
```

При потребі викликаємо на персонажі метод Teleport, який зберігає позицію і поворот нових даних. Так як Photon сам переписує дані мережевих об’єктів, то вписуємо в метод FixedUpdateNetwork телепорт на нову позицію.

Результати гри – один з найголовніших методів заохочення гравців. В даному проєкті після гри або міні-гри гравці з’являються на ступінчастому подіумі з номерами зайнятих місць і набраних очків. Поява гравців на подіумі супроводжується ефектами, такими як конфетті. Частина коду виглядає так:

```
public void PlacePlayerOnScorePedestal(PlayerMovement playerMovement, int playerPlace)
{
   ScorePlayerPedestal scorePedestal = _playersPodiumsByNumberPlacesDict[playerPlace];
   playerMovement.Teleport(scorePedestal.PlayerSpawnPoint.position + _yOffset,
scorePedestal.PlayerSpawnPoint.rotation);
}

public void SetPlayerScoreToPedestal(int playerPlace, int playerScore)
{
   _playersPodiumsByNumberPlacesDict[playerPlace].SetScore(playerScore);
}
```

де метод PlacePlayerOnScorePedestal телепортує гравця на його п’ядестал, використовуючи механіку описану вище на платформі вимкнених персонажів. Метод SetPlayerScoreToPedestal назначає п’ядесталу кількість набраних очків персонажем.

Головний код, який сортує гравців і розставляє їх по п'єдесталам, представлений нижче:

```
protected virtual void SetUpPodiumAndPlayers()
{
   ReadOnlyDictionary<PlayerRef, int> playersScores = GetPlayersScores();
   List<PlayerRef> sortedPlayersByScore = SortPlayersByScore(playersScores);
   SetScoresForPlayersPlaces(sortedPlayersByScore, playersScores);
   _scorePodiumCamera.StartAnimationLookTransferToPlayersScorePodium(OnScorePodiumLookTransferCompleted);

   if (Runner.IsServer)
   {
      PlacePlayersOnScorePodium(sortedPlayersByScore);
      PlayPlayersStateEmotions(sortedPlayersByScore);
   }
}
```

Цей метод сортує гравців по набраним очкам і розставляє їх, використовуючи код вище.

Кінцевий вигляд п'єдесталу результатів на рисунку 2.26.

Рисунок 2.26 – П’ядестал результатів

Розробка міні-ігор

Раніше в розділі розповідалось, як після заходу всіх гравців, підготовки сцени, запускається анімація входу на сцену ефектом Fade out. Далі вже йде процес запуску міні-гри. Після того, як анімація входу виконалась, вона повідомляє класу NetworkMiniGameStarter, що треба починати міні-гру у гравців. Виглядає це так:

```
[Rpc(RpcSources.StateAuthority, RpcTargets.All)]
private void RPC_StartMiniGame()
{
    _countdownTimer.StartCountdown(_countdownTimerTimesData.TimeToStartMiniGame, OnCountdownEnded);
}

private void OnCountdownEnded()
{
    if (Runner.IsServer)
    {
        _miniGameStarter.StartMiniGame();
    }

    _applicationInputBlocker.UnblockUserInput();
}
```

У цьому коді хост викликає у всіх метод RPC_StartMiniGame, що запускає зворотній таймер початку міні-гри у всіх гравців з відкладеним методом OnCountdownEnded. Коли таймер закінчився, OnCountdownEnded викликається.

Цей метод для всіх розблоковує керування, що дає змогу гравцям рухатись, а також починає міні-гру у хоста. Тобто, міні-гра хоста буде керувати своїми копіями у інших гравців. Підмічу, що це все запускається по інтерфейсу IMiniGameStarter. Тобто, щоб створити нову міні-гру, достатньо створити новий клас який реалізує цей інтерфейс і надати його в потрібний клас, що відповідає одному з принципів SOLID – Open/Closed principle.

Потрібно поставити умови на створення міні-ігор. Так як в цій грі планується зіграти 2+ міні-гри за гру, а одна гра не повинна йти більше 10 хвилин, тоді одна міні-гра не може тривати більше 3хв. Також є потреба створити цікаві, невеликі, обмежені колайдерами сцени, щоб сфокусувати гравця в невеликій зоні та надати більш захоплюючого геймплею з противниками.

Японські лазери

Ідея міні-гри була в створені арени на японській карті. На міні-гру дається хвилина. Час від часу випускаються лазери. На початку найдовше очікування наступних лазерів і менший шанс запуску великої кількості лазерів підряд. Під кінець йдуть найважчі хвилі.

Для початку створюємо підлогу зовнішньої сцени за ареною. Буде вона розміром 12 на 16 плиток. Всередині залишаємо місце під арену розміром 6 на 6. Камера повинна дивитись паралельно найбільшій стороні ігрової сцени.

Наступним етапом буде додавання будівель на сцену. Будемо використовувати оптимізовані низькополігональні модельки. Розставляємо так, щоб будівлі закрили більшість фону камери, яка не використовується. Це надасть ефект повноти картинки. Результат на рисунку 2.27.

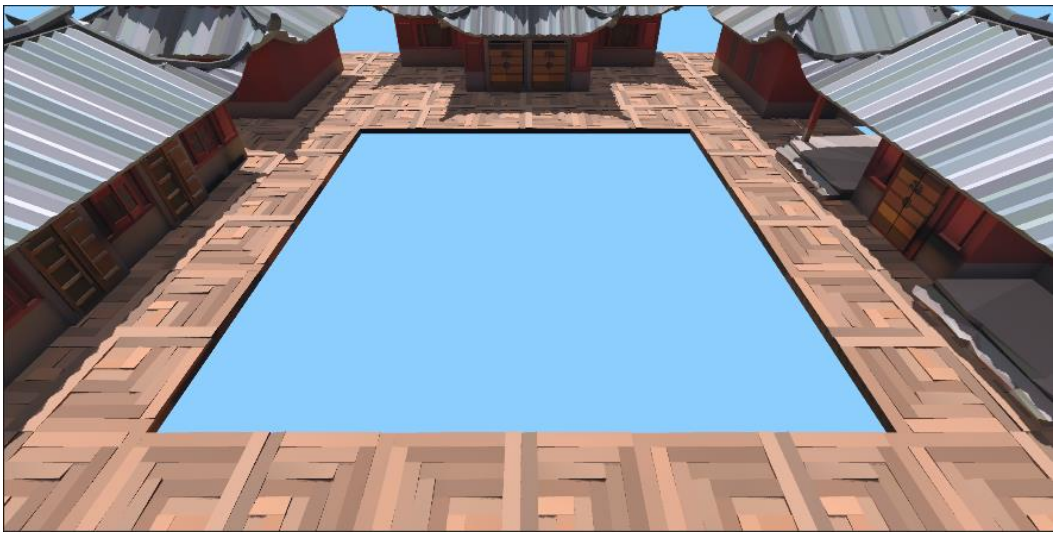

Рисунок 2.27 – Початковий етап створення японської сцени

Після того як базова сцена зроблена, настав час для декору. Будемо розставляти стіни, дерева, рослини та інше. Поставимо стіни на задній фон, щоб закрити великі пробіли між будинками. Розставимо дерева та рослини по периметру. З декору у нас є статуї, ящики та мішки з їжею, що символізують локацію як міні-ринок. Розставимо їх по локації і побільше перед камерою. На сакури поставимо ефект падаючих листків. Також не забудемо, що треба налаштувати джерело світла на теплі тони, щоб був ефект вечора. В кінці у нас вийде така цікава локація, дивитись рисунок 2.28.

Рисунок 2.28 – Майже закінчене зовнішнє кільце сцени

Так само як і в лобі очікування, на цій сцені для оптимізації будуть використовуватись запечені/е тіні/світло. Ця система надасть не тільки оптимізацію, а й кращу картинку, так як освітлення в реальному часі не використовує відбиття світла від поверхонь. З цією системою створяться світлові текстури, які будуть використовуватися об’єктами. Налаштування на рисунку 2.29.

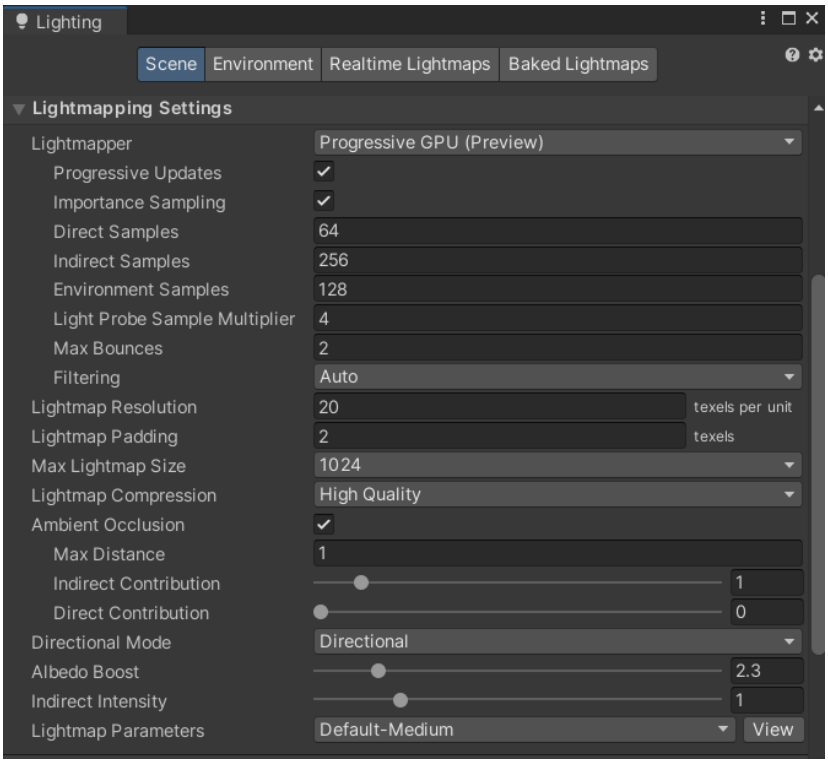

Рисунок 2.29 – Налаштування запікання світла на сцені

Останнім етапом роботи над модельками сцени є створення ігрової арени. Як і було сказано, арена буде 6 на 6 плиток. Кожна плитка повинна мати колайдер. Арена з кожної сторони оточена невеликою стіною з колайдером, яка буде слугувати фізичною стіною для стримування гравців, так і орієнтиром місця появи наступної хвилі лазерів. Фінальна картинка на рисунку 2.30 нижче. Сцена вийшла дуже насиченою і гарною. Кожен елемент стоїть на своїх місцях. В грі персонажів видно, модельки не зливаються одна з одною. Вечірня сцена надає приємних емоцій і відчуттів від гри.

Рисунок 2.30 – Фінальний вигляд японської сцени

Що ж, настав час оживити сцену і перетворити її на повноцінну міні-гру. Як і описувалось на початку розділу, для того щоб створити міні-гру, нам потрібен клас, успадкований від MiniGameStarter. Він надасть дозвіл головним мережевим класам почати міні-гру.

Почнемо з написання шансу створення різної кількості лазерів в кожній хвилі. Щоб вписати дані кожної хвилі, нам потрібна буде структура даних. Назвемо її AncientLasersCountThrowChance, вона матиме такий вигляд:

```
[Serializable]
public struct AncientLasersCountThrowChance
{
   [field: SerializeField, Unit(Units.Percent), MinValue(0), MaxValue(100)]
   public int ChanceThrowLasers { get; private set; }

   [field: SerializeField,  MinValue(1), MaxValue(3)]
   public int ThrowLasersCount{ get; private set; }
}
```

де ChanceThrowLasers – мінімальний шанс в випадковій рулетці, щоб отримати цю хвилю. ThrowLasersCount – кількість лазерів в цій хвилі. Ці налаштування в редакторі виглядають так, як на рисунку 2.31.

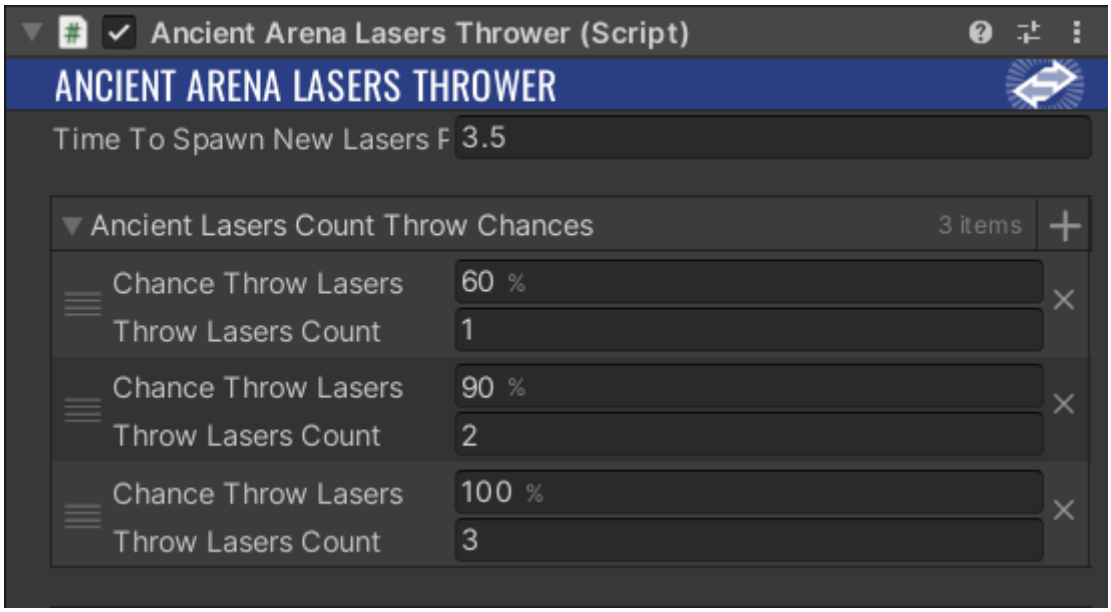

Рисунок 2.31 – Налаштування шансів лазерних хвиль

Коли наступає час пускання лазерної хвилі, нам знадобиться клас, який буде видавати нам випадкову кількість лазерів з потрібної нам калькуляції. Для цього створимо клас AncientLasersThrowCountRandomizer, основний метод якого буде таким:

```
public int GetRandomLasersCountToThrow()
{
    int randomPercentage = Random.Range(0, 100);

    for (int i = 0; i < _ancientLasersCountThrowChances.Count; i++)
    {
        AncientLasersCountThrowChance ancientLasersCountThrowChance = _ancientLasersCountThrowChances[i];
        int previousChance = i == 0 ? 0 : _ancientLasersCountThrowChances[i - 1].ChanceThrowLasers;

        if (randomPercentage > previousChance && randomPercentage <=
ancientLasersCountThrowChance.ChanceThrowLasers)
        {
            return ancientLasersCountThrowChance.ThrowLasersCount;
        }
    }

    return 0;
}
```

Метод GetRandomLasersCountToThrow робить перебірку шансів в відсортованому масиві. Рандомізує випадкове число від 0 до 100, що представить відсоток. І шукає цей відсоток між попереднім значенням шансу і наступним. Наприклад, на рисунку 2.27 є 2-й елемент на 90% і 3-й елемент на 100%, коли цикл

дійде до 3-ого елементу він перевірить наступну формулу. Якщо випадкове число (наприклад 96) більше ніж шанс попереднього (2-ого) елементу і менше за шанс поточного (3-ого) елементу – тоді видати кількість лазерів цієї хвилі. Що значить 90 <= 96 <= 100 видасть 3 лазери.

Все це буде використовувати головний скрипт міні-гри – AncientArenaLasersThrower, який дає команду випустить лазери. Його код представлений в додатку Д (**9**). Почитати детальніше про нього можна в додатку. Даний клас запускає лазери від бортів арени по таймеру, викликаючи активацію лазерів у класа AncientLasersArenaBorder.

AncientLasersArenaBorder має систему попередження гравця про напрямок активації хвилі лазерів. При активації сторона починає блимати червоним кольором. В коді нижче вмикається система попередження з зворотнім викликом.

```
public void ActivateLasers(int lasersCountToSpawn)
{
   _lasersCountToSpawnRemainder = lasersCountToSpawn;
   _warnActivator.ActivateWarnState(OnLasersSpawningWarnEnded);
}
```

Коли попередження закінчується – активується лазер. Якщо планується запускати ще лазер, тоді таймер стіни оновлюється і по закінченню випускається ще лазер, і так далі. Виглядає це наступним чином:

```
private void ActivateLaser()
{
   _lasersCountToSpawnRemainder--;
   _timeRemainderToSpawnNextLaser = _arenaSettings.SpawnTimeDelayBetweenMultiplyLasers;
   float borderWidth = Vector3.Distance(_leftCornerPoint.position, _rightCornerPoint.position);

   AncientLaser ancientLaser = _lasersSpawner.SpawnLaser();
   Vector3 startLaserPos = transform.position + transform.forward * _laserStartForwardOffset + _laserStartYOffset;
   ancientLaser.ActivateLaserInDirection(borderWidth, startLaserPos, transform.rotation,_oppositeWallTrm.position);

   if (_lasersCountToSpawnRemainder == 0)
   {
      this.StopUpdate();
   }
}
```

У цьому коді є спавнер лазерів, який на мережеву оболонку додає різні ефекти лазерів і таким чином не потрібно перестворювати об’єкти. Лазер телепортується до стіни і розвертається в потрібному напрямку.

```
public void ActivateLaserInDirection(float borderWidth, Vector3 startPoint, Quaternion forwardRotation, Vector3
endPoint)
{
   endPoint.y = startPoint.y;
   _movement.Teleport(startPoint, forwardRotation, () =>
   {
      Activate(borderWidth);
      _playersTrigger.SetXSize(borderWidth);
      _movement.MoveTo(endPoint, Deactivate);
   });
}

private void Activate(float borderWidth)
{
   _leftPlate.transform.localPosition = new Vector3(-borderWidth / 2, 0, 0);
   _rightPlate.transform.localPosition = new Vector3(borderWidth / 2, 0, 0);
   SwitchLaserComponents(true);

   if (Runner.IsServer)
   {
      RPC_ActivateSelfOnProxies(borderWidth);
   }
}
```

Так як лазер – це мережевий об’єкт з синхронізацією позиції, він використовує схожу механіку телепорта, як у персонажа, коли миттєве переміщення використовується через мережевий скрипт. По закінченню руху лазера до наступної стіни – він вимикається і повертається назад в пул лазерів.

Щоб знайти гравця, лазер кожного фізичного кадру вмикає трігер і шукає модельки гравців.

```
private void FindPlayers()
{
   int resultsCount = Physics.OverlapBoxNonAlloc(transform.position, _boxOverlapSize, _overlapResults,
_transform.rotation, _playersLayerMask);

   for (int i = 0; i < resultsCount; i++)
   {
      CatchPlayer(_overlapResults[i]);
   }
}
```

Якщо гравець попадається, тоді в нього віднімається життя. При отриманні певної кількості шкоди гравець програє і помирає.

Міні-гра вийшла дуже цікавою і динамічною. Результат чудовий, від карти до механік.

Лавова ущелина

Дана міні-гра вимагає від гравців швидкої реакції і великої удачі. Ідея полягає в створені плиток над лавою, частина яких час від часу будуть падати вниз. Сітчастий розмір ігрової зони буде 7 плиток в довжину, 5 в ширину. Камера повинна дивитись в сторону довшого напрямку. Якщо гравець не втримується і падає вниз – він програє. Якщо гравець/гравці досягають останніх плиток, які вже не впадуть – вони є переможцями. Потрібно розробити оптимізовану по освітленню карту, створити механіку плиток та накинути Post Processing ефекти.

Почнемо, як і в минулій міні-грі, з навколишнього середовища. Так як наша карта по плану має включати лаву, то нам знадобляться міні-вулкани, які допоможуть нам закрити не потрібний камерний фон. Використовуючи безкоштовні ассети, почнемо розставляти вулкани. Також налаштуємо власностворений матеріал, який назначимо на лаву вулканів і поставимо фон камери світло-помаранчевим, щоб відповідав локації. Початок буде мати вигляд, як на рисунку 2.32.

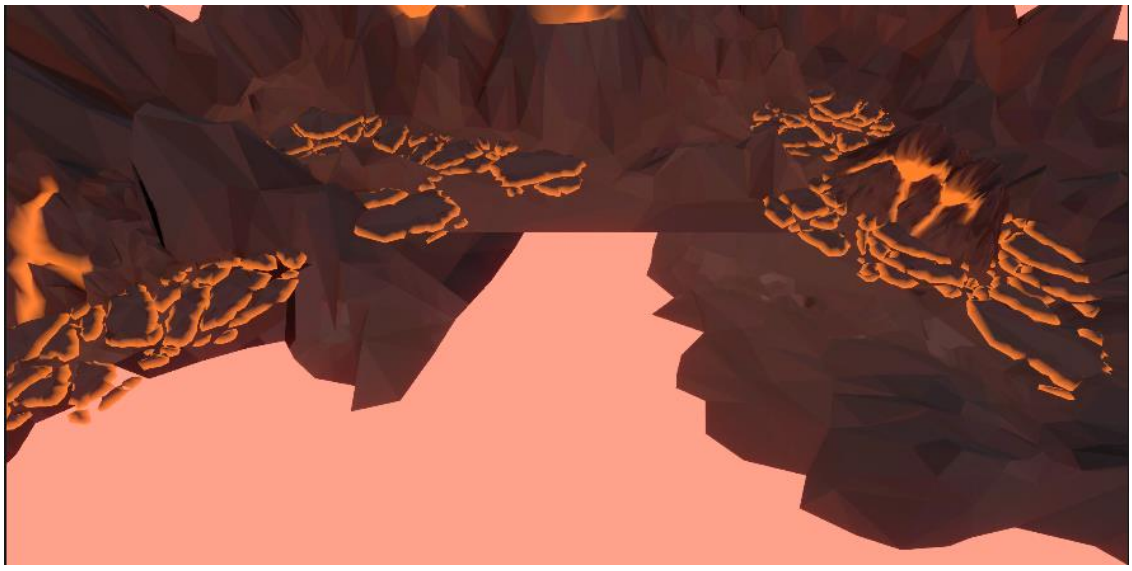

Рисунок 2.32 – Етап налаштування вулканів

Без лави поки не так цікаво. Створимо об’єкт панельки на сцені і розтягнемо її по всій локації під вулканами. Далі створимо новий матеріал на лаву, нам потрібен інший від лави, що на вулканах. Поставимо на новий матеріал Emissive Color, що дасть матеріалу ефект підсвічування при етапі запікання світла. Після чого додамо новий матеріал на панельку. Результат на рисунку 2.33.

Рисунок 2.33 – Етап налаштування лави

Картинка і локація виглядають доволі гарно. Пусте місце всередині залишимо для ігрової зони. Додамо на майбутнє об’єкт випромінення світла, зробимо його помаранчевого кольору і поставимо режим – Baked (для запікання світла). Він потрібен буде для ефекту відсвічування світла лави на наші плитки.

Також одразу налаштуємо Post Processing ефекти. Це ефекти, які накладаються зверху на картинку гри (ефект віньєтки, відсвічування, рипіння і т.д). Нам потрібен ефект Bloom, який продовжує світлові промені і робить більш реалістичний ефект світіння звичайних об’єктів. Налаштування цього ефекту на рисунку 2.34.

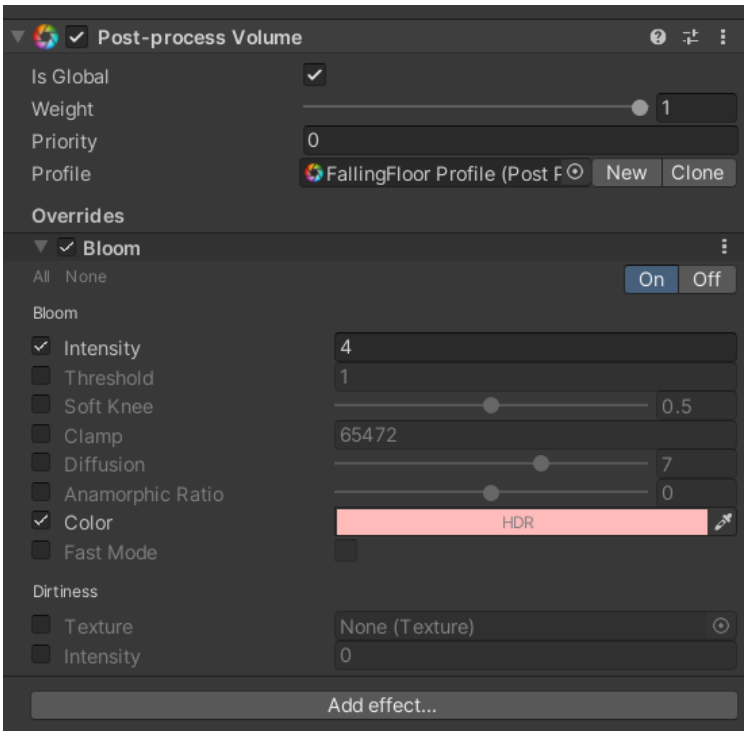

Рисунок 2.34 – Налаштування Bloom ефекту

Далі етап з розробки самої ігрової зони і першою буде плитка. Кожна плитка повинна мати колайдер, фізичне тіло та скрипт мережевої синхронізації – NetworkObject. Коли плитка вибрана на руйнування, вона починає анімацію тряски, після чого у всіх гравців вимикає свій колайдер і падає вниз. Кожна плитка має свій індентифікатор для кращого керування ними. Анімація руйнування зроблена за допомогою бібліотеки DoTween описаної в розділі 2.2. Код наступний:

```
public void StartShakeRotation(float destructionTime, TweenCallback onCompleted)
{
   transform
      .DOShakeRotation(destructionTime, _shakeVector * _shakeStrength, _vibratoPower, 90, false)
      .OnCompleted(onCompleted);
}
```

Метод DoShakeRotation приймає налаштування і трясе об’єкт певний час, по закінченню чого викликається делегат onCompleted, переданий до цього в метод OnCompleted.

Для кращого тестування гейм дизайну треба пробувати різний розмір сітчастої карти плиток. Не тільки 5 на 7. Щоб швидко тестувати розміри ігрової зони, було створено клас FallingFloorsGridEditorGenerator. Який при певних

налаштуваннях генерує сітку з плиток з потрібним відступом, кількістю, розміром плиток і т.д. Код генерації можна знайти в додатку Д (**10**).

Наступним етапом буде написання класу FallingFloorDestructor, який відповідає за руйнування плиток. Цей клас кожного кадру відслідковує кількість живих гравців і рахує час до наступної ітерації руйнування плиток. Кількість зруйнованих плиток кожен раз випадкова. В нашому випадку, від 1-3 плиток за ітерацію можуть бути зламані. Коли приходе час руйнування, клас викликає метод DestructFloor:

```
private void DestructFloor()
{
    _timeToNextDestructionIterationRemainder = _timeToNextDestructionIteration;
    float percentageOfDestructedFloorCells = (float)(_floorCells.Length - _activeFloorCells.Count) / _floorCells.Length *
100;
    int floorCellsCountToDestruction = _fallingFloorCellsCountToDestructionCalculator
        .GetCellsCountToDestruction(percentageOfDestructedFloorCells, _activeFloorCells.Count,
_blockedFloorCellsCountToDestruction);

    if (floorCellsCountToDestruction == 0)
    {
        EndMiniGame();
        return;
    }

    byte[] floorCellsIdToDestruction = GetRandomFloorCellsIdToDestruction(floorCellsCountToDestruction);
    RPC_DestructFloorCells(floorCellsIdToDestruction);
}
```

В ньому одразу перевіряється чи не потрібно закінчити гру і якщо є ще вільні плитки на руйнування, тоді він викликає метод GetRandomFloorCellsIdToDestruction,

```
private byte[] GetRandomFloorCellsIdToDestruction(int floorCellsCountToDestruction)
{
    byte[] floorCellsToDestruction = new byte[floorCellsCountToDestruction];

    for (int i = 0; i < floorCellsCountToDestruction; i++)
    {
        FallingFloorCell floorCell = _activeFloorCells[Random.Range(0, _activeFloorCells.Count)];
        floorCellsToDestruction[i] = floorCell.Id;
        _activeFloorCells.Remove(floorCell);
    }

    return floorCellsToDestruction;
}
```

який повертає байтовий масив з індентифікаторами плиток для руйнування.

Коли хост згенерував індентифікатори плиток, він передає по віддаленому мережевому виклику (RPC_DestructFloorCells) іншим гравцям цей масив, щоб анімація руйнування запустилась на всіх пристроях гравців.

```
[Rpc(RpcSources.StateAuthority, RpcTargets.All)]
private void RPC_DestructFloorCells(byte[] floorCellsIdToDestruction)
{
    for (int i = 0; i < floorCellsIdToDestruction.Length; i++)
    {
        DestructFloorCell(floorCellsIdToDestruction[i]);
    }
}

private void DestructFloorCell(byte cellId)
{
    FallingFloorCell floorCell = _floorCellsIdDict[cellId];
    _floorCellsIdDict.Remove(cellId);

    if (_activeFloorCells.Contains(floorCell))
    {
        _activeFloorCells.Remove(floorCell);
    }

    floorCell.StartDestructionProcess();
}
```

Кінцевий результат міні-гри на рисунку 2.35.

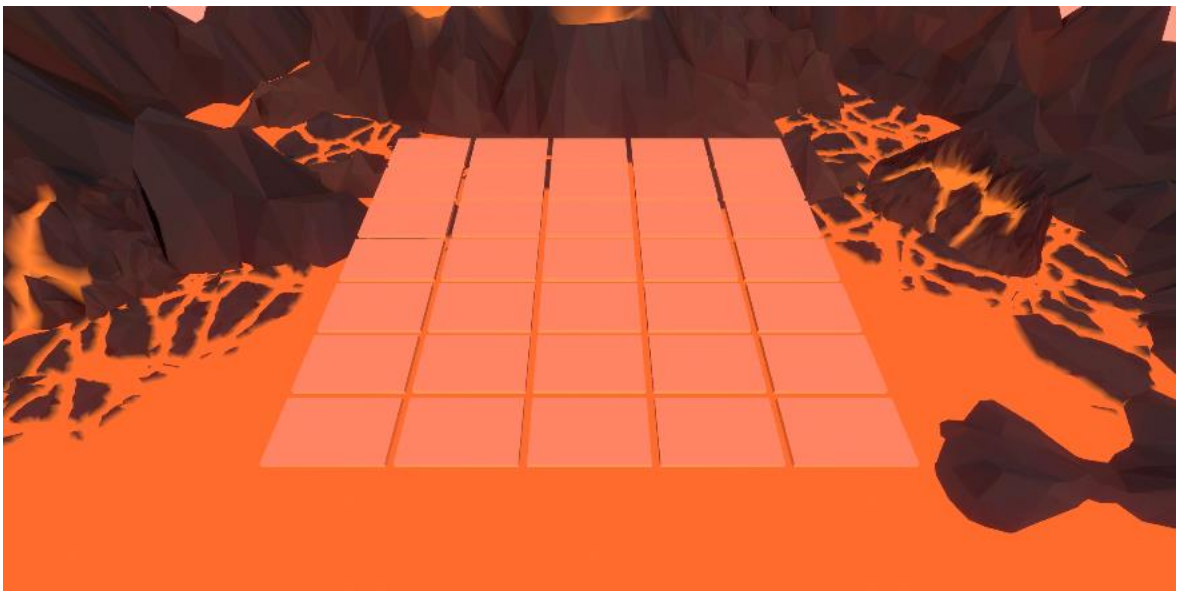

Рисунок 2.35 – Кінцевий вид міні-гри “Лавова Ущелина”

Налаштування для запікання світла такі самі, як в першій міні-грі. Друга міні-гра вийшла так само цікавою і гарною, як і перша. Хоч тут присутня випадковість і удача, але це не завадить гравцям насолодитись механікою та картою.

Тестування

Тестування любої програми – це впевненість в тому, що програма не закриється/завершиться з помилкою чи буде працювати коректно коли користувач взаємодіє з нею.

Тестування гри трішки відрізняється від тестування звичайних програм, чи серверів. Ігри мають ефекти несподіваності, а саме роботу фізики, кадрів та інших різних компонентів ігрового двигуна. В основному фізика ігрового двигуна привносить баги та помилки в гру, бо її ніяк не покрити тестами. Код гри можна покрити тестами, але в основному, якщо це простий фунціонал. Більш складний код важче тестувати. Тому в ігровому сегменті є такі люди як QA (Quality Assurance, людина, яка забезпечує якість, або тестери). Вони власноруч проходять ігрові моменти, тестують механіки, взаємодіють з фізикою та інтерфейсом. Так, це довше, набагато довше, ніж тестування коду покритим Unit-тестом. Але по-іншому протестувати функціонал фізики не вийде.

Так само доведеться і в цій грі тестувати власноруч та шукати баги, проблеми. Але ця гра має розмір кімнати до 4 гравців, невже доведеться шукати 3 друзів для цього? Ми будемо використовувати open source інструмент під назвою ParrelSync. Він надає можливість створювати копії Unity проєктів та запускати окремі вікна проєкту. Таким чином ми зможемо імітувати справжнього гравця, але маючи при цьому тільки один пристрій. Даний пакет можна скачати на GitHub.

Для того, щоб тестувати гру на 4 гравцях, нам потрібно створити 3 клона проєкту через меню ParrelSync. Після того як клони створились, можна запускати гру і заходити в кімнату/міні-гру. Процес тестування 4 гравців на рисунку 2.36:

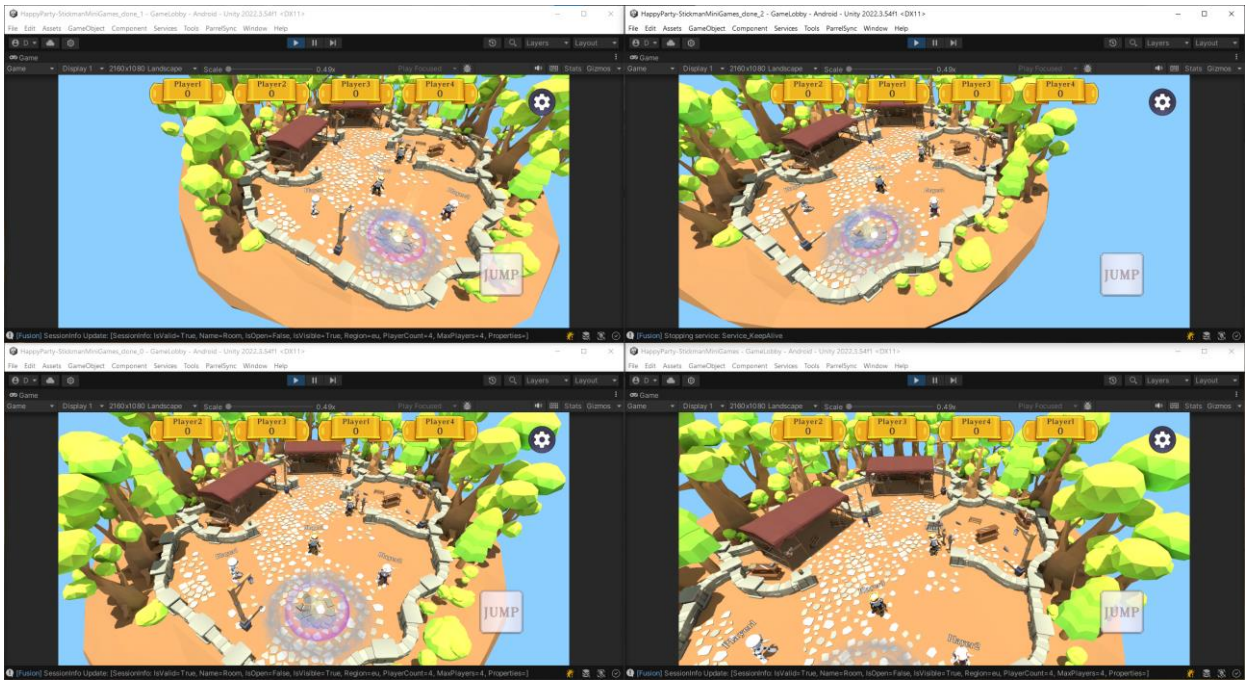

Рисунок 2.36 – Тестування гри на 4-х вікнах редактору

Тестування буде проходити в 3 етапи:

- Тестування меню підключення і його механік.
- Тестування лобі очікування.
- Тестування двох міні-ігор.

Почнемо з тестування меню підключення. Потрібно протестувати мережеві компоненти, підключення до кімнати, створення кімнати. При помилках мережевого підключення виводити на інтерфейс ці помилки. Після успішних тестувань підключень в різних випадках, є потреба протестувати інші механіки. А саме: кік гравця, заміна скіну, початок гри і вихід з кімнати. Після кіку гравця, всі модельки повинні сортуватись по-новому знову, зміна скіна має відбуватись в циклічному варіанті і синхронізуватись з іншими гравцями. Вже після початку гри, завантажується нова сцена синхронно з іншими гравцями.

Лобі-очікування не має багато механік, але ми зобов’язані протестувати його, щоб гравці не могли вийти чи перестрибнути за межі ігрової зони. Також трігер зона гравців початку міні-гри повинна працювати справно та починати гру, коли всі гравці в зоні і таймер закінчився.

Тестування двох міні-ігор займає більшість часу. Окрім основних функцій мережевих компонентів, переходів, потрібно ще тестувати і механіки. При заході в міні-гру, ефект переходу на сцену повинен відбуватись коли всі гравці приєднались до сесії. Після чого таймер відраховує певний час та запускає міні-гру. Тестуємо механіки лазерів та падіння плит на лавовій карті. В результаті, синхронізація гравців, початок міні-гри, інші механіки працюють справно та не викликають багів чи проблем.

Гра працює швидко та не потребує сильного мережевого з’єднання для роботи. Всі системи, компоненти, сервіси були протестовані на 4-х гравцях та не викликали ніяких проблем в роботі. По міркам цієї гри тестування виконано успішно.

3 РЕЗУЛЬТАТИ РОЗРОБКИ МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ

Результатом розробки мережевої багатокористувацької гри на Unity з використанням Photon Fusion є повне виконання всіх пунктів плану проєкту та її мети. Основні механіки гри були протестовані та реалізовані згідно з планом. Багатокористувацька частина працює справно та не викликає помилок/завершень роботи при виконанні програми. Гра має потрібну кількість інтерфейсу. Проєкт містить 2 міні-гри, які працюють незалежно від основної сцени та структурно відповідають принципам SOLID. Тому при додаванні нових міні-ігор достатньо зробити мінімальну кількість дій, щоб інтегрувати їх у основну ігрову сцену. Також, гравець має можливість створити безпечне підключення і кімнату до 4 людей та керувати ігровою кімнатою, якщо він є її власником. Такі фреймворки, як Zenject, UniRx дозволили створити сильну, масштабовану архітектуру проєкту.

Багатокористувацька гра випущена в Google Play Market на авторський аккаунт і має назву “Happy Party: Stickman Games”. Сторінка гри на рисунку 3.1.

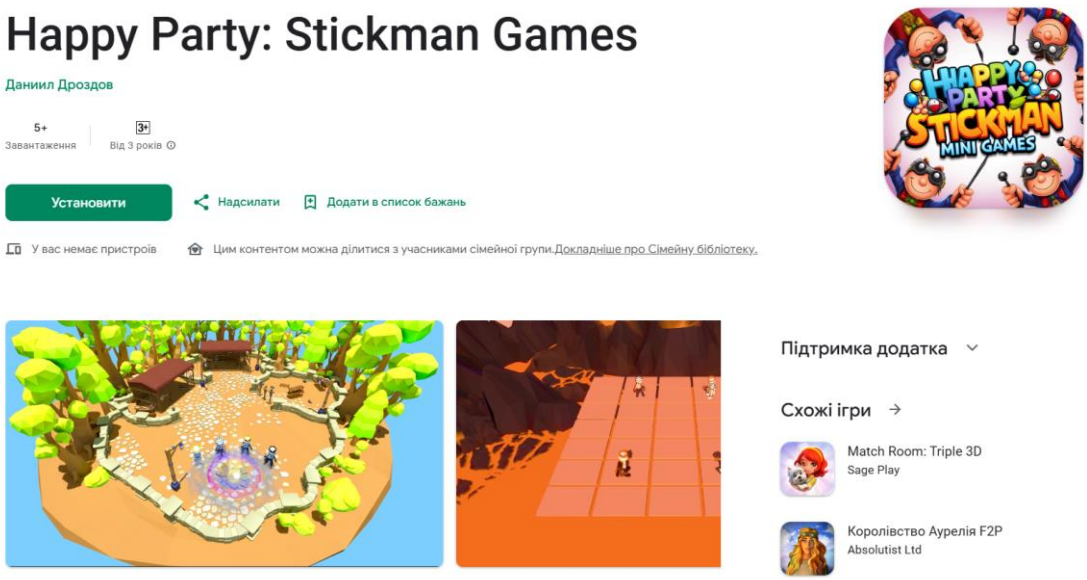

Рисунок 3.1 – Головний екран випущеної гри

Проєкт містить більше 5000 строчок авторського коду. Написання та створення гри зайняло близько 200 годин часу. Оптимізація коду дозволяє запускати гру на мобільних пристроях без відчутних кадрових просадок. Мережева частина не потребує сильного з’єднання для гравців. Але, звісно, для хоста потрібний більш стабільний інтернет. Середній показник витрат процесора на комп’ютері дорівнює близько 3 мс (що еквівалентно 300 фпс). На телефоні показник 12 мс, що дорівнює 80 фпс.

4 ОХОРОНА ПРАЦІ

Охорона праці включає всі необхідні методи і заходи, які призначені для зменшення ризику травматизму та здоров'я, що можуть виникнути серед робітників, у межах підприємств або в окремих галузях.

Основними законодавчими та нормативними документами, які регламентують охорону праці в нашій країні є низка Законів України «Про охорону праці», «Про пожежну безпеку» та «Про охорону здоров’я».

Дефініція поняття «Охорона праці» представлена в першій статті Закону України «Про охорону праці». Охорона праці – це система правових, соціально–економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження здоров'я і працездатності людини в процесі праці. Цей Закон визначає основні положення щодо реалізації конституційного права громадян на охорону їх життя і здоров'я в процесі трудової діяльності, регулює за участю відповідних державних органів відносини між власником підприємства, установи і організації або уповноваженим ним органом і працівником з питань безпеки, гігієни праці та виробничого середовища і встановлює єдиний порядок організації охорони праці в Україні [9].

4.1 Підготовка робочого місця перед роботою з комп’ютером

У роботі з комп’ютером однією з ключових умов безпечної праці є правильна організація робочого місця. Від цього залежить не лише продуктивність, а й стан здоров’я працівника. Робоче місце повинно відповідати санітарно-гігієнічним та ергономічним вимогам.

Зокрема, висота столу має становити 70–80 см, а стілець повинен бути регульованим за висотою зі спинкою, що нахиляється під кутом 90-110°. Монітор комп’ютера необхідно встановлювати на відстані 50–70 см від очей користувача

(або на довжину витягнутої руки), при цьому верхня межа екрана повинна бути на рівні очей або трохи нижче.

Клавіатура повинна бути розміщена на поверхні столу або на спеціальній панелі таким чином, щоб зап’ястки користувача залишалися у природному положенні. Освітлення робочої зони повинно бути достатнім, при цьому необхідно уникати прямих відблисків на екрані монітора.

4.2 Режим праці та профілактика професійної втоми

Тривала робота за комп’ютером призводить до підвищеного зорового та психоемоційного навантаження, що може викликати перевтому, зниження продуктивності та розвиток професійних захворювань.

Згідно з нормами охорони праці, тривалість безперервної роботи за комп’ютером не повинна перевищувати 2 годин. Після цього рекомендується зробити перерву тривалістю 10–15 хвилин. Загальний час роботи з монітором протягом дня не повинен перевищувати 6 годин.

Кожні 45–60 хвилин працівнику слід робити короткі перерви для виконання вправ загальної фізичної активності та гімнастики для очей. Такі вправи сприяють розслабленню м’язів та зняттю втоми з органів зору. Наприклад, рекомендовано на 20–30 секунд переводити погляд на віддалені об’єкти (відстань 20–30 метрів), що суттєво знижує навантаження на очі.

4.3 Електробезпека та пожежна безпека при роботі з комп’ютерною технікою

У домашніх умовах питання електробезпеки часто ігноруються, що створює додаткові ризики. Для забезпечення безпечної роботи комп’ютера вдома необхідно дотримуватись базових правил користування електроприладами.

Передусім, усі пристрої повинні бути підключені до справної електромережі із заземленням. Якщо в будинку заземлення відсутнє, рекомендується

використовувати мережеві фільтри з вбудованим захистом від перенапруги. Це дозволить уникнути виходу з ладу техніки під час перепадів напруги.

Небажано використовувати подовжувачі з перевантаженням – кількість одночасно підключених пристроїв не повинна перевищувати допустиму для конкретного фільтра або розетки. Особливу увагу потрібно звертати на стан кабелів: не допускається використання проводів з тріщинами, надломами чи оголеними ділянками.

У разі перегріву блоку живлення, появи запаху гару або незвичних звуків комп’ютер слід негайно вимкнути з мережі. В приміщенні бажано мати вогнегасник невеликого розміру (наприклад, порошковий), а також знати базовий алгоритм дій при пожежі.

Не рекомендується залишати комп’ютер увімкненим на ніч або без нагляду на тривалий час, особливо якщо він знаходиться в безпосередній близькості до меблів, фіранок або інших легкозаймистих предметів [10]

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було досягнуто поставлену мету та успішно вирішено всі визначені завдання. Проведено аналіз сучасних методів розробки багатокористувацьких ігор на базі Unity, завдяки чому були виявлені сильні й слабкі сторони різних рішень та обрано найефективніші практики. Були розроблені архітектури мережевої частини проєкту та основної ігрової сцени, яка підв’язує до себе міні-ігри .

Під час тестування підтверджено, що система Photon Fusion забезпечує стабільну та надійну мережеву взаємодію. А архітектура проєкту є легко–масштабованою та оптимізованою, що дає можливість в додатковому розвитку гри і її покращення в майбутньому.

Таким чином, поставлена мета кваліфікаційної роботи досягнута, всі завдання успішно виконані, а отримані результати можуть бути використані для подальшого розвитку багатокористувацьких ігор на платформі Unity з застосуванням Photon Fusion.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. VRChat логотип. Website: Steam. URL: https://shared.fastly.steamstatic.com/store_item_assets/steam/apps/438100/header.jpg?t=1733413042 (дата звернення 18.05.2025).
2. Stumble Guys логотип. Website: Steam. URL: https://shared.fastly.steamstatic.com/store_item_assets/steam/apps/1677740/header.jpg?t=1742999151 (дата звернення 18.05.2025).
3. Liar's Bar логотип. Website: Steam. URL: https://shared.fastly.steamstatic.com/store_item_assets/steam/apps/3097560/41301171196173fe525f397e8a0e2bc37f383da2/header.jpg?t=1741802069 (дата звернення 18.05.2025).
4. Lethal Company логотип. Website: Steam. URL: https://shared.fastly.steamstatic.com/store_item_assets/steam/apps/1966720/header.jpg?t=1742986399 (дата звернення 18.05.2025).
5. Pummel Party логотип. Website: Steam. URL: https://shared.fastly.steamstatic.com/store_item_assets/steam/apps/880940/header.jpg?t=1725311353 (дата звернення 18.05.2025).
6. DI (Dependency Injection). Website: Wikipedia. URL: https://uk.wikipedia.org/wiki/%D0%92%D0%BF%D1%80%D0%BE%D0%B2%D0%B0%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%B7%D0%B0%D0%BB%D0%B5%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D0%B5%D0%B9 (дата звернення 18.05.2025).
7. Реактивне програмування. Website: Wikipedia. URL: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B0%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F (дата звернення 18.05.2025).

8. Photon Fusion Documentation. Website: Photon. URL: https://doc.photonengine.com/fusion/current/fusion–intro (дата звернення 18.05.2025).
9. Про охорону праці: Закон України від 21.11.2002 р. No 229–IV. URL: https://zakon.rada.gov.ua/laws/show/2694–12#Text (дата звернення 18.05.2025).
10. Правила безпеки за комп’ютером. URL: https://pro–op.com.ua/article/183–ohoron–prats–pri–robot–z–kompyuterom (дата звернення 18.05.2025).

ДОДАТОК А – ТЕХНІЧНЕ ЗАВДАННЯ НА РОЗРОБКУ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY З ВИКОРИСТАННЯМ PHOTON FUSION

Вступ

Повна назва розроблюваного проекту: “ Розробка багатокористувацької гри на Unity з використанням Photon Fusion”.

Галузь застосування: комп’ютери, телефони та різні пристрої гравців.

1 Підстави для розробки

Підставою для розробки програмного забезпечення є завдання на кваліфікаційну роботу на здобуття ступеня вищої освіти «бакалавр».

2 Призначення розробки

2.1 Функціональне призначення

Функціональним призначенням програмного забезпечення є створення гри на сильному мережевому багатокористувацькому фреймворку, покращення продуктивності мережевої взаємодії та посилення можливостей проекту.

2.2 Експлуатаційне призначення

Експлуатаційним призначенням програмного забезпечення є полегшення розробки багатокористувацької гри та її майбутнього масштабування.

3 Вимоги до програмного забезпечення

Вимоги до складу виконуваних функцій

Гра повинна мати можливість виконання наступних функцій:

- Підключення до сесії: дозволяє гравцям під’єднатися до існуючої ігрової сесії.

- Створення нової сесії: дозволяє гравцю створити нову ігрову сесію.
- Налаштування сесії: дозволяє гравцю налаштувати новостворену ігрову сесію під власні потреби.
- Початок гри: дозволяє почати гру для всіх гравців, які знаходяться в спільній ігровій сесії.
- Синхронізація станів: дозволяє серверу синхронізовувати дані між гравцями.
- Надання списку активних сесій: надає можливість системі отримувати інформацію про активні сесії на сервері.

3.2 Вимоги до надійності

Програмне забезпечення має функціонувати неперервно на персональних обчислювальних системах. У випадку виникнення помилок у роботі комп'ютера, відновлення нормального функціонування повинно відбуватися після перезавантаження програмного продукту.

3.3 Вимоги до умов експлуатації

3.3.1 Кліматичні умови експлуатації

Умови експлуатації програми збігаються з умовами експлуатації технічних засобів, на яких буде завантажена та використовуватись програма для подальшого використання.

3.3.2 Вимоги до чисельності та кваліфікації користувачів

Необхідний рівень підготовки користувачів: мінімальні навички в користуванні комп’ютером.

3.4 Вимоги до інформаційної та програмної сумісності

Програма повинна працювати на всіх версіях Android починаючи з 27 версії.

3.5 Вимоги до складу та параметрів технічних засобів

– RAM: 2 GB

– До 150 МБ пам’яті.

4 Вимоги до програмної документації

Програмна документація повинна містити:

– технічне завдання;

– опис програми;

– програму та методику випробувань;

– інструкцію користувача;

– тексти програмних модулів.

5 Техніко–економічні показники

У цій кваліфікаційній роботі не проводився розрахунок техніко–економічних показників.

6 Стадії та етапи розробки

Стадії та етапи розробки багатокористувацької гри на Unity з використанням Photon Fusion представлені в таблиці А.1.

Таблиця А.1 – Стадії та етапи розробки програмного забезпечення

Стадії розробки	Етапи розробки	Дата початку	Дата завершення
1	2	3	4
1 Технічне завдання	1.1 Обґрунтування необхідності розробки програми	20.03.2025	21.03.2025
	1.2 Розробка технічного завдання	20.03.2025	21.03.2025
	1.3 Затвердження технічного завдання	22.03.2025	23.03.2025
2 Ескізний проект	2.1 Розробка ескізного проекту	25.03.2025	25.03.2025

Кінець таблиці А.1

1	2	3	4
2 Ескізний проект	2.2 Затвердження ескізного проекту	26.03.2025	26.03.2025
3 Технічний проект	3.1 Розробка технічного проекту	02.04.2025	06.04.2025
	3.2 Затвердження технічного проекту	07.04.2025	07.04.2025
4 Робочий проект	4.1 Розробка програми	10.04.2025	22.05.2025
	4.2 Розробка програмної документації	23.05.2025	26.05.2025
	4.3 Випробування програми	26.05.2025	29.05.2025

5 Порядок контролю та приймання

Для контролю приймання має бути надано опис програми, а також програму і методику випробувань. Якщо програма не пройшла випробування, виконавець зобов’язаний виправити помилки і недоліки в строк, не більш ніж 2 тижні від дня випробування.

ДОДАТОК Б – ПРОГРАМА ТА МЕТОДИКА ВИПРОБОВУВАНЬ МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY З ВИКОРИСТАННЯМ PHOTON FUSION

1 Об’єкт випробування

Об’єктом випробувань є мережева багатокористувацька гри на Unity.

2 Мета випробувань

Метою проведення випробувань є перевірка працездатності гри, перевірка відповідності характеристик розробленої гри та вимог викладених в технічному завданні.

3 Вимоги до програми

При проведенні випробувань функціональних характеристик, мережева багатокористувацька гра підлягає перевірці на відповідність вимогам, викладеним у пункті «Вимоги до функціональних характеристик» технічного завдання.

4 Вимоги до програмної документації

До складу програмної документації повинні входити наступні документи:

- Технічне завдання.
- Програма та методика випробувань.
- Інструкція користувача.
- Опис програми.
- Текст програми.

5 Засоби і порядок випробувань

5.1 Склад технічних засобів

До складу технічних засобів, які використовувалися під час випробувань відносяться:

- Процесор: 4 ядра, 2.2 ГГц.
- Оперативна пам’ять: 4 ГБ.
- Жорсткий диск: 64 ГБ.
- Інтернет з’єднання зі швидкістю 10 Мбіт/с.

5.2 Склад програмних засобів

До складу програмних засобів, які використовувались при проведенні випробувань відносяться:

- Операційна система Android 11.

5.3 Порядок проведення випробувань

- Перевірка вимог до функціональних характеристик.
- Виконання тестів в довільному порядку.
- Аналіз отриманих результатів тестування.
- Відповідність програмного продукту вимогам.

Після виявлення недоліків у функціонуванні системи, створюється список цих проблем, який обговорюється з розробником для визначення терміну їх виправлення. Після цього замовник проводить повторне тестування, охоплюючи всі аспекти системи, можливо навіть застосовуючи нові або додаткові тести.

6 Методи випробувань

Методом випробувань для тестування розробленого програмного забезпечення було обрано метод ручного тестування. Так як гра має багато непередбачуваних моментів, таких як: нестабільне з’єднання, випадкова поведінка фізики, дії гравця, то такі моменти потрібно перевіряти вручну і тестувати різні варіації дій. В таблиці Б.1 представлено очікувані результати випробувань для деяких функцій.

Таблиця Б.1 – Очікувані результати випробувань для деяких функцій.

Дія	Очікуваний результат
Створення сесії	Успішне створення сесії з переходом в лобі.
Підключення до існуючої сесії	Успішне підключення до існуючої сесії з переходом в лобі.
Початок гри	Успішний початок гри для всіх гравців з переходом на сцену лоббі-очікування.
Синхронізація станів гравців	Синхронізовані параметри, стани, дані гравців на всіх пристроях.
Налаштування сесії	Гравець має можливість налаштовувати ігрову сесію під власні потреби.
Отримання списку активних сесій	Система успішно отримує список активних ігрових сесій для гравця.
Керування гравцем	Плавне, без переривань керування гравцем на мобільних пристроях.
Завершення гри	Робоче завершення гри для гравців з відображенням результатів та переходом в головне меню.

ДОДАТОК В – ІНСТРУКЦІЯ КОРИСТУВАЧА МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY З ВИКОРИСТАННЯМ PHOTON FUSION

Для роботи з мережевою багатокористувацькою грою на Unity потрібно перейти до сторінки гри на Play Market і завантажити її на мобільний Android пристрій.

Сторінка гри в Play Market наведена на рисунку В.1.

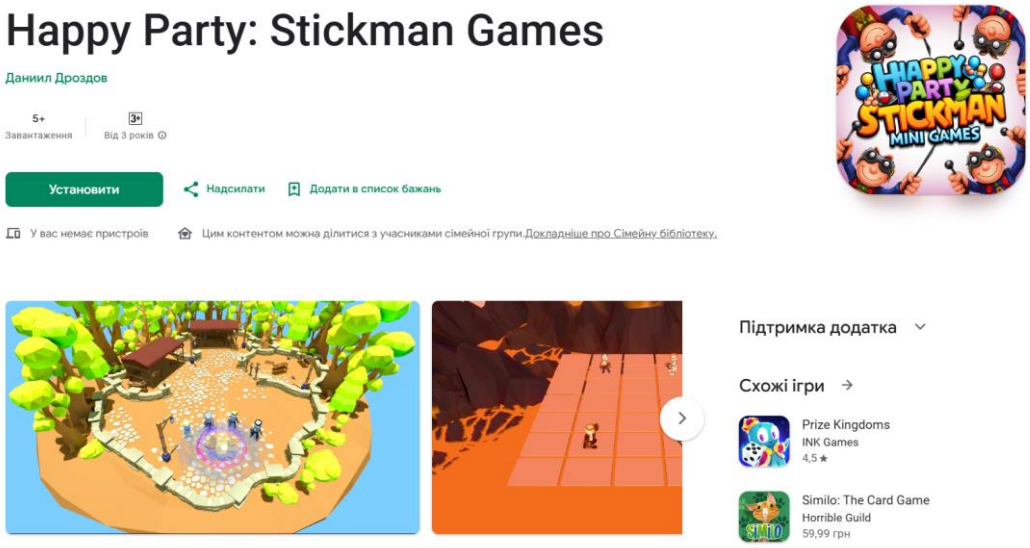

Рисунок В.1 – Головна сторінка гри в Play Market

Після запуску гри, гравець попадає в головне меню, де потрібно ввести нікнейм. Також гравець має дві опції: “Connect” – підключення до існуючої сесії та “Create Room” – створення власної сесії. Ці кнопки показані на рисунку В.2.

Щоб створити сесію, потрібно ввести назву сесії, якої ще не існує. При успішному створенні гравець може очікувати інших користувачів і потім почати гру.

Для підключення до сесії, гравцю треба ввести назву сесії, яка існує. Після успішного підключення, гравцю потрібно чекати на початок гри від хоста.

Рисунок В.2 – Головне меню гри

Коли почалась гра з головного меню, гравці попадають на сцену лобі очікування. Щоб почати міні-гру, усім гравцям потрібно встати в портал і почекати 5 секунд. Портал, який використовується в грі, показаний на рисунку В.3.

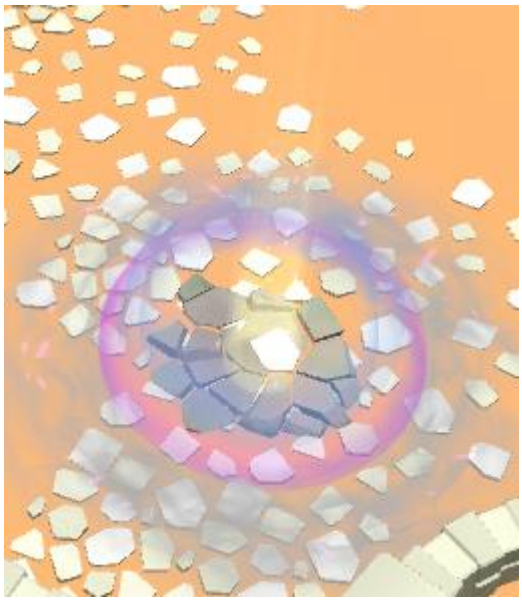

Рисунок В.3 – Портал в міні-гру

Коли почнеться будь-яка міні-гра, задача користувача – грати. Після де-якого часу гри, міні-гра закінчується і у всіх користувачів з’являється сцена показу зароблених очків. Через 10 секунд всіх гравців знову перенаправляють в лобі очікування, де їм потрібно пройти в портал для початку нової міні-гри. І так до того моменту, поки хтось з гравців не переможе.

ДОДАТОК Г – ОПИС МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY З ВИКОРИСТАННЯМ PHOTON FUSION

1 Загальні відомості

1.1 Позначення і найменування програми

Найменування: Мережева багатокористувацька гра на Unity з використанням Photon Fusion.

1.2 Програмне забезпечення, необхідне для функціонування програми

Для роботи програмного забезпечення необхідна наявність Android пристрою як мінімум 6 версії.

1.3 Мови програмування

Для розробки мережевої багатокористувацької гри було використано фреймворк Photon Fusion, мову програмування C#. Інтегроване середовище розробки, що було використано під час розробки – Jebrains Rider 2024.3.

2 Функціональне призначення

2.1 Класи розв’язуваних завдань та призначення програми

Багатокористувацька мережева гра повинна забезпечувати виконання нижче перерахованих функцій:

- Підключення до сесії.
- Створення нової сесії.
- Початок гри.
- Синхронізація станів гравців.
- Функціональні міні-ігри.

2.2 Функціональні обмеження

Одним з функціональних обмежень мережевої багатокористувацької гри є необхідність підключення до мережі інтернету.

3 Опис логічної структури

3.1 Використовуванні методи

При розробці мережевої багатокористувацької гри на Unity та проведення його аналізу було використано об’єктно-орієнтовану методологію та об’єктно-орієнтовану мову програмування C#.

3.2 Алгоритм програми

Програмне забезпечення гри реалізовано з використанням Photon Fusion - фреймворку для розробки мережевих багатокористувацьких ігор. Нижче наведено опис алгоритму заходу користувача в гру:

1) Користувач заходить в гру.
2) Якщо є підключення до мережі, система надсилає запит підключення на сервера Photon Fusion.
3) Сервера Photon Fusion обробляють запит і надають дані пристрою користувача.
4) При успішному запиті система запускає першу сцену в грі – головне меню.

4 Виклик і завантаження

Після розгортання гри в Play Market, гру можна завантажити з сторінки площадки. Після встановлення з Play Market-у, є можливість запустити гру з телефона.

5 Вхідні і вихідні дані

Введення інформації здійснюється за допомогою екранної клавіатури. Вхідні дані повинні вводитися у спеціальні поля в меню. Вихідні дані відображаються на екрані пристрою

ДОДАТОК Д – ТЕКСТ МЕРЕЖЕВОЇ БАГАТОКОРИСТУВАЦЬКОЇ ГРИ НА UNITY З ВИКОРИСТАННЯМ PHOTON FUSION

1 Реалізація AssetsLoader (завантажувач ресурсів)

```
public class AssetsLoader : IAssetsLoader
{
    public async UniTask<T> LoadAssetAsync<T>(AssetReference assetReference)
    {
        AsyncOperationHandle<T> operationHandle =
Addressables.LoadAssetAsync<T>(assetReference);


        return await ProcessAssetLoad(operationHandle, assetReference);
    }

    public async UniTask<GameObject> InstantiateGameObjectAsync(AssetReference
assetReference, Vector3 position, Quaternion rotation)
    {
        AsyncOperationHandle<GameObject> operationHandle =
Addressables.InstantiateAsync(assetReference, position, rotation);


        return await ProcessAssetLoad(operationHandle, assetReference);
    }

    private async UniTask<T> ProcessAssetLoad<T>(AsyncOperationHandle<T>
operationHandle, AssetReference assetReference)
    {
        await operationHandle.Task;

        CheckOnFailedStatus(operationHandle, assetReference);

        return operationHandle.Result;
    }

    private void CheckOnFailedStatus(AsyncOperationHandle operationHandle,
AssetReference assetReference)
    {
        if (operationHandle.Status == AsyncOperationStatus.Failed)
        {
            throw new Exception("Asset load failed. Reference: " + assetReference);
        }
    }
}
```

В даному коді є два основних асинхронних метода. LoadAssetAsync – завантажує асинхронно в пам’ять любий асет (текстуру, звук і т.д). InstantiateGameObjectAsync – завантажує в пам’ять об’єкт і спавнить його на сцені. І базовий метод, який виконує процес – ProcessAssetLoad, CheckOnFailedStatus – перевіряє помилки.

2 Реалізація ObjectsFactory (фабрики об’єктів)

```
public class ObjectsFactory : IObjectsFactory
{
    private IDependenciesInjector _dependenciesInjector;
    private readonly IAssetsLoader _assetsLoader;

    public ObjectsFactory(IAssetsLoader assetsLoader, IDependenciesInjector
dependenciesInjector)
    {
        _assetsLoader = assetsLoader;
        _dependenciesInjector = dependenciesInjector;
    }

    public async UniTask<GameObject> Create(AssetReference assetReference)
    {
        return await Create(assetReference, VectorConstants.Zero, Quaternion.identity);
    }

    public async UniTask<GameObject> Create(AssetReference assetReference, Vector3 pos)
    {
        return await Create(assetReference, pos, Quaternion.identity);
    }

    public async UniTask<GameObject> Create(AssetReference assetReference, Vector3 pos,
Quaternion rotation)
    {
        GameObject instantiatedObj = await
_assetsLoader.InstantiateGameObjectAsync(assetReference, pos, rotation);
        instantiatedObj.AddComponent<SelfResourceReleaser>();
        _dependenciesInjector.InjectGameObject(instantiatedObj);
        return instantiatedObj;
    }
}
```

У цьому класі є конструктор, який заповнює Zenject залежностями. А саме IAssetsLoader (інтерфейс класу з додатку Д (**1**)) і IDependenciesInjector (інтерфейс впровадження залежностей). Є 3 перегружених метода, один з яких це основний в самому низу. Дана фабрика створює об’єкт з потрібним місцем і поворотом, додає компонент і ініцалізує всі скрипти залежностями, що висять на об’єкті.

3 Реалізація ApplicationInputProvider (сервіс вводу гравця)

```
public class ApplicationInputProvider : IApplicationInputProvider, IApplicationInputBlocker
{
    private readonly IPlatformInputProvider _platformInputProvider;
    private bool _isInputBlocked;

    public ApplicationInputProvider()
    {
        if (Application.platform == RuntimePlatform.WindowsEditor)
        {
            _platformInputProvider = new StandaloneInputProvider();
        }
        else
        {
            _platformInputProvider = new MobileInputProvider();
        }
    }

    public bool IsInputBlocked() => _isInputBlocked;

    public PlayerNetworkInput GetPlayerInputForNetwork()
    {
        if (_isInputBlocked)
        {
            return new PlayerNetworkInput();
        }

        return _platformInputProvider.GetInput();
    }

    public void BlockUserInput()
    {
        _isInputBlocked = true;
    }

    public void UnblockUserInput()
    {
        _isInputBlocked = false;
    }
}
```

У цьому класі можна побачити конструктор, який дивиться поточну платформу, на якій запустилась гра, і від результату назначає змінній _platformInputProvider інтерфейсу IPlatformInputProvider об’єкт цільової платформи, що робить код дуже гибким і легким до розширення на нових платформах.

4 Частина реалізації GameDataSaver (зберігач даних гри)

```
private void LoadData()
{
    if (!File.Exists(FullPath))
    {
        _data = new GameSaveData();
        return;
    }

    FileStream fs = null;

    try
    {
        fs = new FileStream(FullPath, FileMode.Open);
        _data = (GameSaveData)Bf.Deserialize(fs);
    }
    finally
    {
        _data ??= new GameSaveData();

        try
        {
            fs?.Close();
        }
    }
}

private void SaveData()
{
    if (SavedThisFrame) return;

    Directory.CreateDirectory(DirPath);
    FileStream fs = null;

    try
    {
        fs = new FileStream(FullPath, FileMode.Create);
        Bf.Serialize(fs, _data);
    }
    finally
    {
        try
        {
            fs?.Close();
        }
    }

    _lastSavedFrame = Time.frameCount;
}
```

LoadData – метод завантаження даних, SaveData – метод зберігання даних.

5 Реалізація NetworkConnector (сервіс підключення до мережевих компонентів)

```
public class NetworkConnector : INetworkConnector, INetworkConnectorCallbacksObserver, INetworkDisconnecter
{
    private readonly IObjectsFactory _objectsFactory;
    private readonly IDependenciesInjector _dependenciesInjector;
    private readonly INetworkActiveRunnerInfoProviderNewRunnerSetter _activeRunnerInfoProviderNewRunnerSetter;
    private readonly AssetReference _networkRunnerRef;
    private readonly NetworkRoomSettings _networkRoomSettings;

    private NetworkRunner _curNetworkRunner;

    public INetworkCallbacksEvents NetworkCallbacksEvents { get; private set; }
    public event Action<NetworkRunner> OnSetNewRoomConnection;

    public NetworkConnector(IObjectsFactory objectsFactory, IDependenciesInjector dependenciesInjector,
        INetworkActiveRunnerInfoProviderNewRunnerSetter activeRunnerInfoProviderNewRunnerSetter, AssetReference
networkRunnerRef, NetworkRoomSettings networkRoomSettings)
    {
        _objectsFactory = objectsFactory;
        _dependenciesInjector = dependenciesInjector;
        _activeRunnerInfoProviderNewRunnerSetter = activeRunnerInfoProviderNewRunnerSetter;
        _networkRunnerRef = networkRunnerRef;
        _networkRoomSettings = networkRoomSettings;
    }

    public bool CanCreateNewConnection() => _curNetworkRunner == null;

    public async UniTask<StartGameResult> ConnectToRoom(string roomName = null)
    {
        return await CreateConnection(GameMode.Client, roomName);
    }

    public async UniTask<StartGameResult> CreateRoom(string roomName)
    {
        return await CreateConnection(GameMode.Host, roomName);
    }

    public void ShutdownConnection()
    {
        if (_curNetworkRunner == null) return;

        _curNetworkRunner.Shutdown();
    }

    private void SetNewRunnerInDataIfResultIsOk(bool isConnectionResultOk)
    {
        if (isConnectionResultOk)
        {
            _activeRunnerInfoProviderNewRunnerSetter.SetNewRunner(_curNetworkRunner);
        }
    }
```

```
private async UniTask<StartGameResult> CreateConnection(GameMode gameMode, string roomName = null)
    {
        await CreateNetworkRunner();
        INetworkSceneManager networkSceneManager =
_curNetworkRunner.GetComponent<INetworkSceneManager>();
        NetworkObjectProvider objectProvider = _curNetworkRunner.GetComponent<NetworkObjectProvider>();
        objectProvider.SetDependencies(_dependenciesInjector);

        StartGameResult startGameResult  = await _curNetworkRunner.StartGame(new StartGameArgs()
        {
            GameMode = gameMode,
            SessionName = roomName,
            ObjectProvider = objectProvider,
            SceneManager = networkSceneManager,
            PlayerCount = _networkRoomSettings.MaxPlayers,
            OnGameStarted = (runner) => OnSetNewRoomConnection?.Invoke(runner)
        });

        SetNewRunnerInDataIfResultIsOk(startGameResult.Ok);

        return startGameResult;
    }

    private async UniTask CreateNetworkRunner()
    {
        GameObject runner = await _objectsFactory.Create(_networkRunnerRef);
        _curNetworkRunner = runner.GetComponent<NetworkRunner>();
        NetworkCallbacksEvents = _curNetworkRunner.GetComponent<INetworkCallbacksEvents>();
    }

    private void SetNewRunnerInDataIfResultIsOk(bool isConnectionResultOk)
    {
        if (isConnectionResultOk)
        {
            _activeRunnerInfoProviderNewRunnerSetter.SetNewRunner(_curNetworkRunner);
        }
    }
}
```

Цей клас створює підключення (створення кімнати – CreateRoom, підключення до існуючої кімнати – ConnectToRoom). Використовує спільний метод – CreateConnection, в який передаються налаштування в залежності від вибраного підключення. Для відключення від сесії використовується метод – ShutdownConnection. Конструктор приймає різні мережеві і проєктні налаштування для успішної роботи підключень.

6 Часткова реалізація StartGameLobbyPortalTrigger (портал в лобі очікування для старту гри)

```
public void OnSystemFixedUpdate(float deltaTime) // physics tick every 0.02ms
{
    FindPlayers();
    CheckIfPlayersInZoneListChanged();
}

private void CheckIfPlayersInZoneListChanged()
{
    if (_playersInZone.Count != _lastPlayersInZoneCount)
    {
        OnPlayersCountInZoneUpdated?.Invoke(_playersInZone.Count);
        _lastPlayersInZoneCount = _playersInZone.Count;
    }
}

private void FindPlayers() // find players in trigger zone
{
    FillListWithPlayersOnLastIteration();
    int foundColliders = Physics.OverlapSphereNonAlloc(_transform.position, _triggerRadius,
_sphereOverlapPlayersResult, _playerLayerMask);

    for (int i = 0; i < foundColliders; i++)
    {
        Collider player = _sphereOverlapPlayersResult[i];

        if (!_playersInZone.Contains(player))
        {
            _playersInZone.Add(player);
        }
        else
        {
            _playersInZoneOnLastIteration.Remove(player);
        }
    }

    for (int i = 0; i < _playersInZoneOnLastIteration.Count; i++)
    {
        _playersInZone.Remove(_playersInZoneOnLastIteration[i]);
    }
}

private void FillListWithPlayersOnLastIteration()
{
    _playersInZoneOnLastIteration.Clear();

    foreach (Collider player in _playersInZone)
    {
        _playersInZoneOnLastIteration.Add(player);
    }
```

7 Реалізація класу PlayersLevelLoadingStateChecker

```
public class PlayersLevelLoadingStateChecker : NetworkBehaviour
{
    private List<int> _playersId = new (4);

    public event Action OnAllPlayersLoaded;

    public override void Spawned()
    {
        RPC_OnLoadedInScene(Runner.LocalPlayer.PlayerId);
    }

    #region Network RPCs

    [Rpc(RpcSources.All, RpcTargets.StateAuthority)]
    private void RPC_OnLoadedInScene(int playerId)
    {
        _playersId.Add(playerId);
        CheckOnServerIfAllPlayersLoaded();
    }

    #endregion

    private void CheckOnServerIfAllPlayersLoaded()
    {
        IEnumerable<PlayerRef> activePlayersList = Runner.ActivePlayers;

        if (activePlayersList.Count() != _playersId.Count) return;

        bool hasBrokenPlayerId = false;

        foreach (PlayerRef playerRef in activePlayersList)
        {
            if (!_playersId.Contains(playerRef.PlayerId))
            {
                hasBrokenPlayerId = true;
            }
        }

        if (!hasBrokenPlayerId)
        {
            OnAllPlayersLoaded?.Invoke();
        }
    }
}
```

Даний клас існує у кожного гравця. Як тільки гравець підключається, він викликає метод Spawned –> RPC_OnLoadedInScene, що передається хосту. CheckOnServerIfAllPlayersLoaded перевіряє чи всі гравці у хоста завантажились на сервер. Як тільки ця умова буде виконана, код викличе івент OnAllPlayersLoaded.

8 Реалізація шейдера (часткова) екрана переходу мовою GLSL.

```
Shader "UI/SceneTransitionShader"
{
    ...
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            ....

            sampler2D _MainTex;
            float4 _MainTex_ST;
            half4 _Color;
            fixed4 _TransparentColor = fixed4(1,1,1,0);
            float _Radius;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                fixed4 col = tex2D(_MainTex, i.uv) * _Color; // 1
                float3 pixelVectorFromCenter = float3(i.uv.x - 0.5f, i.uv.y - 0.5f, 0); // 2
                return length(pixelVectorFromCenter) > _Radius / 100 ? col : _TransparentColor; // 3
            }
            ENDCG
        }
}
```

Повністю розписати, що тут і як це працює на одну сторінку не вийде. Так як це обширна і велика тема. Але нас цікавить frag метод. Він відповідає за малювання пікселю на екрані. Маючи текстуру і додатковий колір, ми отримуємо базовий колір пікселю на коментарі 1 в коді. Наступна строка з коментарем 2 рахує дистанцію пікселя в UV координатах. 3 строка, використовуючи тернарну операцію, підраховує скалярну величину вектора пікселя до центру і порівнює з радіусом кола з зовнішнього коду. І якщо, наприклад, дистанція 0.6 буде більше за 0.55 (радіус кола), то цей піксель буде зафарбованим жовтим (в нашому випадку) кольором. Інакше, це буде прозорий піксель.

9 Реалізація класу AncientArenaLasersThrower (кидач лазерів)

```
public class AncientArenaLasersThrower : MiniGameStarter, IUpdatable
   {
      [SerializeField, MinValue(0)]
      private float _timeToSpawnNewLasersPair;

      [SerializeField, Space(15)]
      private AncientLasersCountThrowChance[] _ancientLasersCountThrowChances;

      [SerializeField, Sirenix.OdinInspector.ReadOnly, Space(15)]
      private AncientLasersArenaBorder[] _arenaBorders;

      private AncientLasersThrowCountRandomizer _lasersThrowCountRandomizer;
      private AncientLasersArenaSettings _ancientLasersArenaSettings;
      private IMiniGamePlayersStorage _playersStorage;
      private IMiniGameDurationCountdownTimer _miniGameDurationCountdownTimer;
      private float _timeToSpawnNewLasersPairRemainder;

      private void Awake()
      {
         _lasersThrowCountRandomizer = new
AncientLasersThrowCountRandomizer(_ancientLasersCountThrowChances);
         _timeToSpawnNewLasersPairRemainder = _timeToSpawnNewLasersPair;
      }

      public void Inject(AncientLasersArenaSettings ancientLasersArenaSettings, IMiniGamePlayersStorage
playersStorage,
         IMiniGameDurationCountdownTimer miniGameDurationCountdownTimer)
      {
         _ancientLasersArenaSettings = ancientLasersArenaSettings;
         _miniGameDurationCountdownTimer = miniGameDurationCountdownTimer;
         _playersStorage = playersStorage;
      }

      public void OnSystemUpdate(float deltaTime)
      {
         if (_playersStorage.ActiveLivePlayers.Count == 0)
         {
            EndMiniGame();
            return;
         }

         CountToNextLaserThrow(deltaTime);
      }

      public override void StartMiniGame()
      {
         base.StartMiniGame();
         this.StartUpdate();

         RPC_StartMiniGameDurationTimer();
      }
```

```
    protected override void EndMiniGame()
    {
        base.EndMiniGame();
        this.StopUpdate();

        _miniGameDurationCountdownTimer.BreakCountdown();
    }

    private void CountToNextLaserThrow(float deltaTime)
    {
        _timeToSpawnNewLasersPairRemainder -= deltaTime;

        if (_timeToSpawnNewLasersPairRemainder <= 0)
        {
            ThrowLasers();
        }
    }

    private void ThrowLasers()
    {
        _timeToSpawnNewLasersPairRemainder = _timeToSpawnNewLasersPair;
        int randomBorderIndex = Random.Range(0, _arenaBorders.Length);
        int lasersCount = _lasersThrowCountRandomizer.GetRandomLasersCountToThrow();
        _arenaBorders[randomBorderIndex].ActivateLasers(lasersCount);
    }

    [Rpc(RpcSources.StateAuthority, RpcTargets.All)]
    private void RPC_StartMiniGameDurationTimer()
    {
        Action endTimerAction = Runner.IsServer ? EndMiniGame : null;

        if (Runner.IsClient)
        {
            OnMiniGameEnded += _miniGameDurationCountdownTimer.BreakCountdown;
        }

        _miniGameDurationCountdownTimer.StartCountdown(_ancientLasersArenaSettings.MiniGamePlayingTime,
endTimerAction);
    }
  }
```

Даний клас має методи початку/закінчення міні-гри – Start/EndMiniGame. Метод CountToNextLaserThrow є таймером до наступної хвилі лазерів, який в метод OnSystemUpdate в кожному кадрі відраховує час таймера. Коли таймер доходить до нуля – час запускати хвилю лазерів методом ThrowLasers. Метод RPC_StartMiniGameDurationTimer викликається на старті міні гри і запускає зворотній таймер на хвилину, по закінченню якого, якщо хтось вижив закінчиться гра.

10 Часткова реалізація класу FallingFloorsGridEditorGenerator (генерація плиток ігрової зони)

```
private void GenerateFloorsGrid()
{
    DeleteOldGrid();

    for (int column = 1; column <= _columnsCount; column++)
    {
        for (int row = 1; row <= _rowsCount; row++)
        {
            CreateFloorCell(column, row);
        }
    }

    _floorDestructor.SetNewFloorCells(GetComponentsInChildren<FallingFloorCell>());
}

private void CreateFloorCell(int column, int row)
{
    Vector3 floorCellPos = GetFloorCellPosition(column, row);
    Vector3 floorSize = new Vector3(_xFloorCellSize, _yFloorCellSize, _zFloorCellSize);

    GameObject floorCell = PrefabUtility.InstantiatePrefab(_floorCellPrefab, transform) as GameObject;
    int floorCellId = (column - 1) * _rowsCount + row;

    FallingFloorCell fallingFloorCell = floorCell.GetComponent<FallingFloorCell>();
    fallingFloorCell.SetId(floorCellId);
    fallingFloorCell.SetSize(floorSize);
    floorCell.name = "Cell " + column + "_" + row;
    floorCell.transform.localPosition = floorCellPos;
}

private Vector3 GetFloorCellPosition(int column, int row)
{
    float totalXOffset = (row - 1) * _xOffsetBetweenCells;
    float totalZOffset = (column - 1) * _zOffsetBetweenCells;
    float cellsSizeX = (row - 1) * _xFloorCellSize + _xFloorCellSize / 2;
    float cellsSizeZ = (column - 1) * _zFloorCellSize + _zFloorCellSize / 2;

    return new Vector3(cellsSizeX + totalXOffset, 0, cellsSizeZ + totalZOffset);
}
```

Метод GenerateFloorsGrid проходить циклом по кожній комірці сітки і створює там плитку методом CreateFloorCell. Метод GetFloorCellPosition надає світову позицію, приймаючи координати комірки в сітці.

Скрипт самостійно встановлює розміри плиток, їх позиції, відступи між ними на основі раніше заповнених даних розробником.