Міністерство освіти і науки України
Університет митної справи та фінансів

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

Кваліфікаційна робота бакалавра

на тему «Створення ігрового додатку складної архітектури в середовищі Unity з використанням фрейм-ворків»

Виконав: студент групи ___К19-1_____
Спеціальність 122 «Комп'ютерні науки»
___Стасовський Антон Вячеславович_____

Керівник _____к.т.н., доцент Чупілко Т.А.
Рецензент_____Дніпропетровський___
державний університет внутрішніх справ
(місце роботи)
_доцент кафедри інформаційних технологій
(посада)
_к. т. н., доцент Насонова С.С._____
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2023

АНОТАЦІЯ

Стасовський А.В. Створення ігрового додатку складної архітектури в середовищі Unity з використанням фрейм-ворків.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю 122 «Комп'ютерні науки». – Університет митної справи та фінансів, Дніпро, 2023.

Пояснювальна записка: 109 с., 12 рисунків, 21 джерело.

Метою роботи є дослідження створення ігрового додатку зі складною архітектурою в середовищі Unity з використанням фреймворків. Робота зосереджена на розробці гри, яка демонструє передові архітектурні патерни та принципи для покращення модульності, масштабованості та супроводжуваності.

У дослідженні вивчаються різні фреймворки, доступні для розробки ігор на Unity, в тому числі для систем типу "сутність-компонент", ін'єкції залежностей, управління станами. Використовуючи ці фреймворки, робота має на меті досягти гнучкої та розширюваної архітектури, яка сприяє повторному використанню коду та полегшує ітеративний процес розробки.

Основними результатами дослідження є розробка та реалізація ігрового додатку, який демонструє застосування цих фреймворків в Unity. Робота демонструє переваги використання фреймворків для спрощення розробки, покращення організації коду та підвищення загальної продуктивності гри.

Крім того, дослідження розглядає наукову новизну застосування цих фреймворків у середовищі Unity, роблячи внесок в існуючий масив знань про архітектуру розробки ігор. Практична цінність полягає в тому, що розробники отримують уявлення про використання фреймворків для ефективного створення складних ігрових додатків.

Ключові слова: Unity, розробка ігор, фреймворки, об'єктно-компонентна система, ін'єкція залежностей, управління станами, модульність, масштабованість, повторне використання коду, ітеративна розробка.

ABSTRACT

Stasovskyi A.V. Creating a game application of complex architecture in the Unity environment using frameworks.

Qualification work for the bachelor’s degree in specialty 122 "Computer Science." – University of Customs and Finance, Dnipro, 2023.

Explanatory note: 109 p., 12 figures, 21 references.

The aim of the work is to study the creation of a game application with a complex architecture in the Unity environment using frameworks. The project focuses on developing a game that demonstrates advanced architectural patterns and principles to improve modularity, scalability, and maintainability.

The research explores various frameworks available for developing games in Unity, including entity-component systems, dependency injection, and state management. By using these frameworks, the project aims to achieve a flexible and extensible architecture that promotes code reuse and facilitates an iterative development process.

The main results of the research are the development and implementation of a game application that demonstrates the use of these frameworks in Unity. The project demonstrates the benefits of using frameworks to simplify development, improve code organization, and increase overall game performance.

In addition, the research considers the scientific novelty of using these frameworks in the Unity environment, contributing to the existing body of knowledge about game development architecture. The practical value lies in the fact that developers get an idea of how to use frameworks to effectively create complex game applications.

Keywords: Unity, game development, frameworks, object-oriented system, dependency injection, state management, modularity, scalability, code reuse, iterative development.

ЗМІСТ

ВСТУП

Останніми роками сфера розробки ігор зазнала значного прогресу, оскільки зростає попит на складні та захоплюючі ігрові процеси. Оскільки популярність ігрових додатків продовжує зростати, розробники стикаються з проблемою створення ігор з надійною архітектурою, яка може ефективно справлятися з усіма складнощами, пов’язаними з ними.

Предметом дослідження даної роботи є розробка ігрового додатку зі складною архітектурою в середовищі Unity з використанням фреймворків.

Об'єктом дослідження цієї роботи є архітектурний дизайн та реалізація ігрового додатку, який демонструє модульність, масштабованість та ремонтопридатність, при цьому ефективно справляючись зі складнощами, пов'язаними з сучасною розробкою ігор.

Мета цієї роботи – дати уявлення про розробку ігор зі складною архітектурою, підкресливши актуальність і важливість пошуку ефективних рішень проблем, з якими стикаються розробники у сфері розробки ігор, що постійно розвивається.

Ціллю є розробка надійного та масштабованого ігрового додатку, який відповідає викликам сучасної ігрової розробки.

Актуальність цієї роботи полягає в подоланні розриву між дослідженнями в галузі архітектури розробки ігор та практичною реалізацією. Використовуючи фреймворки та принципи проектування, ця робота має на меті зробити внесок у розвиток практики розробки ігор та надати уявлення про створення складних ігрових архітектур.

Практична новизна роботи полягає в її потенційному застосуванні в митних органах та інших установах, де необхідна розробка програмного забезпечення для розваг або навчання користувачів. Розроблений ігровий додаток з покращеними технічними характеристиками може запропонувати цікавий та захоплюючий користувацький досвід, забезпечуючи при цьому гнучку та підтримувану кодову базу для майбутніх удосконалень.

В контексті подібних робіт ця дипломна робота виділяється тим, що демонструє використання фреймворків і передових принципів дизайну для створення складного ігрового додатку. Поєднуючи теоретичні знання з практичною реалізацією, ця робота робить свій внесок у розвиток архітектури розробки ігор Unity і є цінним ресурсом для майбутніх досліджень і розробок у цій галузі.

Структура роботи:

– вступ;
– 3 розділи;
– висновки;
– список вокористаних джерел;
– 1 додаток.

Робота спрямована на вирішення проблеми шляхом дослідження створення ігрового додатку зі складною архітектурою в середовищі Unity з використанням фреймворків. Вибір цієї теми дослідження обґрунтований необхідністю подолання розриву між сучасними дослідженнями в галузі розробки ігор та практичною реалізацією складних архітектур.

Завдяки критичному аналізу стає очевидним, що сучасна розробка ігор потребує архітектур, які сприяють модульності, масштабованості та супроводжуваності. Традиційні монолітні підходи часто призводять до створення коду, який важко підтримувати, якому бракує гнучкості та який обмежує майбутні ітерації. Тому суть проблеми полягає в розробці ігрового додатку з архітектурою, яка враховує ці обмеження, дозволяє ефективно розробляти і забезпечує основу для майбутніх удосконалень.

Для забезпечення достовірності результатів і висновків, отриманих у цій кваліфікаційній роботі, було використано комплекс *методів дослідження*. Кожен метод був ретельно відібраний на основі його придатності для вивчення конкретних аспектів теми дослідження.

1. Аналіз літературних джерел. Було проведено всебічний огляд літератури для вивчення існуючих досліджень, фреймворків та найкращих практик в

галузі архітектури розробки ігор. Цей метод включав глибокий аналіз наукових статей, книг та онлайн-ресурсів, пов’язаних з розробкою ігор на Unity, складними архітектурами та фреймворками. Завдяки огляду літератури було створено міцний фундамент, який дозволив критично осмислити предметну область та вибрати відповідні фреймворки для реалізації.

2. Експериментальна розробка. Для проектування та розробки ігрового додатку зі складною архітектурою було застосовано експериментальний підхід. Це включало реалізацію різних архітектурних патернів, фреймворків та принципів проектування, визначених на етапі огляду літератури. Шляхом ітеративної розробки різні комбінації фреймворків були протестовані та оцінені на предмет їхньої ефективності у підвищенні модульності, масштабованості та супроводжуваності.
3. Аналіз продуктивності. Аналіз продуктивності був проведений для оцінки впливу впроваджених фреймворків на продуктивність ігрового додатку. Це включало вимірювання ключових показників продуктивності, таких як частота кадрів, використання пам’яті та час завантаження. Порівняння продуктивності ігрового додатку з впровадженими фреймворками та без них дозволило об’єктивно оцінити ефективність обраного архітектурного підходу.

Вибір цих методів дослідження забезпечує достовірність отриманих результатів і висновків кількома способами. Метод огляду літератури гарантує, що робота ґрунтується на існуючих дослідженнях та усталених фреймворках, забезпечуючи міцне теоретичне підґрунтя. Метод експериментальної розробки дозволяє здійснювати практичну реалізацію та ітеративне вдосконалення, гарантуючи, що висновки ґрунтуються на реальних сценаріях і досвіді. Нарешті, метод аналізу ефективності надає кількісні дані для підтримки оцінки архітектурного підходу, сприяючи об’єктивності та надійності висновків роботи.

Використовуючи ці методи наукового дослідження, ця дипломна робота не тільки досліджує теоретичні аспекти складної ігрової архітектури, але й перевіряє їх практичне застосування через реалізацію та аналіз продуктивності. Поєднання цих методів підвищує надійність результатів та дозволяє зробити змістовні висновки, що сприятиме подальшому пізнанню та розумінню архітектури розробки ігор у середовищі Unity.

Продукт, який буде розроблено, матиме кілька ключових технічних характеристик, що сприятимуть його загальній якості та продуктивності.

1. Модульна архітектура. Ігровий додаток матиме модульну архітектуру, яка сприяє організації коду та його повторному використанню. Завдяки використанню фреймворків для компонентної системи та ін’єкції залежностей, кодову базу буде розділено на зв’язні та незалежні модулі, що полегшить обслуговування, масштабування та майбутні вдосконалення.
2. Масштабованість. Архітектурний дизайн надаватиме пріоритет масштабованості, що дозволить ігровому додатку обробляти зростаючу кількість функцій, ресурсів та взаємодій без шкоди для продуктивності. Завдяки використанню фреймворків для управління станом гри та штучного інтелекту, додаток демонструватиме ефективну масштабованість, забезпечуючи плавний ігровий процес навіть зі складною ігровою механікою.
3. Функціональність. Впроваджена архітектура підкреслюватиме зручність обслуговування, полегшуючи команді розробників можливість оновлювати, модифікувати та додавати нові функції до ігрового додатку. Впровадження фреймворків, які сприяють практикам чистого коду, поділу завдань та розмежуванню функцій, призведе до створення підтримуваної кодової бази, що зменшить ризик внесення помилок та полегшить адаптацію нових членів команди.

Очікуваний технічний ефект від впровадження такої складної архітектури – створення надійного та розширюваного ігрового додатку. Модульний дизайн підвищить ефективність розробки, оскільки окремі

компоненти можуть розроблятися і тестуватися ізольовано. Масштабованість гарантує, що ігровий додаток може рости і адаптуватися до вимог, що змінюються, забезпечуючи основу для майбутніх розширень або оновлень. Крім того, зосередженість на підтримці зменшить технічний борг і дозволить швидше проводити ітерації та виправляти помилки, покращуючи загальний процес розробки.

З економічної точки зору, впровадження добре продуманої та ефективної архітектури може мати кілька позитивних ефектів. Модульна і масштабована природа ігрового додатку скорочує час і зусилля на розробку, що потенційно призводить до економії коштів. Зручність підтримки кодової бази знижує довгострокові витрати на обслуговування і полегшує майбутні оновлення або ітерації. Крім того, ігровий додаток з покращеними технічними характеристиками має потенціал для залучення більшої кількості користувачів, що призводить до збільшення доходів і конкурентоспроможності на ринку.

РОЗДІЛ 1.
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Опис завдання

Основним завданням є розробка багатокористувацького 3D-платформера зі складною архітектурою в середовищі Unity з використанням сучасних фреймворків. Ігровий додаток фокусується на забезпеченні розважального та захоплюючого досвіду для користувачів, демонструючи при цьому використання передових архітектурних патернів та фреймворків.

Концепція гри

Створити багатокористувацький 3D-платформер з головоломками для двох гравців. Розробити візуально привабливі рівні зі складними головоломками та елементами платформера. Реалізувати механіки, які заохочують кооперацію та координацію між двома гравцями.

Архітектура

Розробити модульну архітектуру, яка сприяє організації коду, повторному використанню та підтримці. Використати фреймворк Zenject для ін’єкції залежностей, щоб керувати композицією об’єктів та полегшити роз’єднання ігрових компонентів. Інтегрувати фреймворк PUN 2 для безперебійної багатокористувацької функціональності, що дозволяє гравцям підключатися та взаємодіяти в реальному часі. Інтегрувати DOTween для плавних і візуально приємних анімацій та переходів. Використати UniRx для реалізації реактивного програмування, щоб забезпечити гнучку та керовану подіями ігрову логіку.

Багатокористувацька функціональність

Інтегрувати фреймворк PUN 2 для мережевої синхронізації та підбору гравців. Забезпечити безперешкодну взаємодію гравців, наприклад, спільне розв’язання головоломок та синхронізоване пересування платформою.

Користувацький інтерфейс та користувацький досвід

Розробити інтуїтивно зрозумілий та візуально привабливий користувацький інтерфейс для меню, ігрового HUD та взаємодії гравців. Реалізувати систему підрахунку очок та елементи змагання для покращення ігрового досвіду. Реалізувати механізми зворотного зв'язку, щоб надати гравцям чітку інформацію про їхні дії та прогрес. Забезпечити безперебійну та чуйну роботу користувача завдяки ефективній обробці вводу та адаптивному управлінню.

Тестування та оптимізація

Провести ретельне тестування для виявлення та виправлення помилок, забезпечивши стабільний та відшліфований ігровий досвід. Оптимізувати продуктивність ігрового додатку, враховуючи такі фактори, як затримка в мережі, частота кадрів і використання пам'яті.

1.2 Опис предметної області і напрямків дослідження

Опис предметної області

Предметна область цієї дипломної роботи лежить на перетині розробки ігор, складної архітектури та сучасних фреймворків. Вона охоплює вивчення та застосування архітектурних патернів, фреймворків та технологій у середовищі Unity для створення багатокористувацької 3D-платформерної гри.

Предметна область заглиблюється у виклики, з якими стикаються розробники ігор при проектуванні та впровадженні складних архітектур, що підвищують модульність, масштабованість та ремонтопридатність. Він досліджує інтеграцію сучасних фреймворків, таких як Zenject, PUN 2, DOTween та UniRx, для оптимізації розробки, покращення організації коду та підвищення загальної продуктивності ігрового додатку.

Напрямки досліджень

Архітектура розробки ігор:

– дослідження різних архітектурних патернів та принципів, що застосовуються до розробки ігор;
– аналіз переваг та недоліків різних архітектурних підходів у контексті багатокористувацьких 3D-платформерів;
– визначення стратегій для покращення організації коду, повторного використання та супроводу у складних ігрових проектах.

Фреймворки та технології:

– дослідження та оцінка можливостей та особливостей сучасних фреймворків, зокрема Zenject, PUN 2, DOTween та UniRx;
– вивчення інтеграції та використання цих фреймворків у середовищі Unity для покращення розробки ігор;
– оцінка впливу фреймворків на технічні характеристики, продуктивність та користувацький досвід ігрового додатку.

Багатокористувацька функціональність:

– дослідження викликів та міркувань, пов’язаних з реалізацією безшовної багатокористувацької функціональності в 3D-платформерній грі;
– вивчення методів мережевої синхронізації, алгоритмів підбору гравців та протоколів зв’язку в реальному часі.
– вивчення стратегій для просування кооперативного ігрового процесу та покращення багатокористувацького досвіду.

Інтерфейс користувача та користувацький досвід:

– дослідження принципів дизайну користувацького інтерфейсу та найкращих практик для багатокористувацьких ігор;
– вивчення методів забезпечення чіткого зворотного зв’язку та інтуїтивно зрозумілої взаємодії між гравцями;

– дослідження стратегій оптимізації користувацького досвіду, враховуючи обробку введення, швидкість реакції та візуальну естетику.

Тестування та оптимізація:

– дослідження методологій для ефективного тестування та налагодження багатокористувацьких ігрових додатків;
– вивчення методів оптимізації продуктивності мережевих ігор, включаючи зменшення затримок, оптимізацію пропускної здатності та управління синхронізацією;
– вивчення стратегій збору та аналізу відгуків користувачів для ітеративного покращення ігрового процесу, дизайну рівнів та загального користувацького досвіду.

1.3 Висновки до першого розділу

У цьому розділі представлено опис завдання та предметної області дипломної роботи. Області дослідження включають архітектуру, фреймворки та технології розробки ігор, багатокористувацьку функціональність, користувацький інтерфейс та дизайн користувацького досвіду, а також тестування та оптимізацію. Звертаючись до цих сфер, робота прагне сприяти розумінню та практичному застосуванню складних ігрових архітектур та сучасних фреймворків. Результатом успішного завершення роботи стане повнофункціональний ігровий додаток, який продемонструє переваги цих підходів у створенні захопливого багатокористувацького досвіду.

РОЗДІЛ 2.
ІСНУЮЧІ РОЗВ’ЯЗКИ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Аналіз і характеристика об’єкта проектування

Об’єктом проектування цієї дипломної роботи є багатокористувацький 3D-платформер з акцентом на складну архітектуру та використання сучасних фреймворків. Гра покликана забезпечити захоплюючий та захоплюючий досвід для гравців, поєднуючи виклики платформера з елементами кооперативного вирішення головоломок.

Основна мета об’єкта проектування – створити інтерактивне та динамічне багатокористувацьке середовище, яке сприяє співпраці та координації між двома гравцями. Гра матиме візуально привабливі та хитромудро розроблені рівні, що представлятимуть гравцям різноманітні перешкоди та головоломки для подолання.

Архітектура ігрового додатку є важливим аспектом об’єкта проектування. Вона спрямована на підвищення модульності, масштабованості та ремонтопридатності, гарантуючи, що кодова база залишається організованою та легко розширюваною. Обраний архітектурний підхід фокусується на використанні фреймворків, які полегшують розділення коду та сприяють повторному використанню.

Багатокористувацька функціональність є фундаментальним аспектом об’єкту проектування. Безшовна інтеграція багатокористувацьких функцій забезпечує комунікацію та синхронізацію між гравцями в реальному часі, дозволяючи їм брати участь у кооперативному геймплеї та разом вирішувати головоломки. Реалізація багатокористувацької функціональності вимагає надійної мережевої синхронізації та алгоритмів підбору гравців, щоб забезпечити безперебійну та приємну багатокористувацьку гру.

Дизайн інтерфейсу користувача ігрового додатку є ще одним важливим моментом. Він прагне надати гравцям інтуїтивно зрозумілий і візуально

привабливий інтерфейс, який покращує їхню взаємодію та загальний користувацький досвід. Чіткі та інформативні механізми зворотного зв’язку допомагають гравцям у проходженні гри та дають відчуття успіху після виконання завдань.

2.2 Обґрунтування оптимального варіанта реалізації мети кваліфікаційної роботи

Для реалізації мети розробки багатокористувацького 3D-платформера зі складною архітектурою було ретельно розглянуто різні доступні варіанти. Після ретельного аналізу обраний підхід поєднує в собі кілька факторів, які роблять його найкращим варіантом для досягнення цілей роботи.

Рішення розробити багатокористувацький ігровий додаток відповідає меті забезпечити цікавий та інтерактивний досвід для гравців. Уможливлюючи співпрацю в режимі реального часу та спільний ігровий процес, обраний варіант підвищує загальне задоволення від гри та її цінність для повторного проходження.

Використання складної архітектури має важливе значення для створення надійного, масштабованого та підтримуваного ігрового додатку. Такий вибір дозволяє ефективно управляти організацією коду і сприяє повторному використанню, забезпечуючи добре структуровану і розширювану кодову базу. Використання сучасних фреймворків ще більше зміцнює архітектуру, надаючи інструменти та бібліотеки, які полегшують реалізацію ключових функціональних можливостей та підвищують ефективність розробки.

Обраний варіант 3D-платформера забезпечує багате та захоплююче середовище для гравців. Платформерні завдання та елементи спільного розв’язання головоломок сприяють захопливому ігровому процесу, який приваблює широку аудиторію.

Акцент на оптимізації та тестуванні в рамках обраного підходу гарантує, що ігровий додаток добре працює на різних платформах і пристроях. Така

увага до продуктивності та стабільності гарантує безперебійний та приємний користувацький досвід, що сприяє загальному успіху роботи.

Враховуючи ці фактори, обраний варіант розробки багатокористувацького 3D-платформерного ігрового додатку зі складною архітектурою, з використанням сучасних фреймворків, виявляється найкращим підходом для реалізації цілей кваліфікаційної роботи. Цей варіант поєднує в собі елементи захопливого геймплею, надійної архітектури, масштабованості, ремонтопридатності та оптимізації, створюючи високоякісний ігровий додаток, який відповідає цілям роботи та забезпечує захоплюючий та приємний досвід для гравців.

2.3 Опис алгоритму і програмного забезпечення

Обґрунтування вибору C#, як мови програмування

Вибір мови програмування C# є цілком виправданим для реалізації алгоритму та програмного забезпечення в даній дипломній роботі.

C# є офіційною мовою рушія розробки ігор Unity, що є обраним середовищем для розробки багатокористувацької 3D-платформерної гри. Використовуючи C#, розробники можуть скористатися широкими можливостями та функціоналом Unity, включаючи потужний набір інструментів, бібліотек та API, спеціально розроблених для розробки ігор. Така тісна інтеграція між C# та Unity забезпечує бездоганну сумісність та оптимальну продуктивність.

C# - це об’єктно-орієнтована мова програмування з високим рівнем абстракції, що робить її добре придатною для розробки складного програмного забезпечення. Багатий набір мовних можливостей, таких як класи, інтерфейси та успадкування, сприяє організації коду, модульності та багаторазовому використанню. Ці характеристики мають вирішальне значення для розробки складної архітектури та підтримки структурованої і зручної для підтримки кодової бази.

C# широко відома своєю потужною системою типів, яка підвищує надійність та безпеку коду. Мова забезпечує сувору типізацію, зменшуючи ймовірність помилок під час виконання та забезпечуючи кращу перевірку коду в процесі розробки. Цей аспект особливо важливий при реалізації багатокористувацької функціональності та мережевої синхронізації, оскільки допомагає забезпечити узгодженість і надійність комунікації між гравцями.

Ще однією перевагою використання C# є велика та активна спільнота розробників. Ця спільнота надає безліч ресурсів, навчальних посібників та форумів, присвячених розробці ігор на C#. Доступ до цієї бази знань полегшує вирішення проблем, пришвидшує час розробки та заохочує до співпраці з іншими розробниками.

C# пропонує чудову підтримку асинхронного програмування завдяки таким функціям, як async/await, що має вирішальне значення для обробки взаємодії в реальному часі, мережевого зв’язку та швидкості реагування в багатокористувацькому ігровому додатку.

Універсальність C# виходить за межі розробки ігор. Це універсальна мова, яка широко використовується в різних сферах, включаючи веб-розробку, корпоративне програмне забезпечення та розробку мобільних додатків. Ця універсальність відкриває можливості для майбутнього розширення та потенційної інтеграції з іншими програмними компонентами або платформами.

Враховуючи ці причини, вибір мови програмування C# є виправданим для реалізації алгоритму та програмного забезпечення в даній дипломній роботі. Тісна інтеграція з Unity, сильні об’єктно-орієнтовані можливості, сувора типізація, активна підтримка спільноти та універсальність роблять її чудовою для розробки складного багатокористувацького 3D-платформера з надійною архітектурою та високими характеристиками продуктивності.

Обґрунтування вибору Unity, як середовища розробки

Вибір Unity як середовища розробки ігор в даній дипломній роботі є цілком виправданим через декілька ключових факторів, які роблять його ідеальним вибором.

Unity – це широко відома та стандартна в індустрії платформа для розробки ігор, яка пропонує повний набір інструментів, функцій та ресурсів. Це потужне та інтуїтивно зрозуміле середовище розробки, яке спрощує створення 2D та 3D ігор, включаючи розробку багатокористувацької функціональності. Візуальний редактор Unity дозволяє швидко створювати прототипи та ітерації, що дає змогу розробникам швидко втілювати свої ідеї в життя.

Unity пропонує відмінну крос-платформну сумісність, що дозволяє розгортати ігровий додаток на різних платформах з мінімальними додатковими зусиллями при розробці. Незалежно від того, чи орієнтовані вони на ПК, консолі, мобільні пристрої або навіть платформи віртуальної реальності, Unity надає надійну підтримку та можливості оптимізації для різних платформ. Така універсальність забезпечує ширше охоплення та потенційну базу користувачів для розробленого ігрового додатку.

Велике сховище ресурсів та екосистема Unity надає розробникам величезну бібліотеку готових ресурсів, скриптів та плагінів. Це багате на ресурси середовище прискорює час розробки та полегшує реалізацію складних функцій і можливостей. Доступність плагінів та фреймворків для розробки багатокористувацьких ігор, таких як Photon Unity Networking (PUN), підвищує простоту та ефективність реалізації багатокористувацької функціональності в ігровому додатку.

Ще однією значною перевагою Unity є потужна підтримка спільноти. Спільнота Unity є великою, активною та різноманітною, пропонуючи безліч ресурсів, форумів та навчальних посібників. Розробники можуть звертатися до цієї спільноти за порадами, співпрацювати та вчитися на досвіді інших. Активна спільнота Unity забезпечує доступ до постійних оновлень,

виправлень та покращень, що сприяє стабільності та надійності розробленого програмного забезпечення.

Unity надає якісну документацію та навчальні ресурси, що робить її доступною для розробників різного рівня кваліфікації. Наявність офіційних навчальних посібників, онлайн-курсів та документації дозволяє розробникам швидко вивчити та освоїти Unity, що дає їм змогу ефективно реалізовувати бажані алгоритми та функції програмного забезпечення.

Вбудована підтримка 2D та 3D графіки, фізики, анімації та аудіо в Unity спрощує реалізацію захоплюючого та візуально привабливого ігрового процесу. Інтеграція цих функцій в Unity дозволяє безперешкодно імпортувати та редагувати ресурси, зменшуючи потребу в зовнішніх інструментах або складних робочих процесах.

2.4 Вибір і обґрунтування структури проектування системи та її компонентів

Photon Unity Networking як фреймворк для багатокористувацьких ігор у реальному часі

Вибір Photon Unity Networking в якості фреймворку для багатокористувацької функціональності в реальному часі в цій дипломній роботі виправданий кількома ключовими причинами, які роблять його чудовим вибором.

Photon Unity Networking спеціально розроблений для розробки ігор на Unity, що робить його бездоганно інтегрованим з рушієм Unity. Ця тісна інтеграція спрощує реалізацію багатокористувацької функціональності, оскільки Photon Unity Networking надає високорівневий API, який абстрагується від значної частини складнощів, пов’язаних з мережевою взаємодією. Розробники можуть зосередитися на геймплеї та механіці, а не на низькорівневому мережевому програмуванні.

Photon Unity Networking пропонує надійні та ефективні можливості мережевої синхронізації, забезпечуючи плавний та послідовний ігровий процес для всіх гравців. Він обробляє такі важливі аспекти, як компенсація затримок, інтерполяція та компенсація затримок, забезпечуючи надійний та чуйний багатокористувацький досвід. Ці функції мають вирішальне значення для багатокористувацьких ігор у реальному часі, оскільки вони мінімізують вплив мережевих затримок і гарантують, що всі гравці синхронізовані.

Ще однією значною перевагою Photon Unity Networking є його масштабованість. Він підтримує як малі, так і великі багатокористувацькі налаштування, що охоплюють широкий спектр типів ігор та кількості гравців. Незалежно від того, чи це кооперативна гра для двох гравців, чи масова багатокористувацька онлайн гра, Photon Unity Networking може масштабуватися відповідно до вимог, забезпечуючи гнучке рішення для розробки багатокористувацьких ігор.

Обширна документація Photon Unity Networking та активна підтримка спільноти є додатковими факторами, які роблять його привабливим вибором. Фреймворк пропонує документацію, навчальні посібники та приклади, що дозволяє розробникам швидко зрозуміти його концепції та реалізацію.

Спільнота Photon Unity Networking є енергійною та активною, з форумами та ресурсами, доступними для пошуку допомоги та обміну знаннями. Така потужна підтримка спільноти забезпечує доступ до постійних оновлень, виправлень та покращень, що сприяє стабільності та надійності багатокористувацької функціональності.

Photon Unity Networking сумісна з різними платформами, включаючи ПК, консолі та мобільні пристрої, що дозволяє грати в багатокористувацьку гру на різних платформах. Ця сумісність розширює охоплення розробленого ігрового додатку, дозволяючи гравцям на різних пристроях безперешкодно підключатися і грати разом.

Простота використання Photon Unity Networking та можливості швидкого створення прототипів добре узгоджуються з ітеративним

характером розробки ігор. Він забезпечує простий робочий процес для створення багатокористувацької функціональності, дозволяючи розробникам швидко тестувати та ітерації своїх ігрових ідей. Цей гнучкий підхід до розробки сприяє ефективній ітерації та вдосконаленню багатокористувацького ігрового досвіду.

Zenject як фреймворк для ін'єкції залежностей

Рішення використовувати Zenject як фреймворк для ін'єкції залежностей у цій дипломній роботі обґрунтовано кількома ключовими причинами, які роблять його високоефективним вибором.

Zenject пропонує легкий та гнучкий підхід до ін'єкції залежностей, що дозволяє розробникам ефективно керувати залежностями. Дотримуючись принципів інверсії управління та ін'єкції залежностей, Zenject сприяє вільному зв'язку та модульності компонентів системи. Така модульна конструкція покращує ремонтопридатність та розширюваність кодової бази, оскільки залежності можна легко замінити або модифікувати, не впливаючи на всю систему.

Zenject надає простий та інтуїтивно зрозумілий API, що полегшує інтеграцію та роботу з ним у проектах Unity. Фреймворк використовує існуючі концепції Unity, такі як MonoBehaviours та ScriptableObjects, що дозволяє безперешкодно інтегруватися в рушій Unity. Ця інтеграція полегшує ін'єкцію залежностей у скрипти, ігрові об'єкти та інші компоненти Unity, спрощуючи процес розробки та зменшуючи кількість шаблонного коду.

Ще однією сильною стороною Zenject є підтримка інжекції конструкторів, інжекції властивостей та інжекції методів, що забезпечує гнучкість у тому, як залежності вбудовуються в компоненти. Інжекція конструктора забезпечує вирішення залежностей при створенні об'єкта, гарантуючи, що компоненти мають всі необхідні залежності з самого початку. Інжекція властивостей і методів дозволяє більш детально контролювати вирішення залежностей, дозволяючи компонентам отримувати залежності за потребою під час виконання.

Zenject також пропонує такі функції, як об’єднання об’єктів, що може значно підвищити продуктивність за рахунок повторного використання та переробки об’єктів замість того, щоб створювати та знищувати їх багаторазово. Ця функція особливо цінна у ресурсоємних сценаріях, таких як керування ігровими об’єктами або обробка ефектів частинок.

Zenject сприяє використанню інтерфейсів та абстракцій, що дозволяє легко імітувати та тестувати об’єкти. Покладаючись на інтерфейси, розробники можуть створювати тестований код, який відокремлений від конкретних реалізацій. Цей аспект покращує тестованість системи, дозволяючи реалізовувати модульні тести і полегшуючи виявлення потенційних проблем або помилок на ранній стадії циклу розробки.

Zenject надає чудову документацію та підтримує спільноту [19]. Документація фреймворку вичерпна і добре підтримується, пропонуючи детальні пояснення, навчальні посібники та приклади, які допоможуть розробникам ефективно використовувати Zenject. Активна спільнота навколо Zenject забезпечує доступ до форумів, дискусій та додаткових ресурсів, що дозволяє розробникам шукати настанови, ділитися знаннями та співпрацювати над вирішенням проблем.

Вибір патерну проектування MVP

Рішення про використання патерну MVP (Model-View-Presenter) (рисунок 2.1) в даній дипломній роботі обґрунтовано кількома ключовими причинами, які роблять його ідеальним вибором для структури проектування системи.

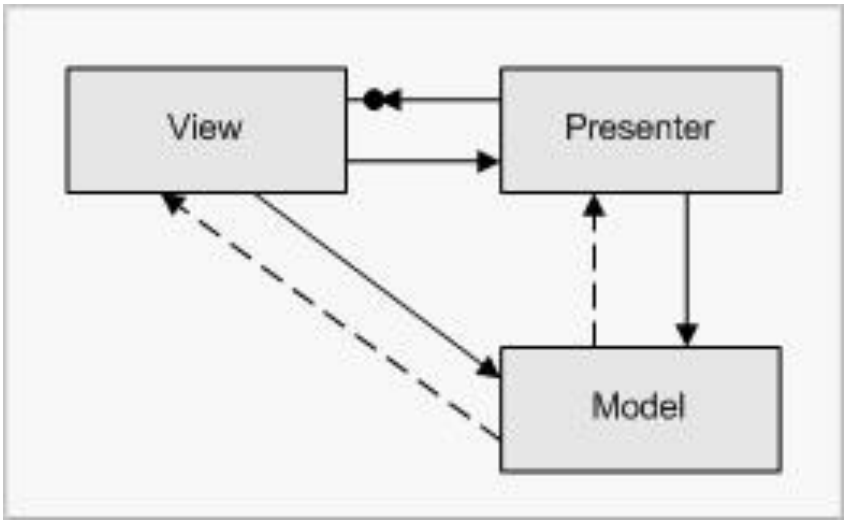

Рисунок 2.1 – Патерн проектування MVP

Модель MVP розділяє проблеми і сприяє чіткому розподілу обов'язків між різними компонентами системи. Модель представляє дані та бізнес-логіку, представлення обробляє користувацький інтерфейс та презентацію, а доповідач виступає в ролі посередника, полегшуючи комунікацію між моделлю та представленням. Такий поділ забезпечує кращу організацію коду, модульність і ремонтопридатність, оскільки кожен компонент має конкретну і чітко визначену роль.

Дотримуючись патерну MVP, розробники можуть досягти високого ступеня керованості. Розділення завдань дозволяє проводити модульне тестування окремих компонентів, таких як модель і доповідач, ізольовано один від одного. Це полегшує написання комплексних тестових кейсів, перевірку правильності логіки та виявлення потенційних проблем або помилок на ранніх стадіях розробки.

Ще однією перевагою патерну MVP є його підтримка розширюваності та гнучкості. Завдяки чіткому поділу між моделлю, представленням і презентатором, стає легше модифікувати або замінити будь-який компонент, не впливаючи на інші. Така гнучкість особливо цінна при розробці ігор, де вимоги та функції часто розвиваються і змінюються з часом. Патерн MVP дозволяє легко адаптуватися до нових вимог і спрощує процес інтеграції додаткових функцій в систему.

Також патерн MVP сприяє створенню чистої та зручної для підтримки кодової бази. Розподіл завдань і модульна структура компонентів роблять код більш організованим і читабельним. Це покращує співпрацю між розробниками та підвищує ремонтопридатність коду, оскільки стає легше знаходити та розуміти конкретні функціональні можливості в системі.

Патерн MVP також підтримує повторне використання. Відокремлюючи бізнес-логіку від користувацького інтерфейсу, модель і презентатор можуть бути повторно використані в різних представленнях або інтерфейсах. Таке багаторазове використання зменшує дублювання коду і сприяє більш ефективному процесу розробки.

Патерн MVP добре узгоджується з середовищем розробки ігор Unity. Компонентна архітектура Unity природно поєднується з патерном MVP, оскільки кожен компонент може відповідати певній ролі в тріаді MVP. Ця інтеграція дозволяє безперешкодно інтегрувати патерн MVP у проекти Unity, полегшуючи процес розробки та уможливлюючи кращу організацію та управління ігровою логікою та взаємодією з користувацьким інтерфейсом.

2.5 Висновки до другого розділу

Отже, аналіз та характеристика об’єкту проектування, разом з обґрунтуванням обраних варіантів, забезпечили міцний фундамент для реалізації мети кваліфікаційної роботи. Опис алгоритму та програмного забезпечення висвітлює використання відповідних фреймворків та мов програмування для досягнення бажаної функціональності та ефективності. Крім того, вибір та обґрунтування структури проектування, включаючи прийняття MVP-паттерна проектування та вибір конкретних фреймворків, забезпечують добре організовану, модульну та підтримувану систему. Ці міркування в сукупності сприяють успішній реалізації дипломної роботи, в результаті чого було створено надійний та ефективний ігровий додаток зі складною архітектурою та сучасними фреймворками.

РОЗДІЛ 3.
ВИРІШЕННЯ ПОСТАВЛЕНОЇ В КВАЛІФІКАЦІЙНІЙ РОБОТІ ЗАДАЧІ

3.1 Основні рішення з реалізації системи та її компонентів

Реалізація патерну MVP

Було прийнято кілька ключових рішень щодо реалізації патерну MVP (Model-View-Presenter), що забезпечило ефективне та узгоджене проектування компонентів системи.

Одне з основних рішень – встановити чіткий розподіл завдань між моделлю, представленням і презентатором. Модель представляє дані та бізнес-логіку, подання відповідає за інтерфейс користувача та презентацію, а доповідач виступає посередником між моделлю та поданням, полегшуючи комунікацію та координацію. Такий поділ дозволяє здійснювати модульну розробку, оскільки кожен компонент може бути розроблений і протестований незалежно, що сприяє повторному використанню коду і його підтримці.

Іншим важливим рішенням є визначення чітко визначених контрактів та інтерфейсів між моделлю, представленням та доповідачем. Завдяки визначенню чітких інтерфейсів взаємодія між компонентами стає стандартизованою, що дозволяє безперешкодно співпрацювати та зменшує залежність. Ці контракти надають схему для комунікації і гарантують, що кожен компонент розуміє свої обов’язки і взаємодію в рамках тріади MVP.

Було прийнято рішення надати пріоритет ін’єкції залежностей як методу управління залежностями між компонентами. Використовуючи фреймворк для ін’єкції залежностей, такий як Zenject, залежності можна легко вбудовувати в presenter, зменшуючи тісний зв’язок і підвищуючи гнучкість.

```
public class MainSceneInstaller : MonoInstaller
   {
      public override void InstallBindings()
      {
```

```
            BindScreens();
        }

        private void BindScreens()
        {
            Container.BindViewAndPresenter<MainMenuScreenView,
MainMenuScreenPresenter>();
            Container.BindViewAndPresenter<LobbyPopupView,
LobbyPopupPresenter>();
        }
    }
```

Такий підхід забезпечує вільний зв’язок між компонентами, що полегшує модифікацію, заміну або розширення функціональності, не впливаючи на всю систему [16].

Крім того, програмування, кероване подіями, було обрано як найкращий механізм зв’язку між компонентами. Події дозволяють роз’єднати комунікацію, коли один компонент може повідомляти інші про певні дії або зміни стану, не вимагаючи прямих посилань. Таке розділення підвищує модульність, розширюваність та тестуємість системи.

```
public class MainMenuScreenView : MonoBehaviour
    {
        [SerializeField] private Button multiplayerButton;

        public event Action ClickedMultiplayerButton;

        private void Awake()
        {
            multiplayerButton.OnClickAsObservable().Subscribe(delegate {
ClickedMultiplayerButton?.Invoke(); })
                .AddTo(this);
```

```
        }
    }

    public class MainMenuScreenPresenter : IInitializable
    {
        private readonly MainMenuScreenView _view;
        private readonly ScreenNavigationSystem _screenNavigationSystem;

        public MainMenuScreenPresenter(MainMenuScreenView view,
ScreenNavigationSystem screenNavigationSystem)
        {
            _view = view;
            _screenNavigationSystem = screenNavigationSystem;
        }

        public void Initialize()
        {
            _view.ClickedMultiplayerButton += delegate {
_screenNavigationSystem.Show<LobbyPopupView>(); };
        }
    }
```

Також було прийнято рішення використовувати компонентну архітектуру Unity для реалізації патерну MVP. Компоненти Unity, такі як MonoBehaviours (рисунок 3.1) та ScriptableObjects, добре узгоджуються з компонентами MVP (модель, подання, доповідач), забезпечуючи безшовну інтеграцію та використання вбудованої функціональності Unity для полегшення розробки.

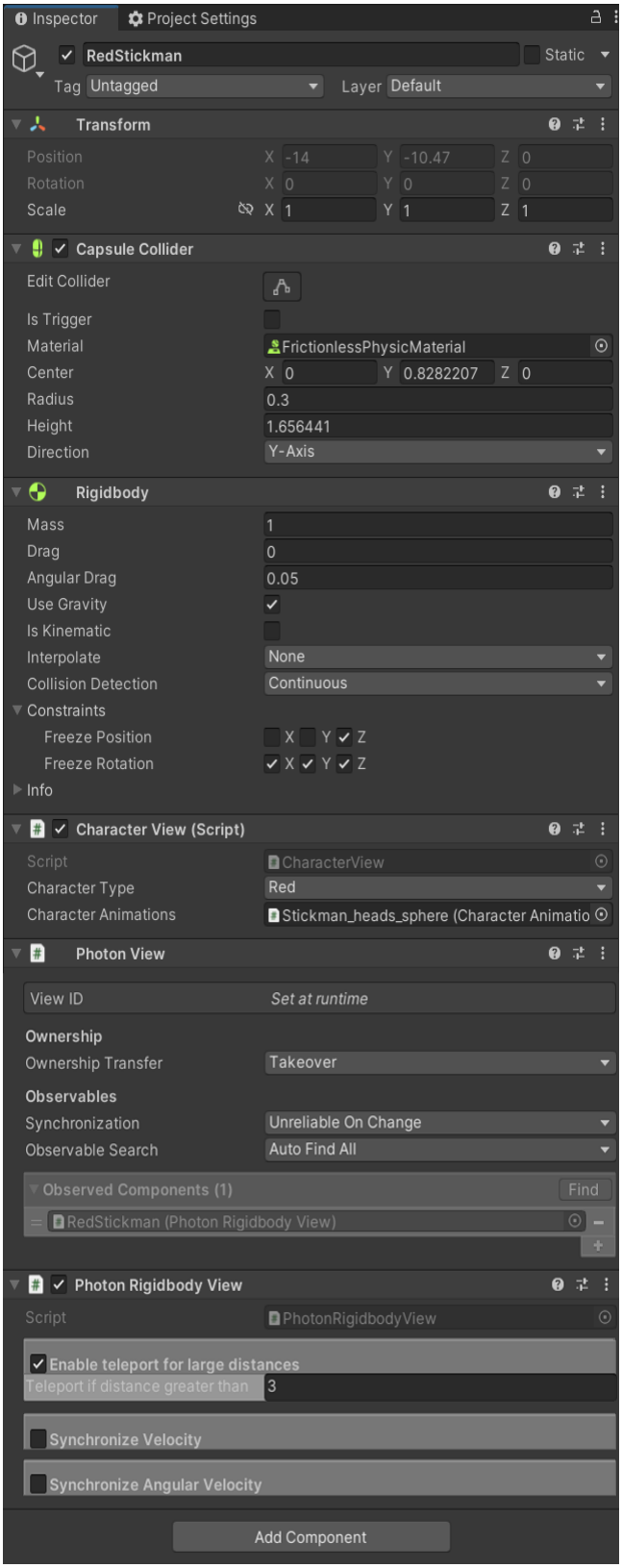

Рисунок 3.1 – MonoBehavours компоненти Unity на ігровому об’єкті головного персонажу

Очікується, що завдяки цим ключовим рішенням реалізація патерну MVP призведе до створення добре структурованої, модульної та

підтримуваної системи. Поділ завдань, визначення контрактів та інтерфейсів, ін’єкція залежностей, програмування на основі подій та інтеграція з компонентною архітектурою Unity в сукупності сприяють ефективному та масштабованому дизайну, що сприяє повторному використанню коду, тестуванню та гнучкості в процесі розробки.

Імплементація Zenject фреймворку

Впровадження фреймворку Zenject в систему передбачає кілька ключових кроків для забезпечення його успішної інтеграції та використання в рамках проекту [19].

Встановлення

Першим кроком є завантаження та встановлення фреймворку Zenject до проекту Unity. Це можна зробити шляхом імпорту відповідного пакунка Zenject або за допомогою менеджера пакунків, наприклад, Unity’s Package Manager (рисунок 3.2) або стороннього менеджера пакунків, наприклад, OpenUPM.

Налаштування

Після встановлення Zenject наступним кроком є налаштування проекту для ефективного використання його можливостей. Це передбачає встановлення необхідних прив’язок і залежностей, щоб уможливити ін’єкцію залежностей. Прив’язки визначають асоціації між інтерфейсами та їх конкретними реалізаціями, що дозволяє Zenject вирішувати та впроваджувати залежності під час виконання.

Рисунок 3.2 – Встановлення фреймворку Zenject

Налаштування контейнера

Після конфігурування проекту необхідно налаштувати контейнер Zenject (рисунок 3.3). Контейнер діє як центральний реєстр для керування створенням об’єктів, ін’єкцією залежностей та видаленням об’єктів. Він відповідає за вирішення залежностей і створення екземплярів об’єктів за запитом.

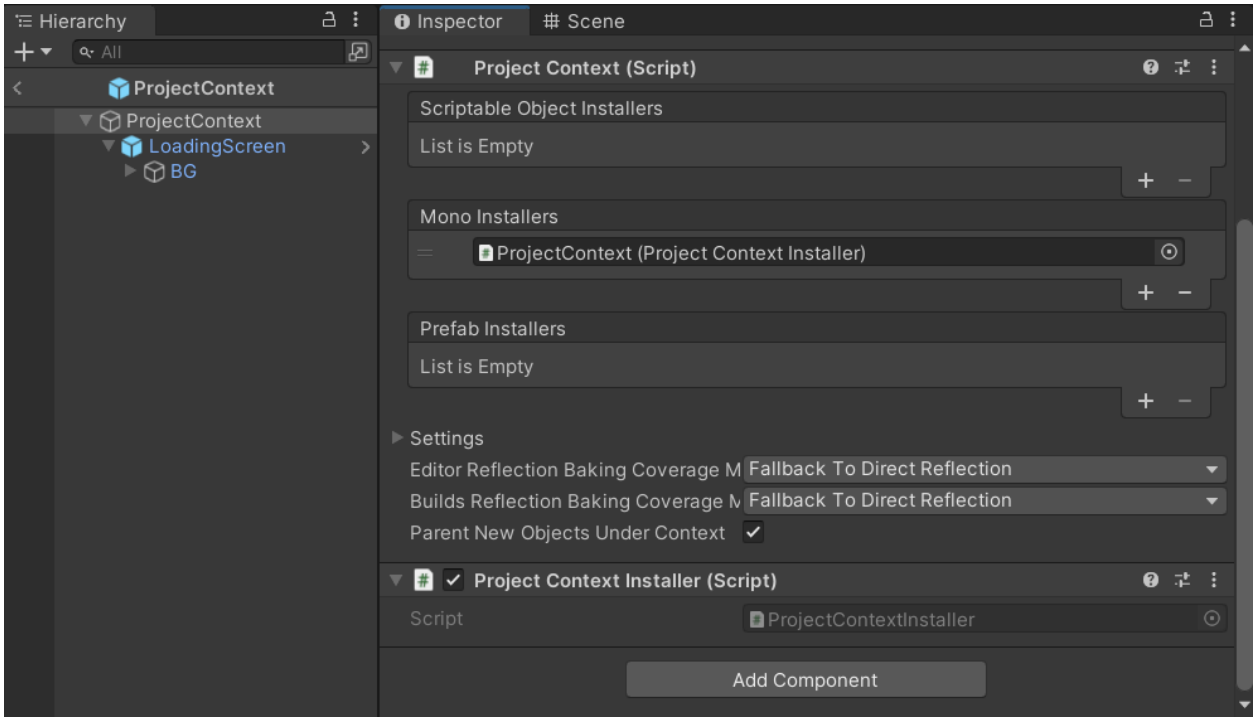

Рисунок 3.3 – Налаштування головного контейнеру Project Context

Склад об'єктів

Після налаштування контейнера наступним кроком є визначення складу об'єктів у системі. Це передбачає створення скриптових об'єктів, моноповедінок або інших компонентів, до яких будуть додані необхідні залежності [5]. Залежності можуть бути додані через ін'єкцію конструктора, ін'єкцію властивості або ін'єкцію методу, залежно від конкретних потреб кожного компонента.

Зв'язування та ін'єкція

Після того, як об'єкти визначено, наступним кроком є встановлення необхідних зв'язків та ін'єкція залежностей у відповідні компоненти. Прив'язки можуть бути встановлені за допомогою зручного API Zenject або за допомогою конфігураційних файлів, таких як інсталятори Zenject або скриптові інсталятори об'єктів. Залежності можна вставляти, вказуючи бажані точки вбудовування в компонентах.

```
public class GameSceneInstaller : MonoInstaller
    {
        public override void InstallBindings()
```

```
            {
                BindControllers();
                BindObjects();
                BindInput();
            }
            private void BindControllers()
            {
        Container.BindInterfacesAndSelfTo<CharactersController>().AsSingle().No
nLazy();
            }
            private void BindObjects()
            {
        Container.BindInterfacesAndSelfTo<CharacterView>().FromComponentsIn
Hierarchy().AsSingle();
            }
            private void BindInput()
            {
        #if !UNITY_EDITOR && (UNITY_ANDROID || UNITY_IOS)
        Container.BindInterfacesAndSelfTo<MobileInputService>().AsSingle().Non
Lazy();
        #else
        Container.BindInterfacesAndSelfTo<StandaloneInputService>().AsSingle().
NonLazy();
        #endif
            }
        }
```

Тестування та валідація

Після впровадження Zenject та ін’єкції залежностей важливо протестувати та перевірити функціональність системи. Це передбачає написання модульних тестів для окремих компонентів, щоб переконатися, що

залежності правильно вирішуються і досягається бажана поведінка. Тестування допомагає виявити будь-які проблеми або помилки в реалізації і дозволяє на ранній стадії виявити і вирішити потенційні проблеми.

Дотримуючись цих ключових кроків, фреймворк Zenject було інтегровано в систему, що дозволяє ефективно впроваджувати залежності і сприяти створенню модульного і підтримуваного коду. Етапи конфігурації, налаштування контейнерів, композиції об’єктів, зв’язування, ін’єкції, тестування та валідації гарантують, що Zenject впроваджено правильно та використано для покращення загального процесу розробки та вдосконалення архітектури системи.

Створення базових механік, об’єктів на сцені та користувальницького інтерфейсу

Щоб створити базову механіку, об’єкти на сцені та користувацький інтерфейс (UI) для ігрового додатку, було виконано кілька ключових кроків.

По-перше, визначено основні механіки гри, такі як рух гравця, взаємодія з об’єктами та вирішення головоломок. Це передбачало розробку та реалізацію необхідних скриптів і компонентів для роботи з цими механізмами, забезпечуючи плавний та інтуїтивно зрозумілий ігровий процес [7].

Наступні поля є важливими компонентами скрипту гравця: characterType (використовується для визначення типу персонажа), characterAnimations (вид, що відповідає за анімацію персонажа), Speed (представляє швидкість руху), JumpForce (визначає силу, що застосовується для стрибка), RotationSpeed (визначає швидкість обертання), MinDistanceToGround (визначає мінімальну відстань від гравця до землі), GravityScale (визначає шкалу гравітації для гравця), _transform (містить компонент трансформації гравця), _rigidbody (посилається на компонент жорсткого тіла гравця), _photonView (посилається на компонент PhotonView для багатокористувацької синхронізації), _characterAnimationType (представляє тип анімації персонажа) та _isGrounded (вказує, чи гравець заземлений).

```
[SerializeField] private CharacterType characterType;
[SerializeField] private CharacterAnimationsView characterAnimations;
private const float Speed = 6;
private const float JumpForce = 11;
private const float RotationSpeed = 30;
private const float MinDistanceToGround = 0.3f;
private const float GravityScale = 1.5f;
private Transform _transform;
private Rigidbody _rigidbody;
private PhotonView _photonView;
private CharacterAnimationType _characterAnimationType;
private bool _isGrounded = true;
public CharacterType CharacterType => characterType;
public PhotonView PhotonView => _photonView;
```

Також скрипт гравця у дипломному проекті містить наступні методи для руху та взаємодії з об’єктам: Move(float directionX), який відповідає за рух гравця та застосування гравітації, Jump() для виконання стрибка, якщо гравець заземлений, та Awake() для ініціалізації необхідних компонентів, таких як transform, rigidbody та PhotonView. Крім того, скрипт містить Update() для оновлення анімації персонажа, IsGrounded() для перевірки, чи гравець заземлений за допомогою променевого кастингу, та UpdateRotation(float directionX) для обробки обертання моделі гравця на основі введеного напрямку.

```
public void Move(float directionX)
{
    Vector3 velocity = _rigidbody.velocity;
    velocity.x = directionX * Speed;
    _rigidbody.velocity = velocity;
    Vector3 gravity = -Physics.gravity.y * GravityScale * Vector3.down;
```

```
            _rigidbody.AddForce(gravity, ForceMode.Acceleration);
            _isGrounded = IsGrounded();
           UpdateRotation(directionX);
        }
        public void Jump()
        {
            if (_isGrounded)
            {
            _rigidbody.velocity = new Vector3(_rigidbody.velocity.x, JumpForce,
0f);
            }
        }
        private void Awake()
        {
            _transform = transform;
            _rigidbody = GetComponent<Rigidbody>();
            _photonView = GetComponent<PhotonView>();
        }
```

Метод IsGrounded() у скрипті гравця перевіряє, чи є персонаж на даний момент заземленим, виконуючи промінь вниз з певної позиції і повертаючи true, якщо промінь потрапляє на нетригерний колайдер, а метод UpdateRotation() оновлює обертання моделі гравця на основі напрямку введення, плавно переходячи до потрібного кута за допомогою LerpAngle і встановлюючи обертання трансформації відповідним чином.

```
        private bool IsGrounded()
        {
            bool isGrounded = false;
            Vector3 raycastOrigin = _transform.position;
            raycastOrigin.y += MinDistanceToGround / 2;
```

```
        if (Physics.Raycast(raycastOrigin, Vector3.down, out RaycastHit raycastHit, MinDistanceToGround))
        {
            isGrounded = !raycastHit.collider.isTrigger;
        }
        return isGrounded;
    }
    private void UpdateRotation(float directionX)
    {
        float desiredAngle = directionX switch
        {
            < 0 => 90,
            > 0 => -90,
            _ => 0
        };
        float targetAngle = Mathf.LerpAngle(_transform.eulerAngles.y, desiredAngle, RotationSpeed * Time.deltaTime);
        _transform.rotation = Quaternion.Euler(0, targetAngle, 0);
        }
    }
```

Далі було створено об’єкти на сцені та правильно їх розташовано. Це включало в себе проектування та моделювання 3D-активів (рисунок 3.4), і розміщення їх у ігровому світі за допомогою редактора сцен Unity (рисунок 3.5). Об’єкти повинні відповідати механіці гри та слугувати своєму призначенню, будь то перешкоди, платформи, колекційні предмети чи елементи головоломки [1].

Рисунок 3.4 – Список 3D-активів проекту

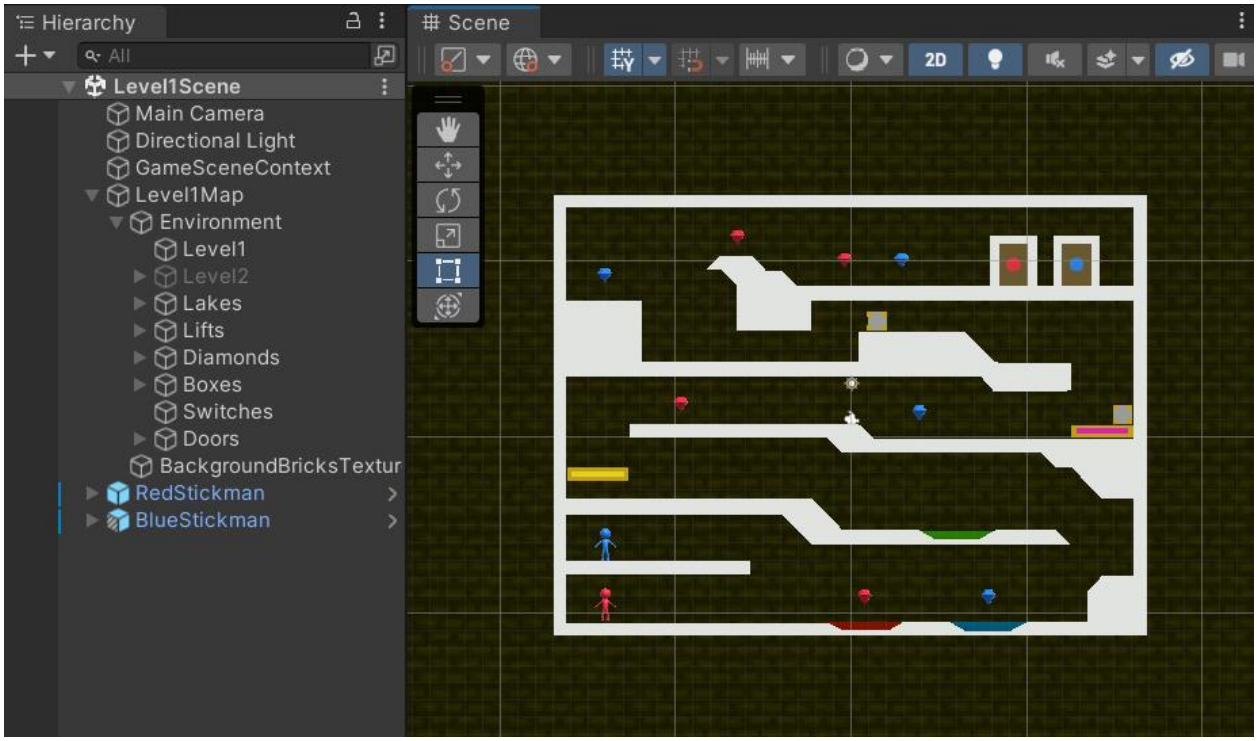

Рисунок 3.5 – Об’єкти на сцені

Елементи користувацького інтерфейсу були розроблені та реалізовані так, щоб забезпечити безперебійну роботу користувача (рисунок 3.6). Це передбало створення полотен інтерфейсу, кнопок, написів та інших інтерактивних елементів, які полегшують введення та зворотній зв’язок з

гравцем [3]. Інтерфейс став інтуїтивно зрозумілим, візуально привабливим і відповідає загальній темі та стилю гри.

Рисунок 3.6 – Екран пошуку та створення кімнат для гри

Після того, як механіки, об’єкти та елементи інтерфейсу були визначені, було написано сценарій і програму, щоб втілити їх у життя. Сюди входить написання коду для обробки вводу гравця, взаємодії об’єктів, фізичного моделювання, анімації та інших специфічних для гри функцій. Сценарії організовані (рисунок 3.7), модульні та добре задокументовані для зручності обслуговування та майбутніх удосконалень.

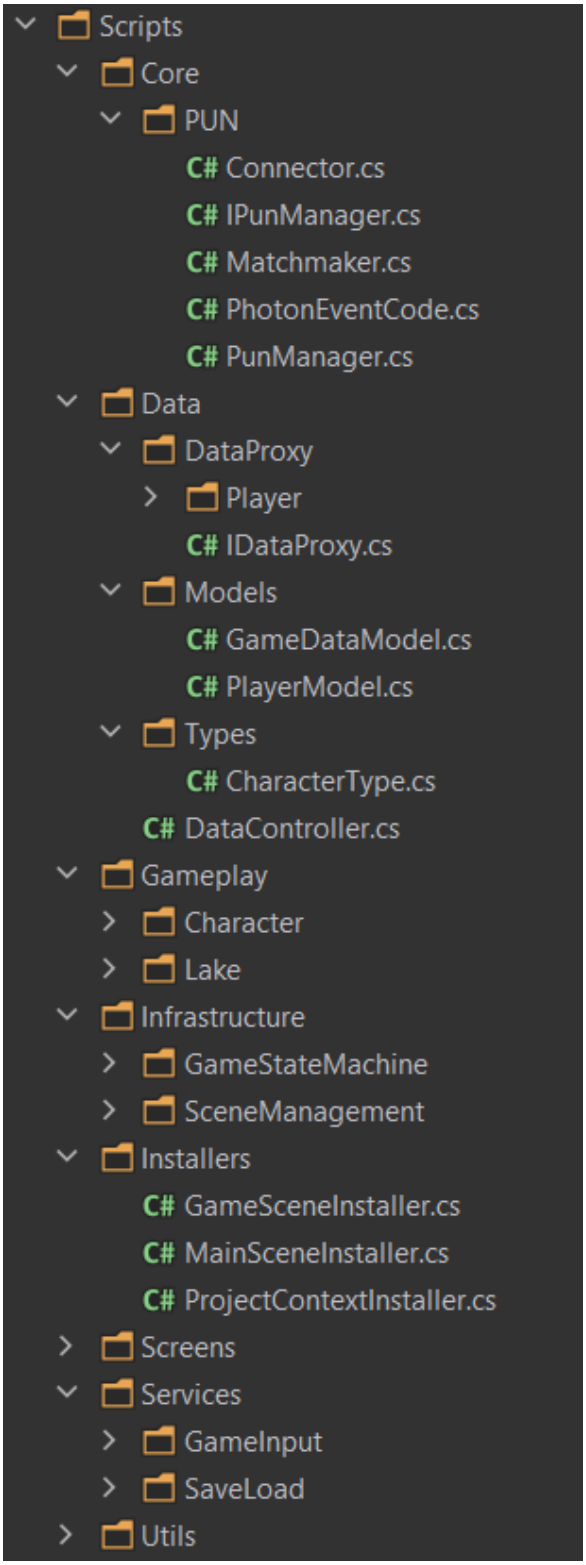

Рисунок 3.7 – Структура скриптів проекту

Інтеграція мережевого фреймворку Photon Unity Networking 2 для багатокористувацьких ігор

Спочатку фреймворк було завантажено за допомогою Unity Package Manager (рисунок 3.8) та імпортовано до проекту Unity.

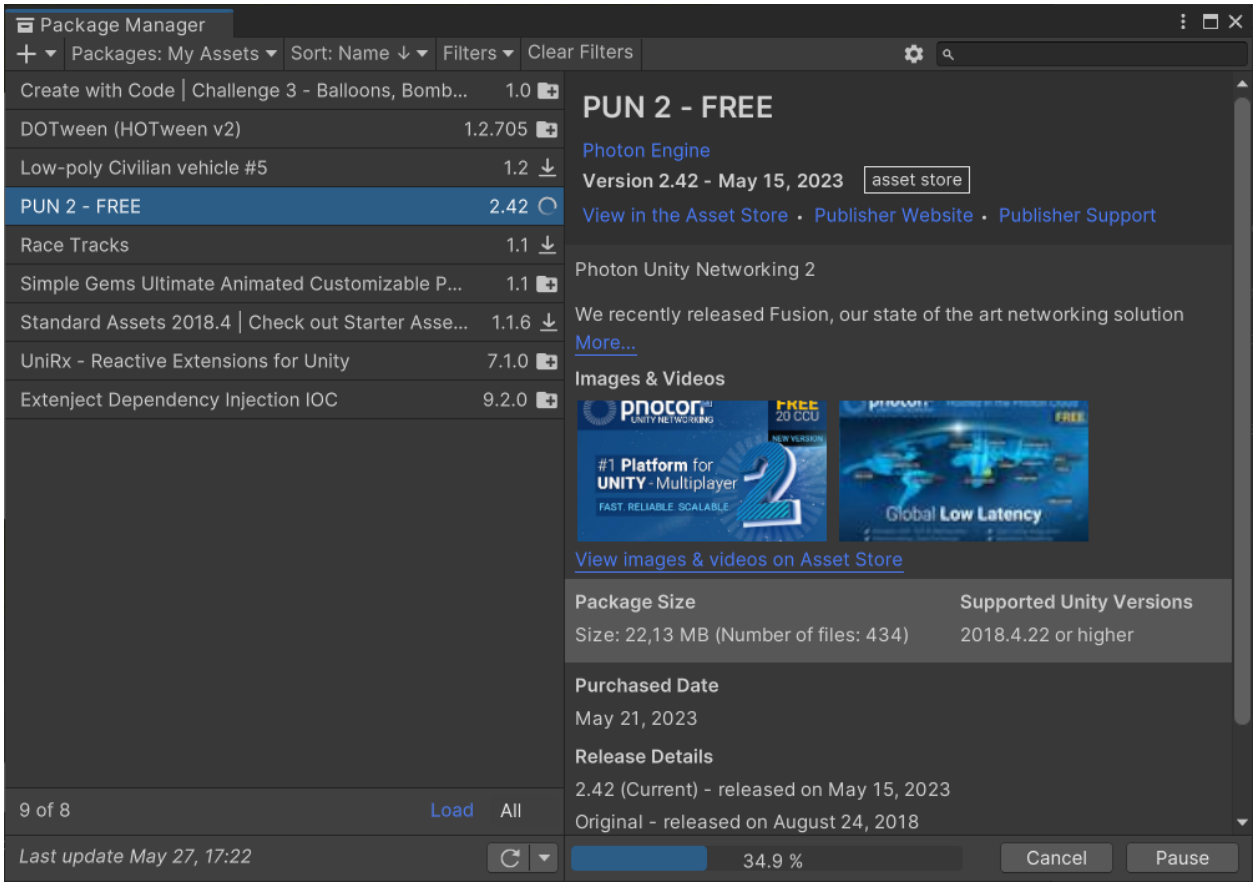

Рисунок 3.8 – Завантаження Photon Unity Networking 2

Далі проект було налаштовано для встановлення з’єднання з серверами Photon. Були визначені відповідні мережеві налаштування, включаючи адресу сервера, номер порту та протокол з’єднання. Ці налаштування дозволили ігровому додатку встановити надійне та безпечне з’єднання з хмарними сервісами Фотон.

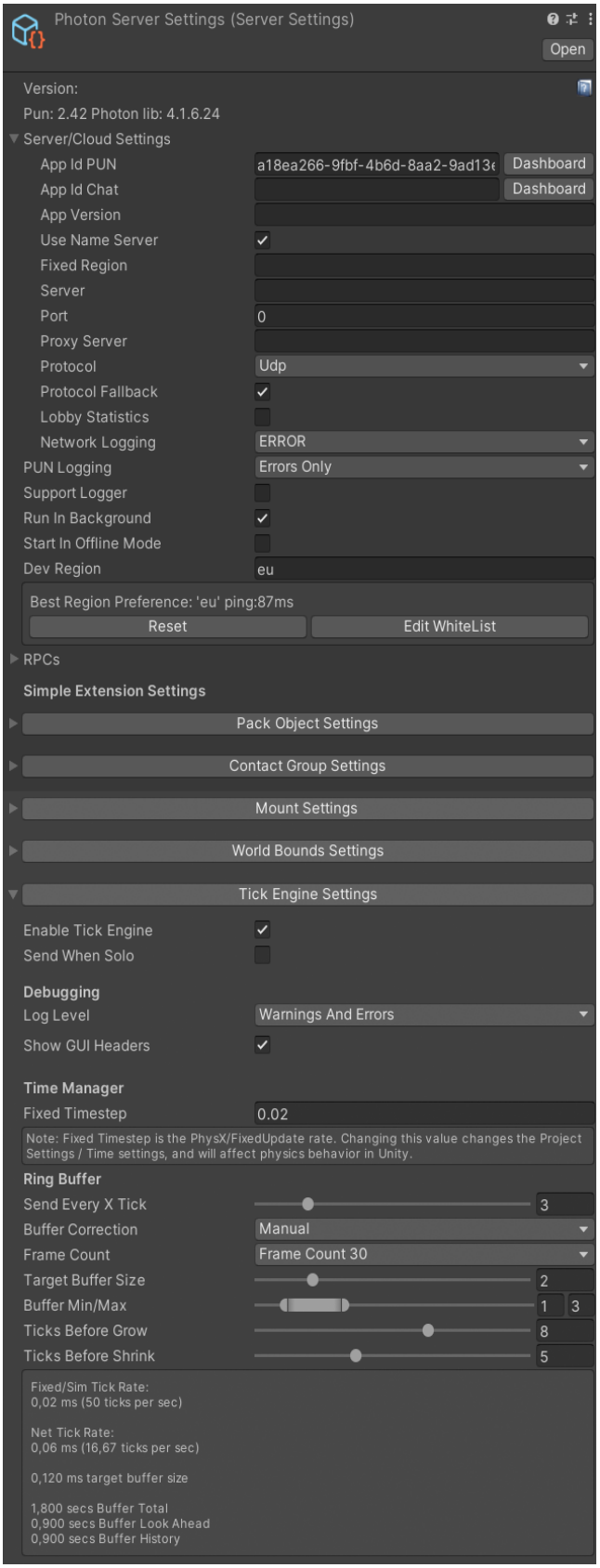

Рисунок 3.9 – Налаштування та встановлення з’єднання в серверами Photon

Після конфігурації була спроектована та реалізована мережева архітектура. Це включало визначення мережевих об’єктів, таких як персонажі

гравців, ігрові предмети та інтерактивні елементи, а також синхронізацію їхніх станів у мережі. Було створено користувацьку мережеву поведінку для обробки об’єктів, передачі прав власності та віддалених викликів процедур (RPC), щоб забезпечити послідовний і синхронізований ігровий процес для всіх підключених гравців.

Крім того, були створені мережеві події та системи обміну повідомленнями для полегшення комунікації між гравцями. Це включало впровадження механізмів обробки подій Photon, що дозволило гравцям надсилати та отримувати повідомлення, сповіщення та оновлення в режимі реального часу. Це забезпечило безперебійну та ефективну багатокористувацьку взаємодію в грі.

Щоб забезпечити безперебійну багатокористувацьку взаємодію, було реалізовано функції знайомства гравців та управління кімнатами з використанням API для знайомства та створення кімнат від Photon. Це дозволило гравцям приєднуватися або створювати ігрові кімнати, шукати доступні матчі на основі бажаних критеріїв та ефективно керувати ігровими сесіями.

Змінні та події класу Mtchmaker з функціями знайомства гравців та управління кімнатами:

```
private const int MaxPlayers = 2;
private readonly IPunManager _punManager;
private readonly Connector _connector;
private readonly ReactiveProperty<int> _playersCount = new();
private readonly CompositeDisposable _compositeDisposable = new();
private List<RoomInfo> _roomInfos;
private string _gameSceneName;
private readonly Subject<bool> _joiningRoom = new();
private readonly Subject<bool> _joinedRoom = new();
private readonly Subject<bool> _startingGame = new();
public IObservable<bool> JoinedRoom => _joinedRoom;
```

```
public IObservable<bool> JoiningRoom => _joiningRoom;
public IObservable<bool> StartingGame => _startingGame;
public IReactiveProperty<int> PlayersCount => _playersCount;
```

Скрипт Matchmaker включає в себе методи з з функціями створення або підключення до кімнат та знайомства гравців. JoinRandomRoom(string gameSceneName) ініціює приєднання випадкової кімнати шляхом підключення до лобі та виклику процедури, JoinRoom() обробляє процес приєднання кімнати, перевіряючи доступні кімнати, сортуючи їх на основі кількості гравців та приєднуючись до існуючої кімнати або створюючи нову кімнату, якщо не знайдено жодної підходящої кімнати.

```
public void JoinRandomRoom(string gameSceneName)
{
    _gameSceneName = gameSceneName;
    _connector.ConnectToLobby(Connector.DefaultLobby,
        delegate { MainThreadDispatcher.StartCoroutine(JoinRoom()); });
}

private IEnumerator JoinRoom()
{
    yield return new WaitWhile(() => _roomInfos == null);
    _playersCount.Value = 1;
    _joiningRoom.OnNext(true);
    PhotonNetwork.AutomaticallySyncScene = true;
    _punManager.JoinedRoom.Take(1).Subscribe(delegate { OnJoinedRoom(); }).AddTo(_compositeDisposable);
    List<RoomInfo> list = _roomInfos.Where(info => info.IsOpen && info.PlayerCount < info.MaxPlayers).ToList();
    if (list.Any())
    {
```

```
                list.Sort((roomInfoA, roomInfoB) => roomInfoA.PlayerCount - roomInfoB.PlayerCount);
                RoomInfo roomInfo = list.Last();
                PhotonNetwork.JoinRoom(roomInfo.Name);
            }
            else
            {
                PhotonNetwork.CreateRoom(Random.Range(1, 101).ToString(), new RoomOptions
                {
                    IsVisible = true,
                    IsOpen = true,
                    MaxPlayers = MaxPlayers,
                    PlayerTtl = 0,
                    CleanupCacheOnLeave = false,
                });
            }
        }
```

Метод OnJoinedRoom() у скрипті Matchmaker обробляє дії, що виконуються при успішному приєднанні до кімнати, такі як оновлення кількості гравців, підписка на події входу та виходу гравців з кімнати, автоматична синхронізація сцени, перевірка максимальної кількості гравців та ініціювання процесу запуску гри, а також завантаження ігрової сцени, якщо поточний клієнт є головним клієнтом.

```
    private void OnJoinedRoom()
    {
        Debug.Log("Joined room");
        _joinedRoom.OnNext(true);
        _playersCount.Value = PhotonNetwork.CurrentRoom.PlayerCount;
```

```
            _punManager.PlayerEnteredRoom.Subscribe(delegate
            {
                _playersCount.Value = PhotonNetwork.CurrentRoom.PlayerCount;
            }).AddTo(_compositeDisposable);
            _punManager.PlayerLeftRoom.Subscribe(delegate
            {
                _playersCount.Value = PhotonNetwork.CurrentRoom.PlayerCount;
            }).AddTo(_compositeDisposable);
            PhotonNetwork.AutomaticallySyncScene = true;
            _playersCount.Where(i => i == PhotonNetwork.CurrentRoom.MaxPlayers).Take(1).Subscribe(delegate
            {
                if (PhotonNetwork.IsMasterClient)
                {
                    PhotonNetwork.CurrentRoom.IsOpen = false;
                    PhotonNetwork.CurrentRoom.IsVisible = false;
                }
                Observable.Timer(TimeSpan.FromSeconds(1)).Subscribe(delegate
                {
                    _startingGame.OnNext(true);
                    if (!PhotonNetwork.IsMasterClient) return;
                    PhotonNetwork.LoadLevel(_gameSceneName);
                });
            }).AddTo(_compositeDisposable);
        }
```

Протягом усього процесу реалізації проводилося ретельне тестування для забезпечення надійності та стабільності мережевого ігрового процесу.

Завдяки успішному впровадженню фреймворку Photon Unity Networking 2, система досягла надійної та масштабованої багатокористувацької функціональності в режимі реального часу.

3.2 Інструкція роботи користувача з системою.

Ця інструкція надасть основні вказівки та інформацію, які допоможуть ефективно орієнтуватися в системі та користуватися нею.

Огляд системи

Ігровий додаток являє собою багатокористувацький 3D-платформер з елементами головоломки, розрахований на двох гравців. Мета гри полягає у співпраці з вашим партнером для подолання викликів, вирішення головоломок і досягнення кінця кожного рівня.

Початок гри

Після запуску гри з’явиться екран головного меню (рисунок 3.10). На ньому можна розпочати нову гру, змінити налаштування або отримати доступ до додаткових функцій.

Рисунок 3.10 – Головне меню

Елементи керування грою

Щоб пересуватися, стрибати, взаємодіяти з об’єктами та виконувати інші дії, необхідні для проходження гри потрібно використовувати клавіші WAD на клавіатурі.

Ігровий процес

Гра складається з декількох рівнів, кожен з яких представляє унікальні завдання та головоломки. Потрібно працювати разом з партнером, щоб подолати перешкоди та вирішити головоломки. Комунікація та співпраця - ключ до успіху. Потрібно звертати увагу на підказки навколишнього середовища, об’єкти та інтерактивні елементи, щоб знайти рішення і перейти на наступний рівень. Можна відстежувати свій прогрес і поточні цілі за допомогою екранних елементів інтерфейсу.

Багатокористувацькі функції

Ігровий додаток використовує багатокористувацьку функцію в режимі реального часу, що дозволяє підключатися та грати з іншим гравцем. Можна приєднатися до системи пошуку партнерів, щоб знайти підходящого партнера або створити власну кімнату для гри.

Збереження та завантаження

Гра автоматично зберігає прогрес після завершення рівня (рисунок 3.11). Можна продовжити гру з того місця, на якому зупинилися.

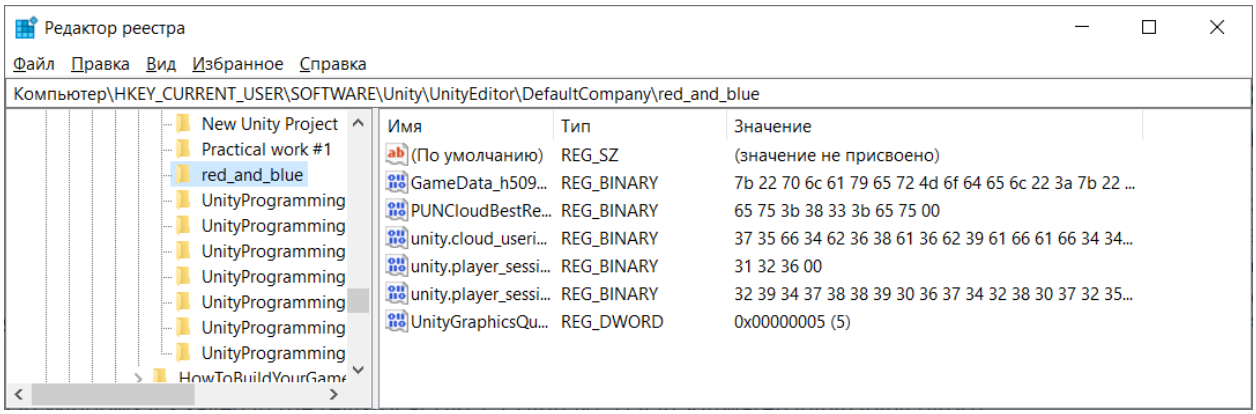

Рисунок 3.11 – Збереженний прогрес гри в реєстрі

3.3 Висновки до третього розділу

Отже, ключові рішення, прийняті під час реалізації системи, такі як прийняття патерну проектування MVP, використання фреймворку Zenject для ефективної ін’єкції залежностей та інтеграція Photon Unity Networking 2 для багатокористувацької функціональності в режимі реального часу, сприяли успішній реалізації цілей роботи.

Інструкція роботи користувача з системою надає користувачам всебічний огляд ігрового додатку та його можливостей. Він пропонує чіткі інструкції щодо початку роботи, розуміння механіки ігрового процесу та навігації в грі. Дотримуючись посібника, користувачі можуть легко долучатися до багатокористувацької співпраці, вирішувати головоломки та проходити рівні без перешкод і з задоволенням.

Разом ці розділи демонструють продумане планування, технічні рішення та орієнтований на користувача підхід, застосований під час розробки ігрового додатку. Ключові рішення та керівництво користувача слугують важливими компонентами у створенні добре продуманої та зручної для користувача системи, яка відповідає цілям дипломної роботи.

ВИСНОВКИ

У ході роботи над дипломною роботою було успішно розроблено ігровий додаток складної архітектури в середовищі Unity з використанням фреймворків. Реалізований програмний продукт являє собою багатокористувацький 3D-платформер з головоломками, що включає в себе сучасні технології та патерни дизайну для забезпечення захоплюючого та спільного ігрового досвіду.

На основі критичного аналізу та досліджень були прийняті ключові рішення щодо реалізації системи. Прийняття шаблону проектування Model-View-Presenter (MVP) забезпечило структурований і модульний підхід до архітектури програмного забезпечення, що полегшило підтримку та розширення коду. Інтеграція таких фреймворків, як Zenject для ін’єкції залежностей та Photon Unity Networking 2 для багатокористувацької функціональності в реальному часі, значно покращила процес розробки та дозволила створити безперебійний мережевий геймплей.

Програмний продукт продемонстрував свою ефективність та практичну значущість в ігровій індустрії. Посібник користувача містить чіткі інструкції та рекомендації для гравців, що дозволяє їм легко орієнтуватися в грі, співпрацювати з іншими та долати труднощі. Ігрова механіка, дизайн рівнів та користувацький інтерфейс були ретельно розроблені для створення захоплюючого та приємного ігрового досвіду.

Перевагами розробленого програмного продукту є надійна архітектура, ефективна багатокористувацька функціональність та зручний інтерфейс. Реалізація патерну проектування MVP забезпечує чіткий розподіл завдань і полегшує майбутні оновлення та вдосконалення. Використання Photon Unity Networking 2 уможливлює безперебійну багатокористувацьку взаємодію та синхронізує ігровий процес за різних мережевих умов.

Однак важливо визнати певні обмеження та потенційні можливості для вдосконалення. На продуктивність ігрової програми можуть впливати

затримки в мережі та кількість одночасних гравців. Подальша оптимізація та міркування щодо масштабування можуть покращити загальний досвід гравців. Крім того, розширення контенту гри, включаючи більше рівнів, головоломок і функцій, забезпечило б розширений ігровий процес і підвищило б цінність повторного проходження.

З точки зору застосування в митних органах та інших установах, програмний продукт є прикладом використання складної архітектури та сучасних фреймворків для розважальних цілей або навчання користувачів. Реалізація багатокористувацької функціональності в режимі реального часу може бути адаптована до різних сценаріїв, що вимагають співпраці та комунікації між користувачами.

Таким чином, в даній дипломній роботі було успішно розроблено ігровий додаток зі складною архітектурою з використанням Unity та фреймворків. Програмний продукт демонструє ефективну реалізацію патерну дизайну MVP, використання Photon Unity Networking 2 для багатокористувацької функціональності та інтеграцію Zenject для ефективної ін’єкції залежностей. Вона забезпечує захоплюючий ігровий досвід, демонструючи потенціал її застосування в митних органах та інших установах.

На основі отриманих результатів рекомендується провести подальше тестування та оптимізацію для підвищення продуктивності та масштабованості гри. Крім того, постійні оновлення та розширення контенту гри можуть залучити ширшу аудиторію та покращити загальний користувацький досвід. Робота слугує фундаментом для майбутнього розвитку та модернізації, досліджуючи нові технічні рішення та інновації в ігровій індустрії.

В цілому, ця дипломна робота досягнула своїх цілей, продемонструвавши успішну розробку складного ігрового додатку та надавши цінну інформацію про реалізацію сучасних фреймворків та патернів проектування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Майк Гейг. Розробка ігор на Unity за 24 години, 2020. 464 с.
2. Джо Хокінг. Unity в дії: Розробка мультиплатформних ігор на C#, 2015. 352 с.
3. Алан Торн. Опанування Unity Scripting, 2015. 380 с.
4. Джеремі Гібсон Бонд. Unity і C#. Геймдев від ідеї до реалізації, 2022. № 3. 944 с.
5. Алекс Окіта. Вивчаємо програмування на C# з Unity 3D, друге видання, 2019. 690 с.
6. Мюллер Д. П. C# для чайників, 2020. 608 с.
7. Марк Прайс. C# 7 .NET Core. Крос-платформна розробка для професіоналів, 2018. № 3. 640 с.
8. Ендрю Троелсен, Філіп Джепикс. Мова програмування C# 7 і платформи .NET і .NET Core, 2020. № 8. 1000 с.
9. Джон Скит. Основні характеристики C # для професіоналів: тонкощі програмування, 2014. № 3. 608 с.
10. Джозеф Албахари, Бен Албахари. C# 7.0. Довідник. Повний опис мови, 2019. 1024 с.
11. Адам Фрімен. ASP.NET Core MVC 2 з прикладами на C# для професіоналів, 2018. № 7. 1000 с.
12. Метью Макдональд. WPF: Windows Presentation Foundation в .NET 4.5 з прикладами на C# 5.0 для професіоналів, 2017. № 4. 1024 с.
13. Джеффри Рихтер. Програмування на платформі Microsoft .NET Framework 4.5 на мові C#, 2020. № 4. 896 с.
14. Петцольд Чарльз. Програмування для Microsoft Windows 8, 2014. № 6. 1008 с.
15. Эндрю Стиллмен. Вивчаємо C#, 2013. № 3. 1312 с.
16. Тепляков С. В. Патерни проектування на платформі .NET, 2018. 320 с.

17. Документи та посібники для роботи з екосистемою Unity. URL: https://docs.unity.com
18. Офіційна документація для Photon Engine. URL: https://doc-api.photonengine.com
19. Офіційна документація для Zenject. URL: https://github.com/modesttree/Zenject
20. Офіційна документація для UniRx. URL: https://github.com/neuecc/UniRx
21. Офіційна документація для DoTween. URL: http://dotween.demigiant.com/documentation.php

ДОДАТОК А

Текст програмного коду.

Connector.cs

```
using System;
using Data.DataProxy.Player;
using Photon.Pun;
using Photon.Realtime;
using UniRx;
using UnityEngine;
namespace Core.PUN
{
    public class Connector : IDisposable
    {
        private readonly IPunManager _punManager;
        private readonly IPlayerDataProxy _playerDataProxy;
        private readonly CompositeDisposable _compositeDisposable = new();
        private Action _onConnectedToLobby;
        private string _lobbyName;
        public const string DefaultLobby = "Game";
        public Connector(IPunManager punManager, IPlayerDataProxy playerDataProxy)
        {
            _punManager = punManager;
            _playerDataProxy = playerDataProxy;
        }
        public void ConnectToLobby(string lobbyName, Action onConnectedToLobby = null)
        {
```

```
            _lobbyName = lobbyName;
            _onConnectedToLobby = onConnectedToLobby;
            if (PhotonNetwork.IsConnected)
            {
                OnConnected();
            }
            else
            {
                _punManager.ConnectedToMaster.Take(1).Subscribe(delegate { OnConnected(); })
                    .AddTo(_compositeDisposable);
                PhotonNetwork.LocalPlayer.NickName = _playerDataProxy.Nickname.Value;
                PhotonNetwork.ConnectUsingSettings();
            }
        }
        private void OnConnected()
        {
            if (PhotonNetwork.InLobby && PhotonNetwork.CurrentLobby.Name == _lobbyName)
            {
                OnJoinedLobby();
            }
            else
            {
                _punManager.JoinedLobby.Take(1).Subscribe(delegate { OnJoinedLobby(); })
                    .AddTo(_compositeDisposable);
                PhotonNetwork.JoinLobby(new TypedLobby(_lobbyName, LobbyType.Default));
```

```
            }
        }
        private void OnJoinedLobby()
        {
            Debug.Log("Connected to lobby");
            _onConnectedToLobby?.Invoke();
        }
        public void Dispose()
        {
            _compositeDisposable.Dispose();
        }
    }
}
```

IPunManager.cs

```
using System;
using System.Collections.Generic;
using ExitGames.Client.Photon;
using Photon.Realtime;
using UniRx;
namespace Core.PUN
{
    public interface IPunManager
    {
        IObservable<bool> ConnectedToMaster { get; }
        IObservable<bool> Connected { get; }
        IObservable<DisconnectCause> Disconnected { get; }
        IObservable<bool> JoinedLobby { get; }
        IObservable<List<RoomInfo>> RoomListUpdate { get; }
```

```
            IObservable<bool> JoinedRoom { get; }
            IObservable<(short, string)> JoinedRandomFailed { get; }
            IObservable<(short, string)> JoinedRoomFailed { get; }
            IObservable<bool> LeftRoom { get; }
            IObservable<Player> PlayerEnteredRoom { get; }
            IObservable<Player> PlayerLeftRoom { get; }
            IObservable<Hashtable> RoomPropertiesUpdate { get; }
            IObservable<(Player, Hashtable)> PlayerPropertiesUpdate { get; }
            IReadOnlyReactiveProperty<string> CurrentRegion { get; }
            IReadOnlyReactiveProperty<IEnumerable<KeyValuePair<string,
int>>> AvailableRegions { get; }
            void SendEvent(PhotonEventCode eventCode, object content,
              ReceiverGroup receiverGroup = ReceiverGroup.All, EventCaching
cachingOption = EventCaching.DoNotCache);
            IDisposable SubscribeToEvent(PhotonEventCode eventCode,
Action<object> action);
        }
    }
```

Matchmaker.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using Photon.Pun;
using Photon.Realtime;
using UniRx;
using UnityEngine;
using Random = UnityEngine.Random;
```

```
namespace Core.PUN
{
    public class Matchmaker : IDisposable
    {
        private const int MaxPlayers = 2;
        private readonly IPunManager _punManager;
        private readonly Connector _connector;
        private readonly ReactiveProperty<int> _playersCount = new();
        private readonly CompositeDisposable _compositeDisposable = new();
        private List<RoomInfo> _roomInfos;
        private string _gameSceneName;
        private readonly Subject<bool> _joiningRoom = new();
        private readonly Subject<bool> _joinedRoom = new();
        private readonly Subject<bool> _startingGame = new();
        public Matchmaker(IPunManager punManager, Connector connector)
        {
            _connector = connector;
            _punManager = punManager;
            _punManager.RoomListUpdate.Subscribe(delegate(List<RoomInfo>
list) { _roomInfos = list; })
                .AddTo(_compositeDisposable);
        }
        public IObservable<bool> JoinedRoom => _joinedRoom;
        public IObservable<bool> JoiningRoom => _joiningRoom;
        public IObservable<bool> StartingGame => _startingGame;
        public IReactiveProperty<int> PlayersCount => _playersCount;
        public void JoinRandomRoom(string gameSceneName)
        {
            _gameSceneName = gameSceneName;
            _connector.ConnectToLobby(Connector.DefaultLobby,
```

```
            delegate { MainThreadDispatcher.StartCoroutine(JoinRoom()); });
        }
        public void LeaveRoom()
        {
            if (!PhotonNetwork.InRoom)
            {
                throw new InvalidOperationException();
            }
            PhotonNetwork.LeaveRoom();
            _compositeDisposable.Clear();
        }
        private IEnumerator JoinRoom()
        {
            yield return new WaitWhile(() => _roomInfos == null);
            _playersCount.Value = 1;
            _joiningRoom.OnNext(true);
            PhotonNetwork.AutomaticallySyncScene = true;
            _punManager.JoinedRoom.Take(1).Subscribe(delegate { OnJoinedRoom(); }).AddTo(_compositeDisposable);
            List<RoomInfo> list = _roomInfos.Where(info => info.IsOpen && info.PlayerCount < info.MaxPlayers).ToList();
            if (list.Any())
            {
                list.Sort((roomInfoA, roomInfoB) => roomInfoA.PlayerCount - roomInfoB.PlayerCount);
                RoomInfo roomInfo = list.Last();
                PhotonNetwork.JoinRoom(roomInfo.Name);
            }
            else
            {
```

```
                PhotonNetwork.CreateRoom(Random.Range(1,    101).ToString(),
new RoomOptions
                {
                    IsVisible = true,
                    IsOpen = true,
                    MaxPlayers = MaxPlayers,
                    PlayerTtl = 0,
                    CleanupCacheOnLeave = false,
                });
            }
        }
        private void OnJoinedRoom()
        {
            Debug.Log("Joined room");
            _joinedRoom.OnNext(true);
            _playersCount.Value = PhotonNetwork.CurrentRoom.PlayerCount;
            _punManager.PlayerEnteredRoom.Subscribe(delegate
            {
                _playersCount.Value                                    =
PhotonNetwork.CurrentRoom.PlayerCount;
            }).AddTo(_compositeDisposable);
            _punManager.PlayerLeftRoom.Subscribe(delegate
            {
                _playersCount.Value                                    =
PhotonNetwork.CurrentRoom.PlayerCount;
            }).AddTo(_compositeDisposable);
            PhotonNetwork.AutomaticallySyncScene = true;
            _playersCount.Where(i                =>                i                ==
PhotonNetwork.CurrentRoom.MaxPlayers).Take(1).Subscribe(delegate
            {
```

```
            if (PhotonNetwork.IsMasterClient)
            {
                PhotonNetwork.CurrentRoom.IsOpen = false;
                PhotonNetwork.CurrentRoom.IsVisible = false;
            }
            Observable.Timer(TimeSpan.FromSeconds(1)).Subscribe(delegate
            {
                _startingGame.OnNext(true);
                if (!PhotonNetwork.IsMasterClient) return;
                PhotonNetwork.LoadLevel(_gameSceneName);
            });
        }).AddTo(_compositeDisposable);
    }
    public void Dispose()
    {
        _compositeDisposable.Dispose();
    }
  }
}
```

PhotonEventCode.cs

```
namespace Core.PUN
{
  public enum PhotonEventCode
  {
  }
}
```

PunManager.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using ExitGames.Client.Photon;
using Photon.Pun;
using Photon.Realtime;
using UniRx;
using UnityEngine;
using Hashtable = ExitGames.Client.Photon.Hashtable;
namespace Core.PUN
{
    public class PunManager : MonoBehaviourPunCallbacks, IPunManager
    {
        private readonly Subject<bool> _connectedToMaster = new();
        private readonly Subject<bool> _connected = new();
        private readonly Subject<DisconnectCause> _disconnected = new();
        private readonly Subject<bool> _joinedLobby = new();
        private readonly Subject<List<RoomInfo>> _roomListUpdate = new();
        private readonly Subject<bool> _joinedRoom = new();
        private readonly Subject<(short, string)> _joinedRandomFailed = new();
        private readonly Subject<(short, string)> _joinedRoomFailed = new();
        private readonly Subject<bool> _leftRoom = new();
        private readonly Subject<Player> _playerEnteredRoom = new();
        private readonly Subject<Player> _playerLeftRoom = new();
        private readonly Subject<Hashtable> _roomPropertiesUpdate = new();
        private readonly Subject<(Player, Hashtable)> _playerPropertiesUpdate
= new();
        private readonly Subject<EventData> _eventReceived = new();
```

```
        private readonly ReactiveProperty<IEnumerable<KeyValuePair<string,
int>>> _availableRegions =
            new(new Dictionary<string, int>());
        private readonly ReactiveProperty<string> _currentRegion = new("");
        public IObservable<bool> ConnectedToMaster =>
_connectedToMaster;
        public IObservable<bool> Connected => _connected;
        public IObservable<DisconnectCause> Disconnected => _disconnected;
        public IObservable<bool> JoinedLobby => _joinedLobby;
        public IObservable<List<RoomInfo>> RoomListUpdate =>
_roomListUpdate;
        public IObservable<bool> JoinedRoom => _joinedRoom;
        public IObservable<(short, string)> JoinedRandomFailed =>
_joinedRandomFailed;
        public IObservable<(short, string)> JoinedRoomFailed =>
_joinedRoomFailed;
        public IObservable<bool> LeftRoom => _leftRoom;
        public IObservable<Player> PlayerEnteredRoom =>
_playerEnteredRoom;
        public IObservable<Player> PlayerLeftRoom => _playerLeftRoom;
        public IObservable<Hashtable> RoomPropertiesUpdate =>
_roomPropertiesUpdate;
        public IObservable<(Player, Hashtable)> PlayerPropertiesUpdate =>
_playerPropertiesUpdate;
        public IReadOnlyReactiveProperty<string> CurrentRegion =>
_currentRegion;
        public
IReadOnlyReactiveProperty<IEnumerable<KeyValuePair<string, int>>>
AvailableRegions => _availableRegions;
```

```
        public override void OnConnectedToMaster()
        {
            _currentRegion.Value = PhotonNetwork.CloudRegion.Replace("/*",
"");
            _connectedToMaster.OnNext(true);
        }
        public override void OnRegionListReceived(RegionHandler
regionHandler)
        {
            base.OnRegionListReceived(regionHandler);
            _availableRegions.Value = GetAvailableRegions(regionHandler);
        }
        public override void OnDisconnected(DisconnectCause cause)
        {
            _disconnected.OnNext(cause);
            StartCoroutine(Reconnect());
        }
        public override void OnJoinedLobby()
        {
            _joinedLobby.OnNext(true);
        }
        public override void OnRoomListUpdate(List<RoomInfo> roomList)
        {
            _roomListUpdate.OnNext(roomList);
        }
        public override void OnJoinedRoom()
        {
            _joinedRoom.OnNext(true);
        }
```

```
        public override void OnJoinRandomFailed(short returnCode, string
message)
        {
            _joinedRandomFailed.OnNext((returnCode, message));
        }
        public override void OnJoinRoomFailed(short returnCode, string
message)
        {
            _joinedRoomFailed.OnNext((returnCode, message));
        }
        public override void OnLeftRoom()
        {
            _leftRoom.OnNext(true);
        }
        public override void OnPlayerEnteredRoom(Player newPlayer)
        {
            _playerEnteredRoom.OnNext(newPlayer);
        }
        public override void OnPlayerLeftRoom(Player otherPlayer)
        {
            _playerLeftRoom.OnNext(otherPlayer);
        }
        public override void OnRoomPropertiesUpdate(Hashtable
propertiesThatChanged)
        {
            _roomPropertiesUpdate.OnNext(propertiesThatChanged);
        }
        public override void OnPlayerPropertiesUpdate(Player targetPlayer,
Hashtable changedProps)
        {
```

```
            _playerPropertiesUpdate.OnNext((targetPlayer, changedProps));
        }
        public void SendEvent(PhotonEventCode eventCode, object content,
            ReceiverGroup receiverGroup = ReceiverGroup.All, EventCaching
cachingOption = EventCaching.DoNotCache)
        {
            RaiseEventOptions raiseEventOptions = new()
            {
                CachingOption = cachingOption,
                Receivers = receiverGroup,
            };
            PhotonNetwork.RaiseEvent((byte) eventCode, content,
raiseEventOptions, SendOptions.SendReliable);
            PhotonNetwork.SendAllOutgoingCommands();
        }
        public IDisposable SubscribeToEvent(PhotonEventCode eventCode,
Action<object> action)
        {
            return _eventReceived.Where(data => data.Code == (int)
eventCode).Select(data => data.CustomData)
                .Subscribe(action);
        }
        private void OnEventReceived(EventData obj)
        {
            _eventReceived.OnNext(obj);
        }
        private Dictionary<string, int> GetAvailableRegions(RegionHandler
regionHandler)
        {
            Dictionary<string, int> regionsDictionary = new();
```

```
            foreach (Region regHandlerEnabledRegion in
regionHandler.EnabledRegions)
            {
                regionsDictionary[regHandlerEnabledRegion.Code] =
regHandlerEnabledRegion.Ping;
            }
            return regionsDictionary;
        }
        public override void OnConnected()
        {
            base.OnConnected();
            _connected.OnNext(true);
        }
        private static IEnumerator Reconnect()
        {
            while (PhotonNetwork.NetworkClientState ==
ClientState.Disconnected)
            {
                if (!PhotonNetwork.ReconnectAndRejoin())
                {
                    PhotonNetwork.Reconnect();
                }
                yield return new WaitForSeconds(0.1f);
            }
        }
        public override void OnEnable()
        {
            base.OnEnable();
```

```
            PhotonNetwork.NetworkingClient.EventReceived += OnEventReceived;
        }
        public override void OnDisable()
        {
            base.OnDisable();
            PhotonNetwork.NetworkingClient.EventReceived -= OnEventReceived;
        }
    }
}
```

IPlayerDataProxy.cs

```
using UniRx;
namespace Data.DataProxy.Player
{
    public interface IPlayerDataProxy
    {
        IReadOnlyReactiveProperty<string> Nickname { get; }
    }
}
```

PlayerDataProxy.cs

```
using System;
using Data.Models;
using UniRx;
namespace Data.DataProxy.Player
{
```

```
    public class PlayerDataProxy : IDataProxy, IPlayerDataProxy
    {
        private GameDataModel _gameDataModel;
        private readonly ReactiveProperty<string> _nickname = new();
        public event Action GameDataChanged;
        public IReadOnlyReactiveProperty<string> Nickname => _nickname;
        public void UpdateData(GameDataModel gameDataModel)
        {
            _gameDataModel = gameDataModel;
            _nickname.Value = _gameDataModel.playerModel.nickname;
        }
        public void ClearData()
        {
        }
    }
}
```

IDataProxy.cs

```
using System;
using Data.Models;
namespace Data.DataProxy
{
    public interface IDataProxy
    {
        event Action GameDataChanged;
        void UpdateData(GameDataModel gameDataModel);
        void ClearData();
    }
}
```

GameDataModel.cs

```
using System;
namespace Data.Models
{
   [Serializable]
   public class GameDataModel
   {
      public PlayerModel playerModel;
      public static GameDataModel CreateNewGameDataModel()
      {
         return new GameDataModel
         {
            playerModel = PlayerModel.CreateNewPlayerModel()
         };
      }
   }
}
```

PlayerModel.cs

```
using System;
using Random = UnityEngine.Random;
namespace Data.Models
{
   [Serializable]
   public class PlayerModel
   {
      public string nickname;
```

```
        public static PlayerModel CreateNewPlayerModel()
        {
            return new PlayerModel
            {
                nickname = "Player" + Random.Range(1000, 9999)
            };
        }
    }
}
```

CharacterType.cs

```
namespace Data.Types
{
    public enum CharacterType
    {
        Red,
        Blue
    }
}
```

DataController.cs

```
using System.Collections.Generic;
using Data.DataProxy;
using Data.Models;
using Services.SaveLoad;
using UniRx;
namespace Data
```

```
    {
        public class DataController
        {
            private readonly ISaveLoadService _saveLoadService;
            private readonly List<IDataProxy> _dataProxies;
            private GameDataModel _currentGameData;
            public DataController(ISaveLoadService saveLoadService,
List<IDataProxy> dataProxies)
            {
                _saveLoadService = saveLoadService;
                _dataProxies = dataProxies;
                foreach (IDataProxy dataProxy in _dataProxies)
                {
                    dataProxy.GameDataChanged += SaveData;
                }
                Observable.EveryApplicationFocus().Where(hasFocus =>
!hasFocus).Subscribe(delegate
                {
                    if (_currentGameData == null) return;
                    SaveData();
                });
            }

            public void SetGameData(GameDataModel gameDataModel)
            {
                ClearData();
                _currentGameData = gameDataModel;
                UpdateData();
            }
            private void SaveData()
```

```
        {
            _saveLoadService.SaveGameData(_currentGameData);
        }
        private void ClearData()
        {
            foreach (IDataProxy dataProxy in _dataProxies)
            {
                dataProxy.ClearData();
            }
        }
        private void UpdateData()
        {
            foreach (IDataProxy dataProxy in _dataProxies)
            {
                dataProxy.UpdateData(_currentGameData);
            }
        }
    }
}
```

Character.cs

```
using Photon.Pun;
using Services.GameInput;
using UniRx;
namespace Gameplay.Character
{
    public class Character
    {
        private readonly CharacterView _view;
```

```
        private readonly IInputService _inputService;
        private readonly bool _haveStateAuthority;
        public Character(CharacterView view, IInputService inputService, bool
haveStateAuthority)
        {
            _view = view;
            _inputService = inputService;
            _haveStateAuthority = haveStateAuthority;
            Initialize();
        }
        private void Initialize()
        {
            if (!_haveStateAuthority) return;
            RequestOwnership();
            Observable.EveryFixedUpdate().Subscribe(delegate                    {
_view.Move(_inputService.Player1MovementDirectionX); })
                .AddTo(_view);
            _inputService.Player1JumpAction.Subscribe(delegate                  {
_view.Jump(); }).AddTo(_view);
        }
        private void RequestOwnership()
        {
            if (_view.PhotonView.IsMine) return;

_view.PhotonView.TransferOwnership(PhotonNetwork.LocalPlayer);
        }
    }
}
```

CharecterAnimationsView.cs

```
using UnityEngine;
namespace Gameplay.Character
{
    public class CharacterAnimationsView : MonoBehaviour
    {
        private Animator _animator;
        public void SetAnimationState(CharacterAnimationType characterAnimationType)
        {
            foreach (CharacterAnimationType animationType in Utils.Utils.GetEnumValues<CharacterAnimationType>())
            {
                _animator.SetBool(animationType.ToString(), animationType == characterAnimationType);
            }
        }
        private void Awake()
        {
            _animator = GetComponent<Animator>();
        }
    }
    public enum CharacterAnimationType
    {
        IsIdle,
        IsRun,
        IsPush,
        IsFloat
    }
}
```

CharactersController.cs

```
using System.Collections.Generic;
using Data.Types;
using Photon.Pun;
using Services.GameInput;
using Zenject;
namespace Gameplay.Character
{
    public class CharactersController : IInitializable
    {
        private readonly List<CharacterView> _characterViews;
        private readonly IInputService _inputService;
        private readonly List<Character> _characters = new();

        public CharactersController(List<CharacterView> characterViews,
IInputService inputService)
        {
            _characterViews = characterViews;
            _inputService = inputService;
        }
        public void Initialize()
        {
            foreach (CharacterView characterView in _characterViews)
            {
                bool haveStateAuthority =
                    (PhotonNetwork.IsMasterClient &&
characterView.CharacterType == CharacterType.Red) ||
```

```
                    (!PhotonNetwork.IsMasterClient &&
characterView.CharacterType == CharacterType.Blue);
                    _characters.Add(new Character(characterView, _inputService,
haveStateAuthority));
                }
            }
        }
    }
```

CharacterView.cs

```
using Data.Types;
using Photon.Pun;
using UnityEngine;
namespace Gameplay.Character
{
    [RequireComponent(typeof(Rigidbody))]
    public class CharacterView : MonoBehaviour
    {
        [SerializeField] private CharacterType characterType;
        [SerializeField] private CharacterAnimationsView
characterAnimations;
        private const float Speed = 6;
        private const float JumpForce = 11;
        private const float RotationSpeed = 30;
        private const float MinDistanceToGround = 0.3f;
        private const float GravityScale = 1.5f;
        private Transform _transform;
        private Rigidbody _rigidbody;
        private PhotonView _photonView;
```

```
        private CharacterAnimationType _characterAnimationType;
        private bool _isGrounded = true;
        public CharacterType CharacterType => characterType;
        public PhotonView PhotonView => _photonView;
        public void Move(float directionX)
        {
            Vector3 velocity = _rigidbody.velocity;
            velocity.x = directionX * Speed;
            _rigidbody.velocity = velocity;
            Vector3 gravity = -Physics.gravity.y * GravityScale * Vector3.down;
            _rigidbody.AddForce(gravity, ForceMode.Acceleration);
            _isGrounded = IsGrounded();
            UpdateRotation(directionX);
            UpdateAnimation(directionX);
        }
        public void Jump()
        {
            if (_isGrounded)
            {
                _rigidbody.velocity = new Vector3(_rigidbody.velocity.x,
JumpForce, 0f);
            }
        }
        private void Awake()
        {
            _transform = transform;
            _rigidbody = GetComponent<Rigidbody>();
            _photonView = GetComponent<PhotonView>();
        }
        private void Update()
```

```
        {
            characterAnimations.SetAnimationState(_characterAnimationType);
        }
        private bool IsGrounded()
        {
            bool isGrounded = false;
            Vector3 raycastOrigin = _transform.position;
            raycastOrigin.y += MinDistanceToGround / 2;
            if (Physics.Raycast(raycastOrigin, Vector3.down, out RaycastHit raycastHit, MinDistanceToGround))
            {
                isGrounded = !raycastHit.collider.isTrigger;
            }
            return isGrounded;
        }
        private void UpdateRotation(float directionX)
        {
            float desiredAngle = directionX switch
            {
                < 0 => 90,
                > 0 => -90,
                _ => 0
            };
            float targetAngle = Mathf.LerpAngle(_transform.eulerAngles.y, desiredAngle, RotationSpeed * Time.deltaTime);
            _transform.rotation = Quaternion.Euler(0, targetAngle, 0);
        }
        private void UpdateAnimation(float directionX)
        {
```

```
                CharacterAnimationType animationType =
CharacterAnimationType.IsIdle;
                if (!_isGrounded)
                {
                    animationType = CharacterAnimationType.IsFloat;
                }
                else
                {
                    if (directionX != 0)
                    {
                        animationType = CharacterAnimationType.IsRun;
                    }
                }
                _characterAnimationType = animationType;
            }
        }
    }
```

LakeType.cs

```
namespace Gameplay.Lake
{
    public enum LakeType
    {
        Water,
        Lava,
        GreenMud
    }
}
```

LakeView.cs

```
using UnityEngine;
namespace Gameplay.Lake
{
    public class LakeView : MonoBehaviour
    {
        [SerializeField] private LakeType lakeType;
        public LakeType LakeType => lakeType;
    }
}
```

GameStateMachine.cs

```
using Data;
using Infrastructure.GameStateMachine.GameStates;
using Infrastructure.SceneManagement;
using Screens;
using Screens.LoadingsScreen;
using Services.SaveLoad;
using Utils.StateMachine;
using Zenject;
namespace Infrastructure.GameStateMachine
{
    public class GameStateMachine : StateMachine, IInitializable
    {
        public GameStateMachine(SceneLoader sceneLoader,
ISaveLoadService saveLoadService,
            DataController dataController, LoadingScreen loadingScreen,
ScreenNavigationSystem screenNavigationSystem)
```

```
            {
                AddNewState(new BootstrapState(this, sceneLoader));
                AddNewState(new LoadProgressState(this, saveLoadService,
dataController));
                AddNewState(new LoadLevelState(this, sceneLoader,
screenNavigationSystem, loadingScreen));
                AddNewState(new GameLoopState(this));
            }
            public void Initialize()
            {
                Enter<BootstrapState>();
            }
        }
    }
```

BootstrapState.cs

```
using Infrastructure.SceneManagement;
using Utils.StateMachine;
namespace Infrastructure.GameStateMachine.GameStates
{
    public class BootstrapState : IState
    {
        private readonly GameStateMachine _gameStateMachine;
        private readonly SceneLoader _sceneLoader;
        public BootstrapState(GameStateMachine gameStateMachine,
SceneLoader sceneLoader)
        {
            _gameStateMachine = gameStateMachine;
            _sceneLoader = sceneLoader;
```

```
            }
            public void Enter()
            {
                _sceneLoader.Load(SceneName.InitialScene,
OnLoadedInitialScene);
            }
            private void OnLoadedInitialScene()
            {
                _gameStateMachine.Enter<LoadProgressState>();
            }
            public void Exit()
            {
            }
        }
    }
```

GameLoopState.cs

```
using Utils.StateMachine;
namespace Infrastructure.GameStateMachine.GameStates
{
    public class GameLoopState : IState
    {
        public GameLoopState(GameStateMachine gameStateMachine)
        {
        }
        public void Enter()
        {
        }
        public void Exit()
```

```
            {
            }
        }
    }
```

LoadLevelState.cs

```
using System;
using Infrastructure.SceneManagement;
using Screens;
using Screens.LoadingsScreen;
using UniRx;
using Utils.StateMachine;
namespace Infrastructure.GameStateMachine.GameStates
{
    public class LoadLevelState : IPayloadedState<SceneName>
    {
        private const float SceneLoadDelay = 1;
        private readonly GameStateMachine _gameStateMachine;
        private readonly SceneLoader _sceneLoader;
        private readonly ScreenNavigationSystem _screenNavigationSystem;
        private readonly LoadingScreen _loadingScreen;
        public LoadLevelState(GameStateMachine gameStateMachine,
SceneLoader sceneLoader,
            ScreenNavigationSystem screenNavigationSystem, LoadingScreen
loadingScreen)
        {
            _gameStateMachine = gameStateMachine;
            _sceneLoader = sceneLoader;
            _screenNavigationSystem = screenNavigationSystem;
```

```
            _loadingScreen = loadingScreen;
        }
        public void Enter(SceneName sceneName)
        {
            _screenNavigationSystem.ClearAvailableScreens();
            _loadingScreen.Show();

Observable.Timer(TimeSpan.FromSeconds(SceneLoadDelay)).Subscribe(delegate
            {
                _sceneLoader.Load(sceneName, OnLoadedScene);
            });
        }
        private void OnLoadedScene()
        {
            _loadingScreen.Hide();
            _gameStateMachine.Enter<GameLoopState>();
        }
        public void Exit()
        {
        }
    }
}
```

LoadProgressState.cs

```
using Data;
using Data.Models;
using Infrastructure.SceneManagement;
using Services.SaveLoad;
using Utils.StateMachine;
```

```
namespace Infrastructure.GameStateMachine.GameStates
{
    public class LoadProgressState : IState
    {
        private readonly GameStateMachine _gameStateMachine;
        private readonly ISaveLoadService _saveLoadService;
        private readonly DataController _dataController;
        public LoadProgressState(GameStateMachine gameStateMachine,
ISaveLoadService saveLoadService,
            DataController dataController)
        {
            _gameStateMachine = gameStateMachine;
            _saveLoadService = saveLoadService;
            _dataController = dataController;
        }
        public void Enter()
        {
            LoadGameData();
            _gameStateMachine.Enter<LoadLevelState,
SceneName>(SceneName.MainScene);
        }
        public void Exit()
        {
        }
        private void LoadGameData()
        {
            GameDataModel gameDataModel =
_saveLoadService.LoadGameData();
            ValidateGameData(gameDataModel);
```

```
            _dataController.SetGameData(gameDataModel);
        }
        private static void ValidateGameData(GameDataModel gameDataModel)
        {
            GameDataModel emptyGameDataModel = GameDataModel.CreateNewGameDataModel();
            gameDataModel.playerModel ??= emptyGameDataModel.playerModel;
        }
    }
}
```

SceneLoader.cs

```
using System;
using System.Collections;
using UniRx;
using UnityEngine;
using UnityEngine.SceneManagement;
namespace Infrastructure.SceneManagement
{
    public class SceneLoader
    {
        public void Load(SceneName sceneName, Action onLoaded = null)
        {
            MainThreadDispatcher.StartCoroutine(LoadScene(sceneName, onLoaded));
        }
```

```
        private IEnumerator LoadScene(SceneName sceneName, Action onLoaded = null)
        {
            if (SceneManager.GetActiveScene().name == sceneName.ToString())
            {
                onLoaded?.Invoke();
                yield break;
            }
            AsyncOperation waitNextScene = SceneManager.LoadSceneAsync(sceneName.ToString());
            while (!waitNextScene.isDone)
                yield return null;
            onLoaded?.Invoke();
        }
    }
}
```

SceneName.cs

```
namespace Infrastructure.SceneManagement
{
    public enum SceneName
    {
        InitialScene,
        MainScene,
        Level1Scene,
    }
}
```

GameSceneInstaller.cs

```
using Gameplay.Character;
using Services.GameInput;
using Zenject;
namespace Installers
{
    public class GameSceneInstaller : MonoInstaller
    {
        public override void InstallBindings()
        {
            BindControllers();
            BindObjects();
            BindInput();
        }
        private void BindControllers()
        {

Container.BindInterfacesAndSelfTo<CharactersController>().AsSingle().NonLazy
();
        }
        private void BindObjects()
        {

Container.BindInterfacesAndSelfTo<CharacterView>().FromComponentsInHierar
chy().AsSingle();
        }
        private void BindInput()
        {
    #if !UNITY_EDITOR && (UNITY_ANDROID || UNITY_IOS)
```

```
Container.BindInterfacesAndSelfTo<MobileInputService>().AsSingle().NonLazy(
);
        #else

Container.BindInterfacesAndSelfTo<StandaloneInputService>().AsSingle().NonL
azy();
        #endif
            }
        }
    }
```

MainSceneInstaller.cs

```
using Screens.LobbyPopup;
using Screens.MainMenuScreen;
using Utils;
using Zenject;
namespace Installers
{
    public class MainSceneInstaller : MonoInstaller
    {
        public override void InstallBindings()
        {
            BindScreens();
        }
        private void BindScreens()
        {
            Container.BindViewAndPresenter<MainMenuScreenView,
MainMenuScreenPresenter>();
```

```
                Container.BindViewAndPresenter<LobbyPopupView,
LobbyPopupPresenter>();
            }
        }
    }
```

ProjectContextInstaller.cs

```
using Core.PUN;
using Data;
using Data.DataProxy.Player;
using Infrastructure.GameStateMachine;
using Infrastructure.SceneManagement;
using Screens;
using Screens.LoadingsScreen;
using Services.SaveLoad;
using Zenject;
namespace Installers
{
    public class ProjectContextInstaller : MonoInstaller
    {
        public override void InstallBindings()
        {

Container.BindInterfacesAndSelfTo<GameStateMachine>().AsSingle().NonLazy(
);

Container.BindInterfacesAndSelfTo<SceneLoader>().AsSingle().NonLazy();

Container.BindInterfacesAndSelfTo<DataController>().AsSingle().NonLazy();
```

```
Container.BindInterfacesAndSelfTo<ScreenNavigationSystem>().AsSingle().Non
Lazy();

Container.Bind<LoadingScreen>().FromComponentInHierarchy().AsSingle();
            BindServices();
            BindDataProxies();
            BindPhoton();
        }
        private void BindServices()
        {

Container.BindInterfacesAndSelfTo<PlayerPrefsSaveLoadService>().AsSingle().
NonLazy();
        }
        private void BindDataProxies()
        {

Container.BindInterfacesAndSelfTo<PlayerDataProxy>().AsSingle().NonLazy();
        }
        private void BindPhoton()
        {

Container.BindInterfacesTo<PunManager>().FromNewComponentOnNewGameO
bject().AsSingle().NonLazy();

Container.BindInterfacesAndSelfTo<Connector>().AsSingle().NonLazy();

Container.BindInterfacesAndSelfTo<Matchmaker>().AsSingle().NonLazy();
        }
```

```
    }
}
```

LoadingScreen.cs

```
using UnityEngine;
namespace Screens.LoadingsScreen
{
    public class LoadingScreen : MonoBehaviour
    {
        public void Show()
        {
            gameObject.SetActive(true);
        }
        public void Hide()
        {
            gameObject.SetActive(false);
        }
    }
}
```

LobbyPopupPresenter.cs

```
using Core.PUN;
using Infrastructure.SceneManagement;
using Zenject;
namespace Screens.LobbyPopup
{
    public class LobbyPopupPresenter : IInitializable
    {
```

```
            private readonly LobbyPopupView _view;
            private readonly Matchmaker _matchmaker;
            public LobbyPopupPresenter(LobbyPopupView view, Matchmaker
matchmaker)
            {
                _view = view;
                _matchmaker = matchmaker;
            }
            public void Initialize()
            {
                _view.ClickedQuickStartButton += delegate {
_matchmaker.JoinRandomRoom(SceneName.Level1Scene.ToString()); };
            }
        }
    }
```

LobbyPopupView.cs

```
using System;
using UniRx;
using UnityEngine;
using UnityEngine.UI;
namespace Screens.LobbyPopup
{
    public class LobbyPopupView : ScreenView
    {
        [SerializeField] private Button quickStartButton;

        public event Action ClickedQuickStartButton;
```

```
            private new void Awake()
            {
                base.Awake();

                quickStartButton.OnClickAsObservable().Subscribe(delegate {
ClickedQuickStartButton?.Invoke(); })
                    .AddTo(this);
            }
        }
    }
```

MainMenuScreenPresenter.cs

```
    using Screens.LobbyPopup;
    using Zenject;
    namespace Screens.MainMenuScreen
    {
        public class MainMenuScreenPresenter : IInitializable
        {
            private readonly MainMenuScreenView _view;
            private readonly ScreenNavigationSystem _screenNavigationSystem;
            public MainMenuScreenPresenter(MainMenuScreenView view,
ScreenNavigationSystem screenNavigationSystem)
            {
                _view = view;
                _screenNavigationSystem = screenNavigationSystem;
            }
            public void Initialize()
            {
```

```
            _view.ClickedMultiplayerButton += delegate {
_screenNavigationSystem.Show<LobbyPopupView>(); };
        }
    }
}
```

MainMenuScreenView.cs

```
using System;
using UniRx;
using UnityEngine;
using UnityEngine.UI;
namespace Screens.MainMenuScreen
{
    public class MainMenuScreenView : MonoBehaviour
    {
        [SerializeField] private Button multiplayerButton;
        public event Action ClickedMultiplayerButton;
        private void Awake()
        {
            multiplayerButton.OnClickAsObservable().Subscribe(delegate {
ClickedMultiplayerButton?.Invoke(); })
                .AddTo(this);
        }
    }
}
```

ScreenNavigationSystem.cs

```
using System;
```

```
using System.Collections.Generic;
namespace Screens
{
    public class ScreenNavigationSystem
    {
        private readonly Stack<ScreenView> _navigationStack = new();
        private readonly Dictionary<Type, ScreenView> _availableScreens = new();
        public void AddScreen(Type type, ScreenView screenPresenter)
        {
            _availableScreens.Add(type, screenPresenter);
        }
        public void Show<TScreenType>(object extraData = null, bool setAsLastSibling = true)
        {
            Show(typeof(TScreenType), extraData, setAsLastSibling);
        }
        public void Close<TScreenType>() where TScreenType : ScreenView
        {
            if (_availableScreens.TryGetValue(typeof(TScreenType), out ScreenView screen))
            {
                Close(screen);
            }
        }
        public void ClearAvailableScreens()
        {
            _availableScreens.Clear();
        }
        public void ForceCloseAllScreens()
```

```
        {
            int stackCount = _navigationStack.Count;
            while (stackCount-- != 0)
            {
                ForceCloseCurrentScreen();
            }
        }
        private void AddScreenToStack(ScreenView screenPresenter)
        {
            screenPresenter.OnClose = delegate { Close(screenPresenter); };
            if (_navigationStack.Count != 0 && _navigationStack.Peek() ==
screenPresenter) return;
            _navigationStack.Push(screenPresenter);
        }
        private void Show(Type screenType, object extraData = null, bool
setAsLastSibling = true)
        {
            if (_navigationStack.Count != 0 &&
!_availableScreens.ContainsKey(screenType))
            {
                return;
            }
            ScreenView screenPresenter = _availableScreens[screenType];
            screenPresenter.ShowOnPosition(setAsLastSibling);
            AddScreenToStack(screenPresenter);
            screenPresenter.OnShowCallback?.Invoke(extraData);
        }
        private void Close(ScreenView screenPresenter)
        {
            if (!_navigationStack.Contains(screenPresenter)) return;
```

```
            screenPresenter.MoveToInitialPosition();
            RemoveScreenFromStack(screenPresenter);
        }
        private void RemoveScreenFromStack(ScreenView screenPresenter)
        {
            if (_navigationStack.Count == 0) return;
            ScreenView peek = _navigationStack.Peek();
            if (peek == screenPresenter)
            {
                _navigationStack.Pop();
            }
        }
        private void ForceCloseCurrentScreen()
        {
            if (_navigationStack.Count == 0) return;
            ScreenView peek = _navigationStack.Peek();
            if (peek == null) return;
            peek.MoveToInitialPosition();
            RemoveScreenFromStack(peek);
        }
    }
}
```

ScreenView.cs

```
using System;
using UniRx;
using UnityEngine;
using UnityEngine.UI;
using Zenject;
```

```
namespace Screens
{
    [RequireComponent(typeof(CanvasGroup))]
    public class ScreenView : MonoBehaviour
    {
        [SerializeField] private Button closeButton;
        private CanvasGroup _canvasGroup;
        private ScreenNavigationSystem _screenNavigationSystem;
        public Action<object> OnShowCallback;
        public Action OnHideCallback;
        public Action OnClickedCloseButton;
        public Action OnClose;
        [Inject]
        private void Construct(ScreenNavigationSystem screenNavigationSystem)
        {
            _screenNavigationSystem = screenNavigationSystem;
        }
        public void ShowOnPosition(bool setAsLastSibling = true)
        {
            if (setAsLastSibling)
            {
                transform.SetAsLastSibling();
            }
            ActivateScreen();
        }
        public void MoveToInitialPosition()
        {
            OnHideCallback?.Invoke();
            DeactivateScreen();
```

```
}
protected void Awake()
{
    if (closeButton != null)
    {
        closeButton.OnClickAsObservable().Subscribe(delegate
        {
            OnClickedCloseButton?.Invoke();
            OnClose();
        }).AddTo(this);
    }
    AddScreenToAvailableScreens();
    _canvasGroup = GetComponent<CanvasGroup>();
    _canvasGroup.alpha = 0;
    _canvasGroup.blocksRaycasts = false;
}
private void Start()
{
    DeactivateScreen();
}
private void AddScreenToAvailableScreens()
{
    _screenNavigationSystem.AddScreen(GetType(), this);
}
private void ActivateScreen()
{
    _canvasGroup.alpha = 1;
    _canvasGroup.blocksRaycasts = true;
    gameObject.SetActive(true);
}
```

```
        private void DeactivateScreen()
        {
            _canvasGroup.alpha = 0;
            _canvasGroup.blocksRaycasts = false;
            gameObject.SetActive(false);
        }
    }
}
```

IInputService.cs

```
using System;
namespace Services.GameInput
{
    public interface IInputService
    {
        float Player1MovementDirectionX { get; }
        IObservable<bool> Player1JumpAction { get; }

        float Player2MovementDirectionX { get; }
        IObservable<bool> Player2JumpAction { get; }
    }
}
```

MobileInputService.cs

```
using System;
namespace Services.GameInput
{
    public class MobileInputService : IInputService
```

```
        {
            public float Player1MovementDirectionX { get; }
            public IObservable<bool> Player1JumpAction { get; }
            public float Player2MovementDirectionX { get; }
            public IObservable<bool> Player2JumpAction { get; }
        }
    }
```

StandaloneInputService.cs

```
    using System;
    using UniRx;
    using UnityEngine;
    using Zenject;
    namespace Services.GameInput
    {
        public class StandaloneInputService : IInputService, IInitializable, IDisposable
        {
            private readonly Subject<bool> _player1JumpAction = new();
            private readonly Subject<bool> _player2JumpAction = new();
            private readonly CompositeDisposable _compositeDisposable = new();
            public float Player1MovementDirectionX => GetMovementDirection(KeyCode.A, KeyCode.D);
            public IObservable<bool> Player1JumpAction => _player1JumpAction;
            public bool IsPlayer1JumpButtonDown => Input.GetKeyDown(KeyCode.W);
            public float Player2MovementDirectionX => GetMovementDirection(KeyCode.LeftArrow, KeyCode.RightArrow);
            public IObservable<bool> Player2JumpAction => _player2JumpAction;
```

```
        public bool IsPlayer2JumpButtonDown => Input.GetKeyDown(KeyCode.UpArrow);
        public void Initialize()
        {
            Observable.EveryUpdate().Subscribe(delegate
            {
                if (Input.GetKeyDown(KeyCode.W))
                {
                    _player1JumpAction.OnNext(true);
                }
                if (Input.GetKeyDown(KeyCode.UpArrow))
                {
                    _player2JumpAction.OnNext(true);
                }
            }).AddTo(_compositeDisposable);
        }
        public void Dispose()
        {
            _compositeDisposable.Dispose();
        }
        private static float GetMovementDirection(KeyCode leftKey, KeyCode rightKey)
        {
            bool leftKeyPressed = Input.GetKey(leftKey);
            bool rightKeyPressed = Input.GetKey(rightKey);
            if (leftKeyPressed && rightKeyPressed)
            {
                return 0;
            }
            if (leftKeyPressed)
```

```
            {
                return -1;
            }
            if (rightKeyPressed)
            {
                return 1;
            }
            return 0;
        }
    }
}
```

ISaveLoadService.cs

```
using Data.Models;

namespace Services.SaveLoad
{
    public interface ISaveLoadService
    {
        public void SaveGameData(GameDataModel gameDataModel);
        public GameDataModel LoadGameData();
    }
}
```

PlayerPrefsSaveLoadService.cs

```
using Data.Models;
using UnityEngine;
```

```
namespace Services.SaveLoad
{
    public class PlayerPrefsSaveLoadService : ISaveLoadService
    {
        private const string GameDataKey = "GameData";
        public void SaveGameData(GameDataModel gameDataModel)
        {
            PlayerPrefs.SetString(GameDataKey,
JsonUtility.ToJson(gameDataModel));
            PlayerPrefs.Save();
        }
        public GameDataModel LoadGameData()
        {
            return
JsonUtility.FromJson<GameDataModel>(PlayerPrefs.GetString(GameDataKey))
??
                GameDataModel.CreateNewGameDataModel();
        }
    }
}
```

IExitableState.cs

```
namespace Utils.StateMachine
{
    public interface IExitableState
    {
        void Exit();
    }
}
```

IPayloadedState.cs

```
namespace Utils.StateMachine
{
    public interface IPayloadedState<TPayload> : IExitableState
    {
        void Enter(TPayload payload);
    }
}
```

IState.cs

```
namespace Utils.StateMachine
{
    public interface IState : IExitableState
    {
        void Enter();
    }
}
```

StateMachine.cs

```
using System;
using System.Collections.Generic;
namespace Utils.StateMachine
{
    public class StateMachine
    {
        private readonly Dictionary<Type, IExitableState> _states = new();
```

```
        private IExitableState _activeState;
        public void Enter<TState>() where TState : class, IState
        {
            IState state = ChangeState<TState>();
            state.Enter();
        }
        public void Enter<TState, TPayload>(TPayload payload) where TState :
class, IPayloadedState<TPayload>
        {
            TState state = ChangeState<TState>();
            state.Enter(payload);
        }
        protected void AddNewState(IExitableState state)
        {
            _states.Add(state.GetType(), state);
        }
        private TState ChangeState<TState>() where TState : class,
IExitableState
        {
            _activeState?.Exit();
            TState state = GetState<TState>();
            _activeState = state;
            return state;
        }
        private TState GetState<TState>() where TState : class, IExitableState
=> _states[typeof(TState)] as TState;
    }
}
```

Extentions.cs

```
using Zenject;
namespace Utils
{
   public static class Extentions
   {
      public static void BindViewAndPresenter<TView, TPresenter>(this
DiContainer container)
      {
    container.Bind<TView>().FromComponentInHierarchy().AsSingle();
    container.BindInterfacesAndSelfTo<TPresenter>().AsSingle().NonLazy();
      }
   }
}
```

Utils.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Utils
{
   public static class Utils
   {
      public static IEnumerable<T> GetEnumValues<T>()
      {
         return Enum.GetValues(typeof(T)).Cast<T>();
      }
   }
}
```