Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Комп’ютерних наук і технологій
(повне найменування факультету)

Комп’ютерні системи та мережі
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалаврський
(ступінь вищої освіти)

на тему: РОЗРОБКА КОМП’ЮТЕРНОЇ СИСТЕМИ ДЛЯ ГРИ В ШАХИ У РЕАЛЬНОМУ ЧАСІ

Виконав(ла): студент(ка) 4 курсу,
групи КНТ-512сп

Спеціальності
123 Комп’ютерна інженерія
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп’ютерна інженерія
(назва освітньої програми (спеціалізації)

РОГАЧ О.В.
(ПРІЗВИЩЕ та ініціали)

Керівник СКРУПСЬКИЙ С.Ю.
(ПРІЗВИЩЕ та ініціали)

Рецензент КОЗІНА Г.Л
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет Комп’ютерних наук і технологій
Кафедра комп’ютерних систем та мереж
Ступінь вищої освіти бакалаврський
Спеціальність 123 Комп’ютерна інженерія
(код і найменування)
Освітня програма (спеціалізація) Комп’ютерна інженерія
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ
Завідувач кафедри КУДЕРМЕТОВ Р.К.
____
«14» квітня 2025 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

РОГАЧА Олександра Вадимовича
(ПРІЗВИЩЕ, ім’я, по батькові)

1. Тема проєкту (роботи) Розробка комп'ютеної системи для гри в шахи у реальному часі.

керівник проєкту (роботи) к.т.н., доцент, СКРУПСЬКИЙ С.Ю. ,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім’я, по батькові)
затверджені наказом закладу вищої освіти від « 08 » квітня 2025 року № 151
2. Строк подання студентом проєкту (роботи) 01.06.2025 р.
3. Вихідні дані до проєкту (роботи) опис предметної області, методи та алгоритми побудови гри шахи у реальному часі.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз предметної області;
2) Проектування архітектури гри;
3) Реалізація системи;
4) Випробовування системи .

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	СКРУПСЬКИЙ С.Ю.		
Нормоконтроль	ЩЕРБАК Н.В.		

7. Дата видачі завдання «14» квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Аналіз предметної області	до 01.04.2025	
2	Загальні вимоги до системи	до 02.04.2025	
3	Функціональні вимоги до системи	до 04.04.2025	
4	Вимоги до користувача	до 05.04.2025	
5	Діаграма використання	до 06.04.2025	
6	Проектування архітектури гри	до 10.04.2025	
7	Створення вертикального зрізу	до 18.04.2025	
8	Реалізація гри	до 15.05.2025	
9	Випробовування системи	до 17.05.2025	
10	Оформлення пояснювальної записки	до 24.05.2025	
11	Проходження нормоконтролю	до 01.06.2025	
12	Перевірка на наявність академічного плагіату	до 03.06.2025	
13	Проходження рецензування	до 10.06.2025	

Студент(ка) ____________ (підпис) Олександр РОГАЧ (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) ____________ (підпис) Степан СКРУПСЬКИЙ (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 89 с., 2 табл., 26 рис., 1 дод., 13 джерел.

3D, .NET, C#, DI, IoC, PHOTON PUN, SOLID, UNITY, ZENJECT, ООП

У ході виконання кваліфікаційної роботи була створена гра "Шахи" у сеттінгу "Стімпанк" з використанням ігрового рушія Unity.

У кваліфікаційній роботі були виконані:

– аналіз предметної області;

– розробка логічної і фізичної структури програмного комплексу для розробки гри;

– розробка мережевого кода для взаємодії між гравцями;

– збереження данних гравців;

– написання unit тестів щоб упевнитися, що код відповідає вимогам архітектури та має очікувану поведінку.

ЗМІСТ

Перелік скорочень та умовних познак 7
Вступ 8
1 Аналіз предметної області 9
1.1 Відеоігри як вид мистецтва 9
1.2 Етапи розробки відеоігор 10
1.3 Ігрові двигуни 15
1.4 Платформа .NET та мова програмування C# 19
1.5 Інтегровані середовища розробки 23
2 Проектування архітектури гри 25
2.1 Загальні вимоги до системи 25
2.2 Функціональні вимоги до системи 26
2.3 Вимоги до користувача 28
2.4 Діаграма використання 29
2.5 Архітектура системи 31
2.5 Аналіз архітектур UI 33
2.6 Принцип інверсії залежностей, використання DI-контейнера Zenject 35
2.7 Налаштування елементів тестування гри 37
3 Реалізація системи 40
3.1 Створення вертикального зрізу 40
3.2 Загальний огляд архітектури 41
3.3 Реалізація шахових фігур 52
3.4 Реалізація спеціальних рухів та використання шаблону «Стратегія» 53

3.5 Реалізація правил гри 55

3.6 Мережевий код і використання шаблону проектування «Фасад» 57

3.7 Налагодження та аналіз продуктивності гри 59

3.8 Локалізація 60

3.9 Реалізація стандартів позначення позицій FEN та PGN 61

3.10 Висновки 62

4 Випробовування системи 64

4.1 Огляд головних вікон гри 64

4.3 Деплой проекту 71

Висновки 73

Перелік джерел посилання 74

Додаток А Лістинги програми 75

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

UI – User Interface, Інтерфейс користувача

ECS – Entity Component System, Сутність-Компонент-Система

CLR – Common Language Runtime, Загальне середовище виконання

CIL – Common Intermediate Language, Загальна проміжна мова

ООП – Об'єктоорієнтоване програмування

VS – Visual Studio

UE – Unreal Engine

CTS – Common Type System, Загальна система типів

JIT – Just In Time, Компіляція під час виконання

DI – Dependency Injection, Впровадження залежностей

FEN – Forsyth–Edwards Notation, Нотація Форсайта–Едвардса

PGN – Portable Game Notation, Універсальний формат запису партій

AOT – Ahead Of Time , Попередня компіляція

IL2CPP – Intermediate Language To C++ , Проміжна мова у C++

ВСТУП

Серед проявів сучасної інформаційної культури вагоме місце посідають комп'ютерні ігри. Не таємниця, що школярі годинами проводять час перед екранами моніторів, захоплюючи планети, знищуючи монстрів, створюючи цивілізації [1].

Комп'ютеризація суспільства розвивається стрімко. Над ігровими комп'ютерними програмами працюють тисячі висококваліфікованих фахівців, намагаючись зробити кожну гру якомога привабливіше й цікавіше. При цьому використовуються різноманітні сюжети [1].

А як щодо того, щоб глянути на відеоігри з нестандартного кута? Їх розробка вимагає колосального багажу знань з найрізноманітніших напрямів. Сучасні ігри поділяються на безліч жанрів — від аркад, шутерів, екшенів і стратегій до симуляторів, квестів, RPG та пригодницьких ігор. Кожен жанр диктує власні вимоги до технічної реалізації. Наприклад, у стратегіях критично важливим є штучний інтелект, адже саме він формує поведінку віртуального опонента, що впливає на геймплей. А от для шутерів акцент зміщується ще й на візуальну складову: анімації, освітлення, рендеринг — усе повинно відповідати стандартам, які задали індустріальні гіганти. Усе це доводить, що розробка ігор — справа далеко не проста [1].

В ході представленої роботи буде розібрано існуючі ігрові рушії, вибрано один із них і на його основі створено онлайн гру Steampunk Chess, яка дозволить насолоджуватися грою в шахи з улюбленним безліччю сеттінгом стімпанку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Відеоігри як вид мистецтва

Відеогра — це електронна гра, в ігровому процесі якої гравець використовує інтерфейс користувача, щоб отримати зворотну інформацію з відеопристрою. Електронні пристрої, які використовуються для того щоб грати, називаються ігровими платформами. Наприклад, до таких платформ належать персональний комп'ютер та гральна консоль. Пристрій введення, який використовується для керування грою, називається ігровим контролером. Це може бути, наприклад, джойстик, клавіатура та мишка, геймпад або сенсорний екран [1].

У 2011 році відеоігри були офіційно визнані видом мистецтва урядом США та Національним фондом мистецтв США. Появі відеоігор передували розвиток програмованих комп'ютерів та технологій формування зображення на екранах електронних пристроїв. Різноманітні електронні та механічні ігрові пристрої існували ще в першій половині XX століття, але не мали досить значного поширення. Попередниками відеоігор є пристрій «Cathode ray-tube Amusement Device», патент на яку Томас Ґолдсміт Молодший та Істл рей Менн отримали 14 грудня 1948 року, і шахова комп'ютерна програма, розроблена у 1947 Аланом Тьюрінгом. Початково ігрові програми, як шахи чи хрестики-нулики, розроблялися в рамках військової програми США у прагненні створити комп'ютер, здатний передбачати дії противника [1].

Перша успішна спроба створити розважальний пристрій, який використовує для зворотного зв'язку із гравцем відео, належить Вільяму Гігінботаму. У 1958 він розробив Tennis For Two, однак не розглядав гру як щось важливе і зрештою розібрав обладнання для інших, наукових, проектів. В 1960-ті студентами Массачусетському технологічному інституту було написано гру Spacewar!, яка у 1966 підштовхнула Sanders Associates на думку про можливість створення грального пристрою, що під'єднувався б до домашніх телевізорів. Коли комп'ютери стали порівняно дешевими і могли використовуватися не лише науковими

установами, студент Стенфорду Білл Піттс, вражений Spacewar!, вирішив, що на основі комп'ютера PDP-11 реально створити пристрій спеціально для ігор — Galaxy Game. Разом з другом Х'ю Таком він розробив ігровий автомат, що за монети надавав можливість пограти і таким чином окупив би себе. За підтримки Нолана Бушнелла з компанії Nutting Associates цей автомат, Computer Space і однойменна гра на ньому 1971 року, став першим комерційним пристроєм з відеогрою. Як екран був використаний чорно-білий телевізор, комп'ютерною основою виступили мікросхеми TTL серії 7400. Вже у 1972 році з'явилась перша домашня гральна консоль, Magnavox Odyssey, яка підключалася до звичайного телевізора. Поштовхом до виникнення індустрії відеоігор стала висока популярність спочатку аркадної, а потім і домашньої відеогри Pong, оскільки її комерційний успіх призвів до появи великої кількості клонів від інших компаній [1].

У 2000-х роках відеоігрова індустрія зробила суттєвий стрибок уперед — графіка стала набагато реалістичнішою, а активний розвиток мережевих технологій відкрив нову еру онлайн-ігор, де гравці могли змагатися або співпрацювати у режимі реального часу. До кінця десятиліття відбулася інтеграція відеоігор із соціальними платформами та ігровими сервісами. Фізичні носії поступово втрачали популярність, а цифрова дистрибуція виходила на перший план [1].

Водночас широкого розповсюдження набули DLC (завантажувані доповнення), включаючи платні, а також мікротранзакції — внутрішньоігрові покупки, що дозволяють прискорити прогрес або отримати ексклюзивні переваги у грі [1].

1.2 Етапи розробки відеоігор

Процес створення гри складається з кількох послідовних стадій. Спершу формується концепція проєкту — визначається загальна ідея, розробляється дизайн-документ, у якому детально описується ігровий світ, механіки, сюжетна

лінія та основні особливості геймплею. Паралельно створюються візуальні концепти персонажів і локацій, що задають стильову направленість майбутньої гри й слугують орієнтиром для команди художників і 3D-моделерів. Окрім внутрішнього використання, ці матеріали часто відіграють ключову роль у презентації проєкту перед потенційними інвесторами чи видавцями [3].

Якщо відсутні проблеми з фінансуванням, починається повноцінна робота над грою, яка включає розробку чи налаштування під потреби гри ігрового рушія, створення графічного, тривимірного, та аудіо наповнення гри, впровадження та тестування ігрових механік (ігровий дизайн). До своєї остаточної версії гра йде через альфа та бета версії, які ретельно перевіряють і тестують, іноді проводячи відкриті тестування за участю гравців (відкритті альфа/бета-тести). Через якийсь час після останніх перевірок і тестувань, гра «виходить на золото», тобто починає тиражуватися для подальшого продажу. Після виходу триває підтримка гри, в ході якої вона вдосконалюється і доповнюється [3].

1.2.1 Проєктування (пре-продакшн)

Перш за все хтось подає ідею створення нової гри, як правило він і стає керівником проєкту. Поступово чи мозковим штурмом вирішується про що вона оповідатиме, яка її особливість може привернути увагу гравців; гра принесе щось кардинально нове або ж використовуватиме вже відомі та перевірені технології, тематику. Виходячи із запланованих переважних дій гравця, обирається жанр, наприклад, стратегія в реальному часі або шутер від першої особи. В ході роботи жанр може змінитися і вся концепція зазнати суттєвої переробки. Геймдизайнер вирішує якими стануть образи персонажів, ігрового світу, виконує попередні малюнки — концепт-арти. Гра отримує попередню назву, яка з часом може уточнитися чи зовсім змінитися [3].

Збирається команда розробників з програмістів, сценаристів, художників, композиторів, менеджерів і так далі. Визначаються строки виконання кожного виду роботи і орієнтовна дата виходу готової гри, яке обладнання, кошти для цього знадобляться. Та сама людина може займатися кількома справами. Наприклад, бути менеджером, координуючи виконанням роботи інших, і дизайнером, розроблячи

образи персонажів. На цьому етапі складається концепт-документ, в якому обґрунтовується чому буде вигідно створити дану гру. Дизайн-документ описує гру загалом, її початковий сюжет, ігровий процес, містить концепт-арти. Також створюється прототип, початкова проста версія гри, з якої можна скласти уявлення яким буде ігровий процес [3].

1.2.2 Виробництво

Оскільки відеогра є комп'ютерною програмою, її робота, технічні можливості, контент та ігровий процес, забезпечується програмним кодом. Розробка гри включає ті ж етапи, що і розробка програмного забезпечення, але передбачає більше роботи над контентом і створення ігрових механік.

Сучасні ігри здебільшого засновані на готових програмних модулях — ігрових рушіях, де вже реалізовані базові функції, здатні зв'язувати воєдино графіку, звук, об'єкти і їх рухи. Щоб налаштувати рушій для реалізації конкретного задуму програмісти доопрацьовують його, додаючи потрібні функції. Існують як вільні ігрові рушії, доступні будь-кому, так і ті, що вимагають отримання ліцензії на їх використання. Крім того рушії різняться за ліцензіями. Для незалежних розробників їх використання може бути значно дешевшим. Деякі рушії розраховані на створення ігор конкретного жанру, інші — універсальні. Не всі рушії можуть забезпечити однакові внутрішньоігрові можливості та рівень графіки. Частина рушіїв дозволяють створювати ігри для різних платформ, так Unreal Development Kit підтримує розробку інтерактивних творів для PC, Xbox 360, PlayStation 3, Wii та Android. Деякі ігри створюються в спеціальних програмах, які вже мають початкові ресурси, дії, та не вимагають знання мов програмування. Прикладами таких програм є Game Maker, Construct, RPG Maker [3].

Створення відеоігор охоплює багато аспектів: графіку, звук, тексти та інтерактивність. Розробка починається зі створення концепт-артів, які визначають стиль гри та слугують орієнтиром для художників. На їх основі створюються 2D або 3D моделі персонажів, предметів і оточення. Анімації додають рухи до моделей за допомогою спеціального програмного забезпечення, іноді із використанням технологій захоплення руху [3].

Візуальні ефекти, як вибухи, тіні чи заломлення світла, підсилюють атмосферу гри. Стилістика може змінюватись від реалістичної до стилізованої, наприклад, під комікс. Графічні та звукові рушії відповідають за візуальну та аудіо-складову. Музика, звуки, діалоги — все це створюється композиторами, звукорежисерами та акторами озвучки, іноді навіть з оркестровим супроводом [3].

Ігрова механіка визначає, як працює геймплей: управління персонажем, взаємодія з об’єктами, користувацький інтерфейс. Рівні або локації проектуються так, щоб зробити гру цікавою та оптимізованою. Фізичний рушій обробляє взаємодію об’єктів згідно з законами фізики, а штучний інтелект керує поведінкою NPC. Події прописуються сценаристами, а їх реалізація виконується через програмні скрипти [3].

1.2.3 Тестування

Після завершення праці над кодом, контентом і механікою, за яких гра може функціонувати, відбувається її доопрацювання. Гра, не зібрана до кінця, але в яку можливо грати, називається альфа-версією. Вона може містити значні помилки і недоопрацювання, як відсутність певних можливостей, музики або об'єктів. Виявленням проблем займаються тестери, котрі грають в цю гру, намагаючись сповна скористатися всіма доступними можливостями в ній. Зазвичай на цьому етапі розробники записують рекламний трейлер, показуючи ігровий процес на відео, даючи потенційним гравцям уявлення про свою роботу. На пізнішому етапі виходить бета-версія, до тестування якої можуть залучатися і потенційні покупці гри. В бета-версії відбувається подальший пошук помилок, перевірка коректності взаємодії об'єктів ігрового світу, управління. Можливі внесення змін в оформлення, зміна ігрового балансу, можливостей персонажів [3].

1.2.4 Випуск і продажі

За продажі як правило відповідає видавець гри, від якого розробник отримує частку отриманих коштів. В разі якщо розробник і видавець є частинами однієї компанії, розподіл прибутків відбувається за встановленими там правилами. Видавець має піар-менеджерів та ігрових журналістів, котрі певним чином заявляють про існування гри, проводять рекламну кампанію, повідомляють про неї,

викладаючи відео, публікуючи новини в спеціалізованих виданнях, демонструючи гру на виставках відеоігор. Видавець займається локалізацією гри, тобто перекладом її тексту і озвучування, адаптацією до законів і культури тої країни, де гра видаватиметься. Для локалізації існують спеціальні команди фахівців, які крім власне перекладу роблять гру зрозумілою для гравця конкретної країни. В сучасних іграх текст і звуки як правило містяться в окремих файлах, що робить локалізацію простішою. Існують офіційні і неофіційні локалізатори. Часто гра від початку має локалізації на найбільш поширені мови, як англійську, іспанську і китайську. Іншими мовами гра офіційно виходить згодом чи цим неофіційно займаються ентузіасти [3].

Завершена і локалізована гра записується на певні носії, як DVD-диски, або надається для завантаження з Інтернету. Диски поставляються в магазини, а версії для завантаження поширюються через сервіси цифрової дистрибуції, як Steam. Цифрова дистрибуція, на відміну від дисків, іноді передбачає і безкоштовне завантаження гри. Так декотрі ігри, які представляють вже тільки історичний інтерес, надаються безкоштовно. В сервісах електронної дистрибуції часто відбуваються акції, які надають знижки аж до 100 % [3].

1.2.5 Підтримка

Після релізу гри можуть виявлятись баги чи недоліки, про які стало відомо вже під час гри користувачами. У відповідь на це розробники випускають оновлення — патчі, які виправляють помилки або вдосконалюють певні аспекти, наприклад, баланс чи рівень складності. Хоча самі патчі не приносять прямого прибутку, вони здатні підвищити репутацію гри та стимулювати продажі [3].

Щоб зберегти інтерес гравців, до гри додаються оновлення у вигляді доповнень — нових сюжетних гілок, локацій, персонажів або ігрових функцій. Вони можуть бути як безкоштовними, так і платними. Формат DLC (Downloadable Content), тобто контенту, що завантажується через Інтернет, найчастіше передбачає платні розширення, які додають косметичні елементи або додаткові можливості без радикальної зміни основного геймплею [3].

1.3 Ігрові двигуни

Ігровий двигун — програмний рушій, центральна програмна частина будь-якої відеогри, яка відповідає за всю її технічну сторону, дозволяє полегшити розробку гри шляхом уніфікації та систематизації її внутрішньої структури. Важливим значенням двигуна є можливість створення багатоплатформових ігор (сьогодні найчастіше одночасно для ПК, PS4 та Xbox One). Основну функціональність гри зазвичай забезпечує її двигун, до якого входить двигун рендерингу («візуалізатор»), фізичний двигун, звук, система скриптів, анімація, ігровий штучний інтелект, мережевий код, керування пам'яттю, багатонитевість і граф сцени. Часто на процесі розробки можна заощадити шляхом повторного використання одного двигуна гри для створення декількох різних ігор [4].

На додаток до багаторазово використовуваних програмних компонентів, ігрові двигуни надають набір візуальних інструментів для розробки. Ці інструменти зазвичай складають інтегроване середовище розробки для спрощеної, швидкої розробки ігор на зразок потокового виробництва. Такі рушії іноді називають «ігровим підпрограмним забезпеченням», тому що, з погляду бізнесу, вони надають гнучку й багаторазово використовувану програмну платформу з усією необхідною функціональністю для розробки грального застосунку, скорочуючи витрати, складність і час розробки — усі критичні фактори в сильно конкурентній індустрії відеоігор. Як і інші рішення програмного забезпечення, ігрові двигуни зазвичай платформонезалежні й дозволяють деякій грі запускатися на різних платформах, включаючи гральні консолі й персональні комп'ютери, з деякими внесеними у початковий код змінами. Часто гральне ППЗ має компонентну архітектуру, що дозволяє замінювати або розширювати деякі системи рушія спеціалізованішими компонентами ППЗ, наприклад Havok — для фізики, FMOD — для звуку або SpeedTree — для рендерингу. Деякі рушії, такі як RenderWare, проєктують як набір слабо зв'язаних компонентів ППЗ, які можуть вибірково комбінуватися для створення власного рушія, замість традиційнішого

підходу розширення або налаштовування гнучкого інтегрованого рішення. Проте розширюваність досягнута й залишається високопріоритетною в ігрових рушіях через широкі можливості їхнього застосування. Попри специфічність назви, ігрові рушії часто використовуються в інших типах інтерактивних застосунків, що вимагають графіку в реальному часі, таких як рекламні демовідео, архітектурні візуалізації, навчальні симулятори й середовища моделювання [4].

Більшість сучасних 3D рушіїв ґрунтуються на графічних API, таких як Direct3D або OpenGL, які забезпечують доступ до відеокарти через програмну абстракцію. Окрім них, у розробці ігор застосовуються низькорівневі бібліотеки, як-от DirectX, SDL чи OpenAL, що дають змогу взаємодіяти з іншими пристроями комп’ютера — клавіатурою, мишею, джойстиком, звуковими й мережевими картами [4].

До появи графічних процесорів із підтримкою 3D-прискорення широко використовувався програмний рендеринг — обчислення зображення лише за допомогою процесора. Сьогодні такий метод все ще актуальний у випадках, коли потрібна висока візуальна точність або коли графічне обладнання не підтримує сучасні функції, як-от шейдери [4].

1.3.1 Unity 3D

Ціна: Personal - free, Plus - 35 доларів за робоче місце на місяць, Pro - 125 доларів за робоче місце на місяць, Enterprise - за домовленістю.

Ісходний код: закритий

Платформи для використання: Windows, macOS

Завдяки низькому порогу входження, зручності використання це, безсумнівно, найпопулярніший двигун в промисловості. За останні три роки він виріс у ще потужнішу платформу для створення ігор. Unity містить у собі всі мислимі інструменти для розробки ігрових додатків. За допомогою Unity 3D можна створювати і тривимірні, і двовимірні ігри Windows PC, Mac. Одна з найпрекрасніших можливостей Unity 3D - це експортування гри для будь-якої з 21 підтримуваної платформи, серед них: iOS, Android, Windows 8 Store, Windows 10 Store, MacOS, PS3, PS4, Xbox 360, Xbox One, Wii U, Oculus Rift , Gear VR,

PlayStation VR, Samsung Smart TV. З цього списку видно, що підтримуються не лише десктопи, мобільні пристрої, консолі та браузера, а й розумні телевізори та шоломи віртуальної реальності [5].

Unity відрізняє також величезну спільнотою і колосальним магазином компонентів для двигуна - Asset Store, що продає елементи для ігор: моделі, текстури, скрипти, редактори, інструменти, серверні підсистеми. Все це створюють користувачі движка - учасники спільноти.

1.3.2 Unreal Engine

Ціна: безкоштовно (5% роялті від продажу проекту).

Вихідний код: відкритий.

Платформи для використання: Windows, macOS.

За допомогою UE4 можна розробляти як двовимірні, так і тривимірні ігри абсолютно будь-яких жанрів: шутери, стратегії, квести, RPG, симулятори, використовується він і в освітніх цілях. UE4 пропонує багато заготовок для ігор різних жанрів. Є два шляхи створення проекту: Blueprint або C++. У першому випадку використовується графічний скриптовий мову, де описи і послідовність дій укладаються подібно до кінцевого автомата. Тобто реалізовані в движку операції пов'язуються у вигляді графічного інтерфейсу. У другий випадок опис геймплея складає мові C++ [6].

Прямо із коробки за допомогою темплейтів можна створити 13 проектів на C++ та 12 на Blueprint. Серед них ігри різних жанрів: двовимірні скролери, шутери від першої та третьої особи, авіа- та автосимулятори. У тринадцяте оновлення було додано заготівлю для віртуальної реальності [6].

Двигун підтримує велику різноманітність сучасних технологій. Навіть перераховувати немає сенсу — ігри, розроблені на UE4, охоплюють всі платформи та ігрові пристрої (VR, Kinect, Leap Motion тощо).

UE4 має всі необхідні редактори: це конструювання сцен, імпортування, налаштування та анімація моделей, накладання матеріалів, створення фізичних ефектів, розміщення аудіоджерел, налаштування звуку, спеціальний редактор для створення скриптів Blueprint та багато іншого.

1.3.3 CryEngine

Ціна: безкоштовно.

Вихідний код: відкритий.

Платформи для використання: Windows.

CryEngine — це гральний рушій розроблений німецькою компанією Crytek, який вперше був використаний у відеогрі Far Cry. Він був створений як технологічна демонстрація GeForce 3 від Nvidia. Ubisoft же має модифіковану версію CryEngine у грі Far Cry, який називається «Dunia Engine». CryEngine V підтримує Windows, Linux, PlayStation 4, Xbox One. Він головним чином орієнтований на MMO-ігри з передовою графікою. Також у движку добре опрацьована мережева підсистема, є підтримка DirectX 12, PhysX та широченого ряду інших технологій [7].

CryEngine має свій Marketplace, де будь-який користувач може купувати і продавати ігрові ассети.

1.3.4 Godot

Ціна: безкоштовно.

Вихідний код: відкритий.

Платформи для використання: Windows, Linux, OS X.

Godot — це безплатний і відкритий ігровий рушій, який дозволяє створювати 2D та 3D ігри для різних платформ. Він має власну мову програмування GDScript, схожу на Python, а також підтримує C#, C++ та інші мови. Godot вирізняється зручним візуальним редактором, гнучкою системою сцени і активною спільнотою.

Поточна підтримка платформ включає Windows, Linux, OS X, BSD, Haiku, Android, iOS, BlackBerry 10, HTML5. Також можна проводити експорт на інші платформи вручну через компілювання рушія з SDK цільової платформи. Використання Godot незначної кількості зовнішніх бібліотек значно полегшує цей процес [8].

1.3.5 Вибір ігрового двигуна

Розібравши всі сильні та слабкі сторони таких двигунів, як:

– Unity;

– Unreal Engine 4;

– CryEngine;

– Lumberyard;

– Godot.

А також, зробивши аналіз можливостей популярних ігрових рушіїв, можна дійти висновку, що Unity майже ідеально підходить для розробки гри. Рушій підтримує кросплатформенність, має велику спільноту, безліч плагінів і ресурсів, а також зручний інтерфейс для створення 2D та 3D ігор.Єдиним серйозним недоліком можна вважати певну повільність при розробці великих або складних проєктів. Це обумовлено тим, що рушій працює з мовою програмування, яка має систему збору сміття, що іноді призводить до зайвого споживання ресурсів і падінь продуктивності. Однак, завдяки правильній архітектурі проєкту, оптимізації коду, використанню пулінгу об’єктів і відмові від надмірного створення об’єктів у реальному часі — ці недоліки можна успішно мінімізувати.

1.4 Платформа .NET та мова програмування C#

C# був обраний для розробки проекту, оскільки Unity підтримує цю мову програмування як основну для створення ігрової логіки. C# є частиною платформи .NET, яка забезпечує зручне середовище для розробки програм. Програми, написані на C#, працюють у середовищі .NET, яке забезпечує автоматичне управління ресурсами, обробку винятків і збирання сміття, що робить розробку на C# більш ефективною та зручною.

Завдяки сумісності мов у .NET, код, написаний на C#, може взаємодіяти з кодом, створеним на інших мовах, таких як F#, Visual Basic або C++. Це забезпечує гнучкість і масштабованість проектів. Крім того, .NET включає великі бібліотеки, які організовані в простори імен і надають широкий спектр функцій для розробки різних типів програм.

1.4.1 Компоненти .NET Framework

.NET Framework містить три компоненти, які визначають його структуру: виконавче середовище, інструменти програмування та бібліотеку базових класів (рис. 1.1) [9].

Середовище виконання називають Common Language Runtime (CLR). Воно керує виконанням програм, зокрема: виконує розподіл пам'яті та прибирає "сміття", перевіряє безпечність коду, виконує код, керує потоками виконання та обробкою помилок тощо [9].

Рисунок 1.1 – Компоненти .NET Framework та їх взаємодія

Інструменти програмування є засобами розробки, які поєднують все необхідне для створення та налагодження програм. Вони містять:

– інтегроване середовище розробки Microsoft Visual Studio;

– сумісні з .NET компілятори (C#, Visual Basic .NET, F#);

– налагоджувачі;

– серверні технології для WEB-розробки, такі як ASP.NET.

Бібліотека базових класів (Base Class Library, BCL) містить великий набір класів, який використовується системою .NET Framework і доступний при розробці програмного забезпечення. Сюди входять класи для роботи з різними типами даних, файлами, шифрування; класи колекцій; класи для роботи з потоками та інші класи [9].

1.4.2 Компіляція коду в .NET Framework

Компілятор кожної з мов .NET на основі вихідного коду програми формує вихідний файл, який називають збіркою (assembly). Збірка містить не безпосередній машинний код, а проміжний код мовою Common Intermediate Language. Код збірки містить, крім програмного коду мовою CIL, метадані з описом типів, посилань на інші збірки, інформацію щодо безпеки виконання тощо [9].

Процесор комп'ютера може виконувати виключно власний машинний код. Відповідно, код CIL-код збірки для виконання має бути перетворений у процесорний код. Це здійснюється тільки при виконанні збірки. Виконавчий код збірки компілюється компілятором часу виконання (Just-in-Time compiler), який є складовою частиною виконавчого середовища CLR. Потім цей код кешується, для випадку, якщо він ще раз буде потрібний для виконання [9].

Так як Windows підтримує виконання застосунків, розроблених не для .NET Framework, то з точки зору системи .NET весь код ділять на дві частини:

– керований код (managed code);

– некерований код (unmanaged code).

Різницю виконання керованого та некерованого коду демонструє схема на рис. 1.2

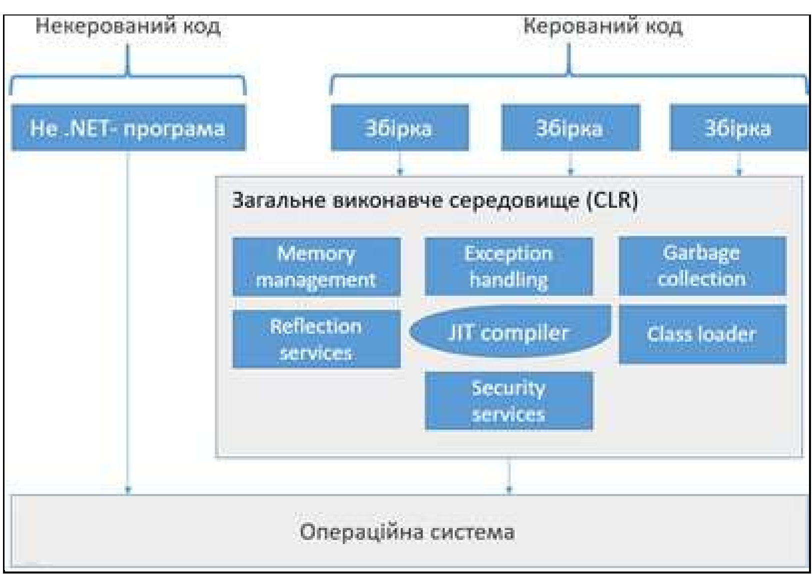


Рисунок 1.2 – Виконання керованого та некерованого коду

Описаний процес компіляції та виконання ілюструє схема на рис. 1.3. Програмний код, написаний однією з мов .NET Framework, компілюється у проміжний код CIL. При необхідності його виконання, задіюється загальне виконавче середовище CLR, і за допомогою компілятора часу виконання формується процесорний код, який передається для виконання внутрішньому механізмові операційної системи Windows [9].

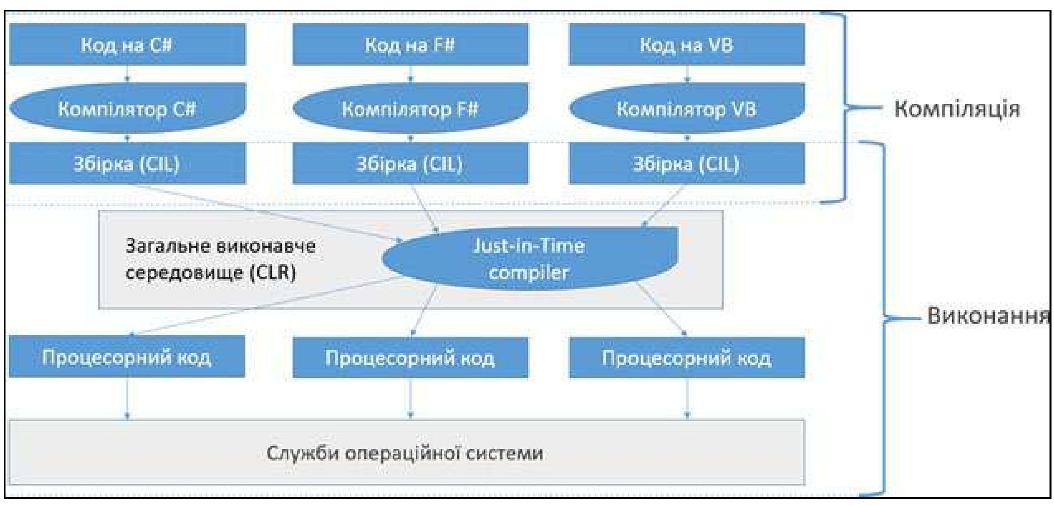


Рисунок 1.3 – Схема процесу компіляції коду для .NET і його виконання

1.4.3 Мова програмування C#

C# — це об'єктно-орієнтована, компонентно-орієнтована мова програмування. C# надає мовні конструкції для безпосередньої підтримки цих концепцій, роблячи C# природною мовою для створення та використання програмних компонентів. З моменту свого виникнення в C# були додані функції для підтримки нових робочих навантажень і нових методів проектування програмного забезпечення. За своєю суттю C# є об'єктно-орієнтованою мовою. Ви визначаєте типи та їх поведінку [9].

Функції C# допомагають створювати надійні та довговічні програми. Збір сміття автоматично відновлює пам’ять, зайняту недоступними невикористаними об’єктами. Типи , які допускають значення NULL, захищають від змінних, які не посилаються на виділені об’єкти. Обробка винятків забезпечує структурований і розширений підхід до виявлення та відновлення помилок. Лямбда-вирази

підтримують методи функціонального програмування. Синтаксис мовного інтегрованого запиту (LINQ) створює загальний шаблон для роботи з даними з будь-якого джерела. Підтримка мови для асинхронних операцій забезпечує синтаксис для побудови розподілених систем. C# має уніфіковану систему типів. Усі типи C#, включаючи примітивні типи, такі як inti double, успадковуються від одного кореневого object типу. Усі типи мають набір загальних операцій. Цінності будь-якого типу можна зберігати, транспортувати та використовувати узгоджено. Крім того, C# підтримує як визначені користувачем типи посилань, так і типи значень . C# дозволяє динамічно розподіляти об'єкти та вбудовано зберігати полегшені структури. C# підтримує загальні методи та типи, які забезпечують підвищену безпеку та продуктивність типів. C# надає ітератори, які дають змогу реалізаторам класів колекції визначати користувацьку поведінку для клієнтського коду [9].

C# робить акцент на версійності , щоб програми та бібліотеки могли розвиватися з часом сумісним чином. Аспекти дизайну C#, на які безпосередньо вплинули міркування щодо версій, включають окремі virtual і override модифікатори, правила вирішення перевантаження методів і підтримку явних декларацій членів інтерфейсу [9].

1.5 Інтегровані середовища розробки

Інтегровані середовища розробки (ІСР) створені для того, щоб зробити процес розробки програмного забезпечення більш зручним і продуктивним. Вони об’єднують усі необхідні інструменти для написання, компілювання, налагодження та розгортання програм в одному додатку. Це дозволяє розробнику працювати в єдиному середовищі, без необхідності налаштовувати окремі інструменти, як у традиційному підході. Завдяки цьому зменшується час на конфігурацію, а продуктивність програміста зростає.

ІСР також забезпечують інтеграцію між своїми компонентами, наприклад, синтаксичний аналіз коду може виконуватися під час його написання, що дозволяє виявляти помилки ще до компіляції. Деякі середовища розробки створені для конкретних мов програмування, надаючи спеціалізовані функції, які відповідають особливостям цих мов. Прикладами таких середовищ є PhpStorm, Xcode, Xojo та Delphi.

1.5.1 Visual Studio

Visual Studio – інтегроване середовище розробки програмного забезпечення від фірми Microsoft. Дане середовище дозволяє створювати різноманітні програмні продукти: консольні програми, програми з графічним інтерфейсом, наприклад віконні додатки Windows Forms, а також Web-додатки тощо [9].

Середовище Visual Studio дозволяє розробляти додатки, використовуючи різні мови програмування: Visual C#, Visual Basic, Visual F#, Visual C++, Python і т.д. Також існує можливість розробляти додатки не тільки під Windows, а і під інші популярні платформи: Android, iOS. Версія Visual Studio Community є абсолютно безкоштовною для учнів, студентів та розробників програм з відкритим програмним кодом.

1.5.2 JetBrains Rider

Rider — це кросплатформена інтегрована середовище розробки програмного забезпечення для платформи .NET, розроблена компанією JetBrains. Підтримуються мови програмування C#, VB.NET та F#. В основі Rider лежить інший продукт JetBrains — ReSharper. Середовище підтримує платформи .NET Framework, .NET та Mono. Воно працює на операційних системах Windows, macOS, Linux.

1.5.3 Вибір IDE

Зараз все більше компаній які працюють з .NET та Unity починають переходити на IDE JetBrains Rider через численну низку переваг.

Переваги над Visual Studio:

– Rider на відміну від Visual Studio(VS), 64-бітна IDE;

– JetBrains Rider є кросплатформним, він може працювати на платформах

Windows, Mac або Linux з однаковою функціональністю та стабільністю;

– середовище Rider включає більшість функцій популярного розширення Visual Studio для розробників .NET - ReSharper;

– у Rider є безліч функцій, успадкованих від платформи IntelliJ;

– підтримка систем контролю версій Git, Mercurial, CVS та Subversion;

– можливість інтеграції з багатьма трекерами проблем, такими як Team Foundation Server і Visual Studio Team Services. Також він підтримує JIRA Software, YouTrack та інші рішення;

– рішення та проекти, з якими працює JetBrains Rider, повністю сумісні з Visual Studio, і не використовують пропрієтарні формати.

Було вирішено що JetBrains Rider прискорить розробку гри та допоможе зробити її краще.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ГРИ

2.1 Загальні вимоги до системи

Під час розробки гри важливо враховувати загальні вимоги, що забезпечують якісну, стабільну та зручну взаємодію користувача з додатком. Ці вимоги не залежать від конкретного ігрового функціоналу, проте є критичними для повноцінного використання продукту та його подальшого розвитку.

До таких вимог належать:

– кросплатформеність;

– продуктивність;

– зручність інтерфейсу;

– масштабованість.

Гра повинна запускатися принаймні на одній з основних платформ, таких як Windows, Android або iOS. Також, потрібно забезпечити мінімальний час відгуку системи при виконанні дій користувача, що є особливо важливим у мережевому

режимі. Інтерфейс має бути зрозумілим і привабливим для користувача, забезпечуючи комфортну взаємодію з грою. Також, у майбутньому повинна бути можливість розширити функціонал.

2.2 Функціональні вимоги до системи

Для досягнення поставлених загальних вимог, система повинна реалізувати такі функціональні можливості:

– реалізація правил класичних шахів, включно з усіма основними фігурами та спеціальними ходами (рокіровка, взяття на проході, шах, мат, пат);

– підтримка мультиплеєрного режима гри, створення кімнат для мережевої гри з різними налаштуваннями, відображення кімнат у вікні лоббі.

– синхронізація стану гри між двома гравцями у мультиплеєрному режимі;

– система авторизації/ідентифікації гравців;

– ведення історії ходів для перегляду або можливості повернення до попередніх кроків (undo/redo);

– базова система оцінки результатів (перемога, нічия, поразка);

– вікно з налаштуваннями, що включає налаштування графіки, мови, а також аудіо;

– реалізація головного меню;

– реалізація базових аудіо еффектів;

– збереження прогресу.

Особливу увагу було приділено забезпеченню зручності гравця, стабільності ігрового процесу та точності відтворення шахових правил.

Для кращої візуалізації функціональної структури системи на етапі планування було створено майндмапу, що слугувала орієнтиром під час реалізації проєкту. Вона скаладається з чотирьох основних пунктів, відображає ключові функціональні компоненти системи та допомагає зрозуміти загальну

архітектуругри. Майндмап наведена на рисунку 2.1.

Також, на основі аналізу вимог до проєкту було визначено ключові етапи його реалізації та встановлено чіткі часові рамки для кожного з них. Це дозволило раціонально організувати робочий процес, забезпечити рівномірне навантаження протягом усього періоду розробки та уникнути накопичення завдань на фінальних стадіях. Кожен етап мав конкретну мету та набір завдань, що сприяло збереженню фокусу на важливих функціональних аспектах системи. Крім того, поділ на етапи полегшив контроль за виконанням плану та вчасне виявлення можливих проблем.

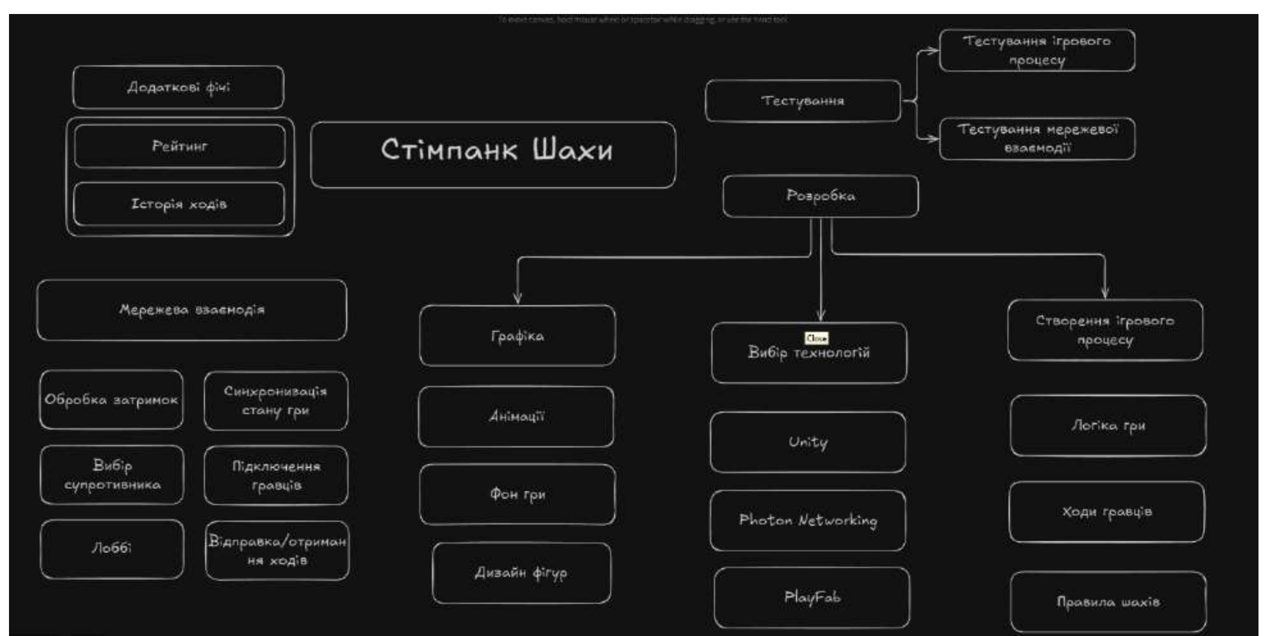


Рисунок 2.1 – Майндмап проекта

Завдяки цьому підходу вдалося не лише оптимізувати використання часу та ресурсів, а й забезпечити своєчасне виявлення проблемних моментів, їхнє усунення та внесення необхідних змін без значних затримок. Чітке планування дозволило ефективно координувати роботу, зменшити кількість непередбачених труднощів та досягти поставлених цілей у межах встановленого графіка. Візуальне представлення етапів реалізації проєкту, їхньої послідовності та тривалості відображено на діаграмі планування, що наведена на рисунку 2.2. Вона допомагає краще зрозуміти загальну структуру робіт, взаємозв'язки між завданнями та ключові контрольні точки в процесі розробки.

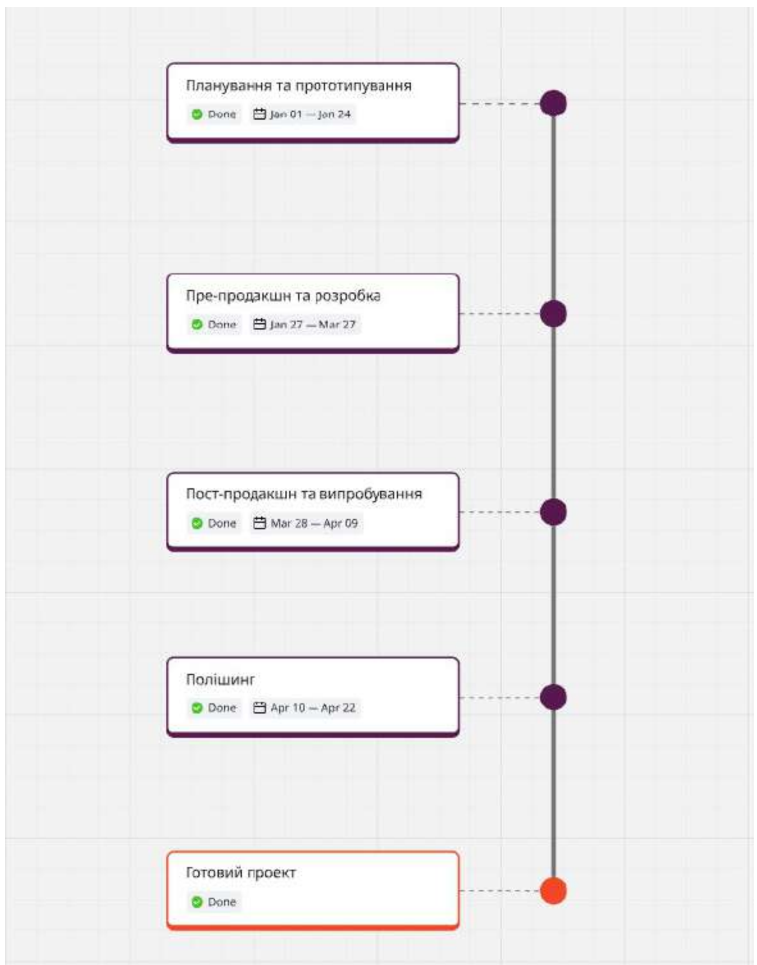


Рисунок 2.2 – Етапи реалізації проекта

2.3 Вимоги до користувача

Потенційний користувач гри має відповідати мінімальним вимогам: користувачі повинні мати пристрій, що відповідає технічним характеристикам гри, стабільне підключення до інтернету для мультиплеєрного режиму, а також операційну систему, сумісну з грою (Windows). Для забезпечення комфортного і безперебійного ігрового процесу рекомендується використовувати пристрої, що перевищують мінімальні технічні характеристики. Рекомендовані та мінімальні вимоги до користувача наведено в таблицях 2.1 та 2.2. Ці вимоги допомагають гарантувати якісну роботу системи та уникнути можливих технічних проблем під час гри.

Таблиця 2.1 – Мінімальні вимоги до користувача

Компонент	Мінімальні вимоги
Процесор	Двоядерний процесор 2.4 Ггц
Оперативна пам'ять	4 ГБ
Операційна система	Windows
Графічний процесор	Radeon HD 7000 або Nvidia GeForce GTX 500
Місце на диску	2 гб
Підключення до Інтернет (тільки при мультиплеєрній грі)	Стабільне підключення, швидкість 2 Мбіт/с

Таблиця 2.2 – Рекомендовані вимоги до користувача

Компонент	Рекомендованні вимоги
Процесор	Intel Core i5-4460 або AMD Ryzen 3 1200
Оперативна пам'ять	8 ГБ
Операційна система	Windows
Графічний процесор	Radeon RX 500 або Nvidia GeForce GTX 900
Місце на диску	4 гб
Підключення до Інтернет (тільки при мультиплеєрній грі)	Стабільне підключення, швидкість 2 Мбіт/с

2.4 Діаграма використання

Для розробки якісного програмного забезпечення важливо не лише враховувати функціональні та загальні вимоги до системи, але й чітко визначити сценарії взаємодії користувача з додатком. Діаграма використання дозволяє візуалізувати основні етапи роботи програми, а також логіку переходів між її

компонентами. Вона допомагає зрозуміти, як користувач взаємодіє з системою, і забезпечує основу для проектування інтуїтивного інтерфейсу. У цьому розділі представлено діаграму використання гри(рис 2.3), яка демонструє структуру додатка та послідовність дій користувача.

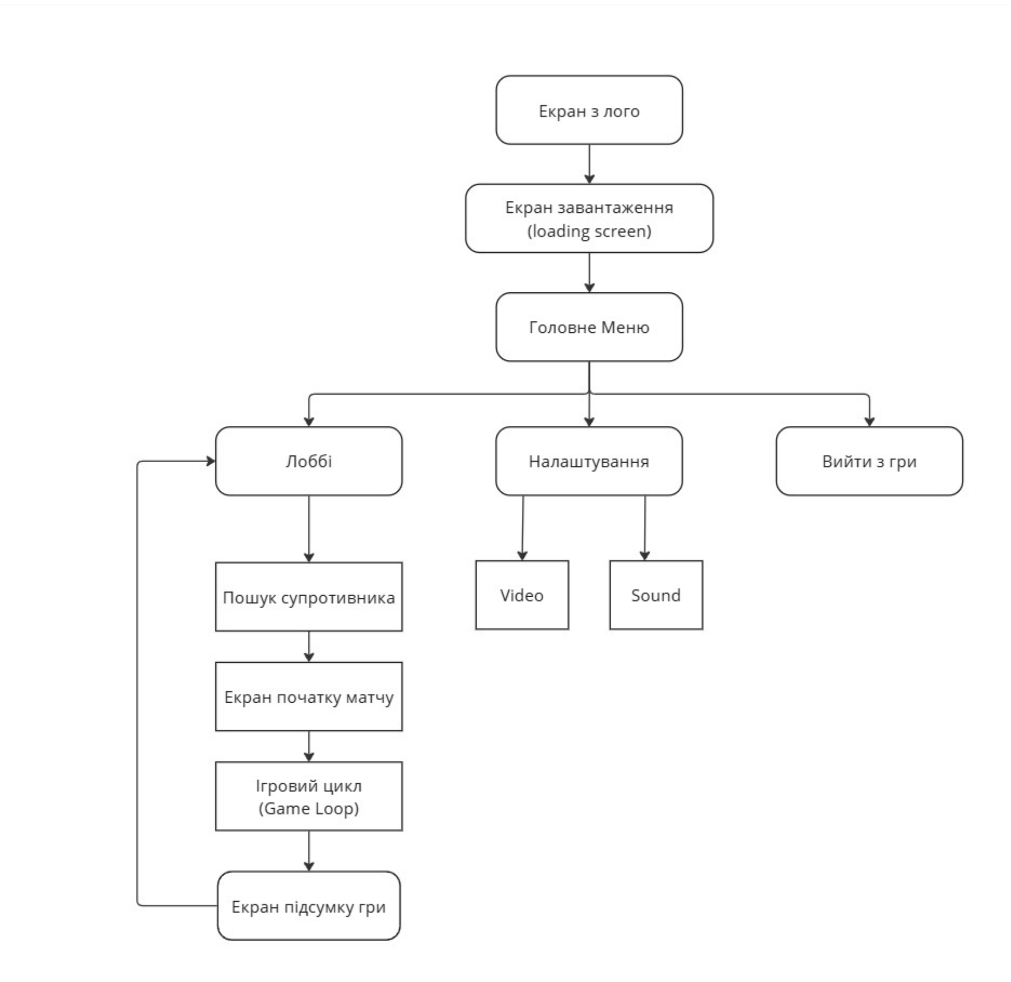


Рисунок 2.3 – Діаграма взаємодії з системою

Основними екранами у грі є:

– екран з логотипом;

– екран завантаження (loading screen);

– головне меню;

– екран початку матчу;

– ігровий цикл (Game Loop);

– екран підсумку гри.

Екран з логотипом є початковим етапом взаємодії користувача з додатком. Він відображається при запуску гри та виконує функцію презентації бренду або назви гри. Цей екран створює перше враження про додаток і забезпечує плавний перехід до наступного етапу.

Екран завантаження з’являється після логотипу та перед переходом до головного меню. На цьому етапі система завантажує необхідні ресурси для роботи програми, забезпечуючи стабільність і готовність додатка до подальшої взаємодії.

Головне меню є центральним вузлом навігації, де користувач може обрати один із трьох варіантів: перейти до лоббі, змінити налаштування або завершити роботу з додатком. Воно забезпечує доступ до основних функцій гри та є ключовим елементом інтерфейсу.

Екран початку матчу відображає основну інформацію про гру перед її початком, наприклад, імена гравців та налаштування. Цей екран допомагає користувачеві підготуватися до гри та перевірити параметри перед стартом.

Ігровий цикл є основним етапом гри, де відбувається сам процес гри в шахи. Тут реалізуються всі функціональні можливості, пов’язані з правилами шахів, ходами фігур, синхронізацією стану гри та іншими механіками.

Екран підсумку гри завершує матч, відображаючи результати (перемога, поразка або нічия). З цього екрану користувач може переглянути історію ходів або повернутися до головного меню. Він забезпечує завершення ігрового процесу та підготовку до наступного матчу.

2.5 Архітектура системи

Для створення якісного програмного забезпечення важливо розробити чітку архітектуру, яка забезпечує модульність, масштабованість та зручність супроводу проекту. Архітектура системи гри побудована на основі взаємозв’язків між

класами, які реалізують основну логіку гри, управління даними та взаємодію з користувачем. На рисунку 2.4 представлено схему частину архітектури проекту, яка демонструє структуру класів та їх взаємодію.

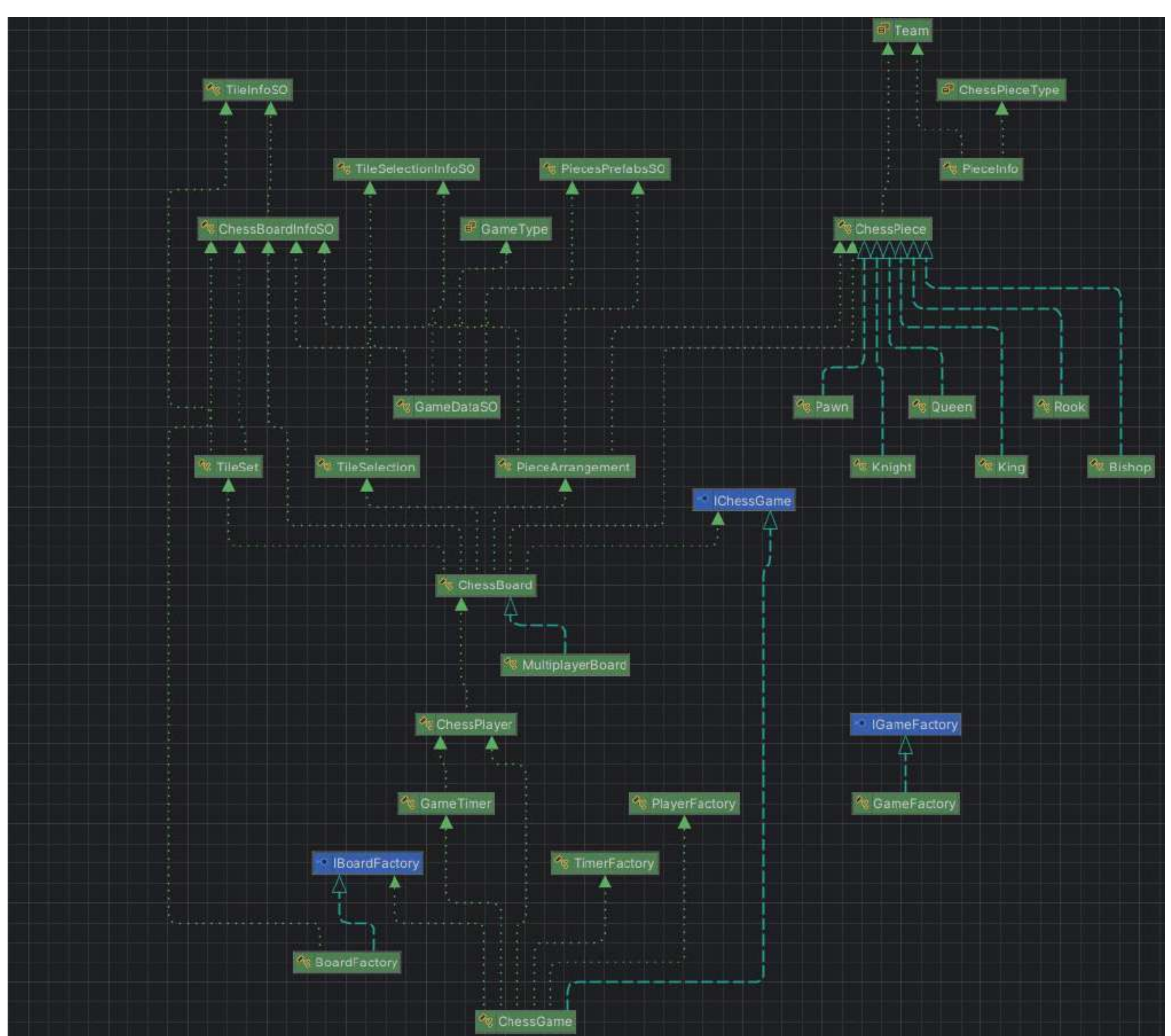


Рисунок 2.4 – Діаграма структури класів

Архітектура гри побудована навколо взаємодії кількох ключових компонентів, кожен з яких виконує свою спеціалізовану роль. Центральним елементом є клас ChessPiece, який представляє шахову фігуру. Він містить інформацію про її тип (пішак, тура, ферзь тощо), команду (білі чи чорні) та поточний стан. Від цього базового класу наслідуються конкретні типи фігур — Pawn, Queen, Rook, Knight, King та Bishop — кожен із яких має свою унікальну

поведінку.

Шахова дошка реалізована у вигляді класу ChessBoard, який відповідає за розміщення фігур та стан клітинок. Для цього він взаємодіє з об'єктами типу TileInfoSO та TileSelectionInfoSO, які містять дані про клітинки та користувацький вибір. У мультиплеєрному режимі використовується розширена версія дошки — клас MultiplayerBoard, що забезпечує синхронізацію стану гри між кількома гравцями.

Конфігураційні параметри та дані про поточну гру зберігаються у GameDataSO. Цей клас визначає тип гри (локальна або мультиплеєрна), початкове розташування фігур (PieceArrangement) та інші налаштування, що використовуються різними частинами системи.

Координацію всіх компонентів здійснює головний клас ChessGame, який керує логікою гри. Він взаємодіє з ChessBoard, ChessPlayer, GameTimer та іншими системами. Крім того, ChessGame реалізує інтерфейс IChessGame, що забезпечує легкість у розширенні функціоналу або зміні реалізації.

Створення об'єктів гри делеговано фабрикам — GameFactory, BoardFactory, PlayerFactory та TimerFactory. Вони відповідають за ініціалізацію ключових елементів, що дозволяє забезпечити модульність, повторне використання коду та гнучкість у масштабуванні проєкту. Зокрема, використання інтерфейсів, таких як IBoardFactory, дозволяє замінювати реалізації без зміни клієнтського коду.

Уся система спроєктована таким чином, щоб кожен компонент був ізольованим, легко замінним і незалежним, зберігаючи при цьому цілісність логіки шахової гри.

2.5 Аналіз архітектур UI

UI це особлива частина гри і вона може відігравати різну роль, може навіть бути частиною геймлея. Через свою унікальність для UI потрібен свій підхід.

Нажаль, в Unity не має якогось єдиного стандартного способу створення UI та його архітектури. Існує багато архітектурних патернів для UI, це MVC, MVVM, MVI, MVP тощо. Ці патерни безумовно мають своє використання, але використання їх в Unity складне, так як ці патерни створювались для Web, в Unity з її EC архітектурою реалізувати їх складно, тому що не завжди можливо виділити всі потрібні патерну класи, а використання цих патернів для геймплею взагалі сурова помилка.

2.5.1 Архітектурний патерн MVVM

Найпопулярнішим патерном для UI на Unity з усіх веб патернів є MVVM. Тому саме він розглядався як варіант архітектури UI.

Model-View-ViewModel — шаблон проєктування, що застосовується під час проєктування архітектури застосунків. Публічно вперше був представлений Джоном Госсманом у 2005 році як модифікація шаблону Presentation Model. MVVM орієнтований на такі сучасні платформи розробки, як Windows Presentation Foundation та Silverlight від компанії Microsoft [10].

MVVM полегшує відокремлення розробки графічного інтерфейсу від розробки бізнес логіки (бек-енд логіки), відомої як модель (можна також сказати, що це відокремлення представлення від моделі). Модель представлення є частиною, яка відповідає за перетворення даних для їх подальшої підтримки і використання. З цієї точки зору, модель представлення більше схожа на модель, ніж на представлення і оброблює більшість, якщо не всю, логіку відображення даних. Модель представлення може також реалізовувати патерн медіатор, організовуючи доступ до бек-енд логіки навколо множини правил використання, які підтримуються представленням [10].

Існує декілька фреймворків з реалізацією цього патерну для Unity, наприклад NoesisGUI.

2.5.2 Шаблон проектування «Посередник»

Визначає об'єкт, що інкапсулює спосіб взаємодії множини об'єктів. Посередник забезпечує слабку зв'язаність системи, звільняючи об'єкти від необхідності явно посилатися один на одного, і дозволяючи тим самим незалежно змінювати взаємодії між ними.

Переваги:

– вказує логіку посередництва між колегами. З цієї причини це легше зрозуміти Медіатор цю логіку, оскільки вона зберігається лише в одному класі;

– класи колег повністю відокремлені. Додавання нового класу колеги дуже легко через цей рівень роз'єднання;

– об'єкти колеги повинні спілкуватися лише з об'єктами посередника. Практично модель медіатора зменшує необхідні канали зв'язку (протоколи) від багатьох до багатьох до одного до багатьох і багатьох до одного;

– оскільки вся логіка зв'язку вказується класом посередника, коли ця логіка потребує розширення, лише клас посередника потрібно розширити.

2.5.3 Вибір архітектури UI

Проаналізувавши всі варіанти, було визначено що застосування додаткового фреймворка для даного проекту тільки ускладнить код і не дасть позитивних змін у аспектах швидкості написання коду, подальшої підтримки проекту тощо. Отже, було вирішено дотримуватися одного з зазначених вище принципів програмування KISS і не ускладнювати проект без потреби. Тому для UI коду використовувався шаблон проектування «Посередник».

2.6 Принцип інверсії залежностей, використання DI-контейнера Zenject

Важливо було продумати як будуть впроваджуватися залежності, як клас буде отримувати доступ к сервісам, необхідним компонентам тощо.

Оскільки було вирішено дотримуватися принципів SOLID, де принцип D це dependency inversion, згідно з яким модулі високого рівня не повинні залежати від модулів низького рівня, було вирішено використовувати шаблон проектування dependency injection(DI).

Впровадження залежності (англ. Dependency injection, DI) — шаблон проєктування програмного забезпечення, що передбачає надання зовнішньої

залежності програмному компоненту, використовуючи «інверсію управління» для розв'язання (отримання) залежностей [11].

Впровадження — це передача залежності (тобто, сервісу) залежному об'єкту (тобто, клієнту). Передавати залежності клієнту замість дозволити клієнту створити сервіс є фундаментальною вимогою до цього шаблону проєктування [11].

Впровадження залежностей необхідно складова, особливо для модульного тестування і має багато переваг:

– оскільки впровадження залежностей не вимагає змін у поведінці коду, його можна застосувати як рефакторинг. В результаті цього клієнти стають більш незалежними і над ними легше проводити модульне тестування в ізоляції з використанням макетів об'єкта, які імітують інші об'єкти, від яких залежить об'єкт, що тестується. Простота тестування найчастіше є першою помітною перевагою використання впровадження залежностей [11];

– впровадження залежностей не вимагає від клієнта знань про конкретну реалізацію, яку йому потрібно використовувати. Це дозволяє ізолювати клієнт від впливу змін проєктування і дефектів. Це сприяє повторному використанню, тестуванню і підтримці коду [11];

– впровадження залежностей може використовуватися для перенесення деталей конфігурації системи в конфігураційні файли, що дозволяє системі змінювати конфігурацію без перекомпіляції. Окремі конфігурації можуть бути написані для різних ситуацій, що вимагають різних реалізацій компонентів [11];

– впровадження залежностей сприяє паралельній і незалежній розробці. Два розробника можуть незалежно створювати класи, які використовують один одного, знаючи тільки про інтерфейси, через які класи співпрацюють [11];

– впровадження залежностей знижує зв'язність між класом і його залежностями [11].

При розробці проекту стояло два вибори вирішення питання залежностей DI або Singleton.

Одинак (англ. Singleton) — шаблон проєктування, відноситься до класу твірних шаблонів. Гарантує, що клас матиме тільки один екземпляр, і забезпечує

глобальну точку доступу до цього екземпляра. Для деяких класів важливо, щоб існував тільки один екземпляр. Наприклад, хоча у системі може існувати декілька принтерів, може бути тільки один спулер. Повинна бути тільки одна файлова система та тільки один активний віконний менеджер. Глобальна змінна не вирішує такої проблеми, бо не забороняє створити інші екземпляри класу. Рішення полягає в тому, щоб сам клас контролював свою «унікальність», забороняючи створення нових екземплярів, та сам забезпечував єдину точку доступу [12].

Не зважаючи на переваги швидкого доступу до всіх елементів цей варіант має великі недоліки:

– по мірі розростання проекту, код стає все більш і більш не контрольованим та дуже складно читаємим, тому що звернення до залежностей не контрольоване і може відбуватися де завгодно. Такий код може міняти все що заманеться і де заманеться;

– при використанні сінглтона модульне тестування стає практично не можливим.

Через суттєві недоліки сінглтона, та через те що він не відповідає принципу dependency inversion було обрано впровадження залежностей.

Впровадження залежностей майже не можливе без фреймворку, а не обачність може привести до перетворення впровадження залежностей у анти-патерн сервіс-локатор. Тому для використання впровадження залежностей було вирішено використати фреймворк для Unity під назвою Zenject.

2.7 Налаштування елементів тестування гри

Існує безліч різних видів тестування, які можна використовувати для забезпечення того, щоб зміни, що вносяться в код, працювали відповідно до очікувань.

Насамперед потрібно розрізняти автоматичне тестування та тести, що

виконуються вручну. Тестування в ручному режимі проводить людина, яка перевіряє роботу всього функціоналу програми вручну або шляхом взаємодії з програмним забезпеченням та API з використанням відповідного інструментарію. Цей спосіб є дуже витратним, оскільки вимагає налаштування середовища та виконання тестів спеціалістом. Крім того, необхідно враховувати людський фактор, тому що тестувальник може допустити друкарську помилку або пропустити якийсь етап тестового скрипту.

Автоматичні тести, навпаки, виконуються машиною, яка використовує заздалегідь написаний скрипт тесту. Такі тести можуть значно відрізнятися за складністю: від перевірки одного методу в класі до забезпечення того, щоб виконання послідовності складних дій в інтерфейсі користувача призводило до однакових результатів. Такий підхід набагато стабільніший і надійніший у порівнянні з тестами, що виконуються вручну. При цьому якість автоматичного тестування залежатиме від якості тестових скриптів.

Види тестування:

– модульні тести;

– інтеграційні тести;

– функціональні тести;

– наскрізні тести.

2.7.1 Налаштування Unit тестів

Для тестування гри було обрано використання юніт-тестів. Основними бібліотеками, що застосовувались у процесі тестування, стали NSubstitute, NUnit та Fluent Assertions. Використання цих бібліотек дозволило створювати чисті, ізольовані тести з можливістю легкої заміни залежностей та чітким формулюванням очікувань. Такий підхід сприяв підвищенню якості коду та забезпечував швидкий зворотній зв’язок під час розробки.

NSubstitute використовується як інструмент для створення підставних об'єктів(моків) у .NET. Основна його перевага полягає у простому й лаконічному синтаксисі, який дозволяє зосередитися безпосередньо на логіці тестів, а не на складній конфігурації тестових подвійників. Це особливо корисно для тих, хто

лише починає працювати з тестуванням, або бажає писати тести швидко та з мінімальною кількістю зайвого коду.

NUnit є платформою з відкритим кодом для написання модульних тестів у середовищі .NET. Вона дозволяє виконувати тести як з консолі, так і безпосередньо у Visual Studio або через сторонні інструменти. NUnit підтримує паралельне виконання тестів, категоризацію для вибіркового запуску, а також зручно працює з параметризованими тестами. Крім того, ця платформа є сумісною з різними .NET-платформами, включаючи .NET Core і Xamarin. NUnit активно підтримується спільнотою розробників і регулярно оновлюється для врахування нових технологічних тенденцій. Для пакетного запуску тестів можна використовувати консольну утиліту nunit3-console, яка через NUnit Test Engine забезпечує завантаження, дослідження й виконання тестів. Якщо виникає потреба в ізольованому середовищі для виконання, використовується окремий агент nunit-agent.

Fluent Assertions — це бібліотека для формулювання очікувань у тестах у зручній і природній формі. Вона добре підходить для стилів розробки TDD або BDD, дозволяючи писати перевірки, які легко читати та підтримувати. Через те, що Unity не підтримує повноцінно систему пакетів NuGet і постійно перегенеровує проєктні файли .csproj, бібліотеку довелося додавати вручну разом із усіма залежностями у вигляді .dll-файлів. Це іноді ускладнювало оновлення бібліотеки, проте дозволяло зберегти стабільність проєкту, також у подальшому цей процес можливо автоматизувати за допомогою білд процесу або використати пакет NuGetForUnity для оновлення бібліотек, щоб уникнути помилок. Водночас приємною несподіванкою стала можливість запускати тести безпосередньо з середовища JetBrains Rider, що значно полегшило процес перевірки й налагодження. Завдяки цьому тестування стало більш інтегрованим у робочий процес і сприяло швидшому виявленню помилок.

Таким чином, використання NSubstitute, NUnit та Fluent Assertions забезпечує ефективний, гнучкий та зручний підхід до автоматизованого тестування, що значно підвищує якість та надійність розроблюваного програмного забезпечення.

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Створення вертикального зрізу

Створення ігор відбувається ітеративно, тобто з часом багато чого змінюється, дороблюється часто буває таке, що те над чим працювали декілька місяців просто викидується через якісь причини, наприклад якщо виявилось, що механіка над якою працювали рушить занурення гравця в сюжет або просто вона нудна. Через таку природу ігор, важливо зрозуміти що ітерування це невідємний процес створення ігор і що переробляти іноді приходиться всю гру. Спочатку було вирішено створити вертикальний зріз з основними механіками, тобто шахи в які можна грати онлайн, це було основною ідеєю першої версії гри.

Вертикальний зріз – мінімально можлива повноцінна версія гри, що включає повністю реалізований основний ігровий процес. При цьому всі базові фічі гри присутні як мінімум у чорновій якості.

Першу версію гри зображено на рисунку 3.1.

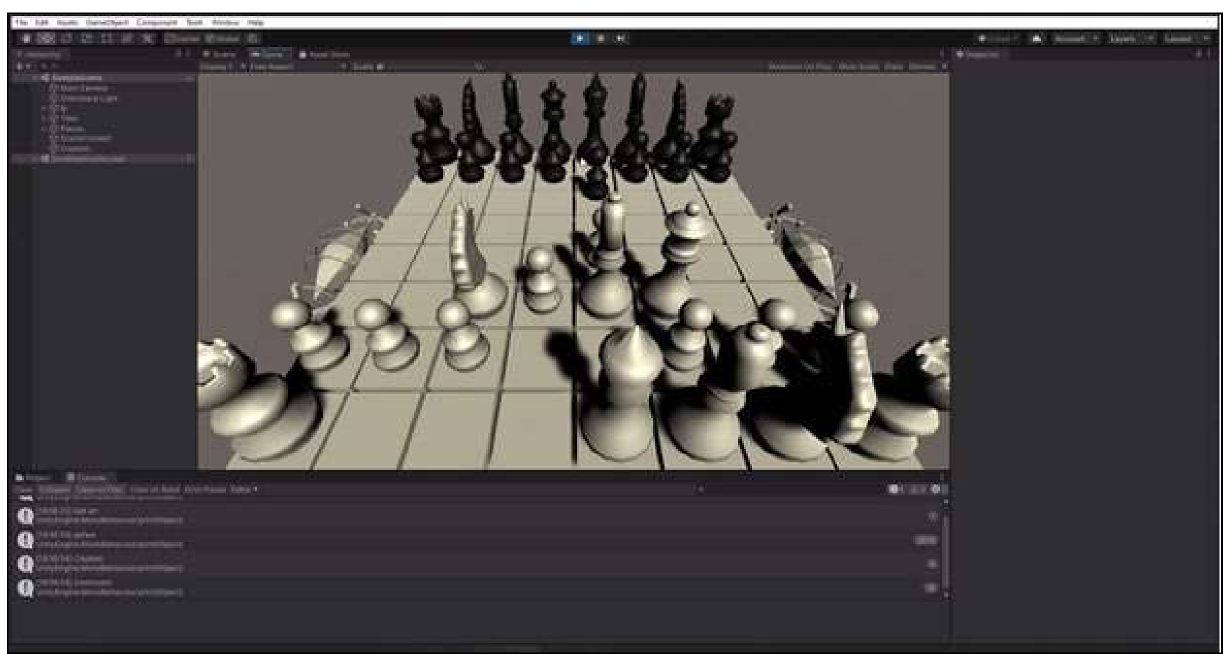


Рисунок 3.1 – Тестова версія гри

3.2 Загальний огляд архітектури

Архітектура гри була створена за всіма стандартами якості та з використанням провідних методів організування проекту. При розробці коду були застосовані принципи SOLID, DRY, YAGNI, KISS. Це допомогло створити якісну, зручну у використанні архітектеру яку можливу швидко розширювати та змінювати. Стандарти допомогли зробити код більш читаємим, легким у розумінні.

3.2.1 Налаштування інверсії залежностей

У кожній сцені є скрипт Installer який успадкувається від ZenjectInstaller, і вирішує залежності до початку роботи сцени у методі InstallBindings, це робиться за допомогою вікна «Script Execution Order» зображеного на рисунку 3.2.

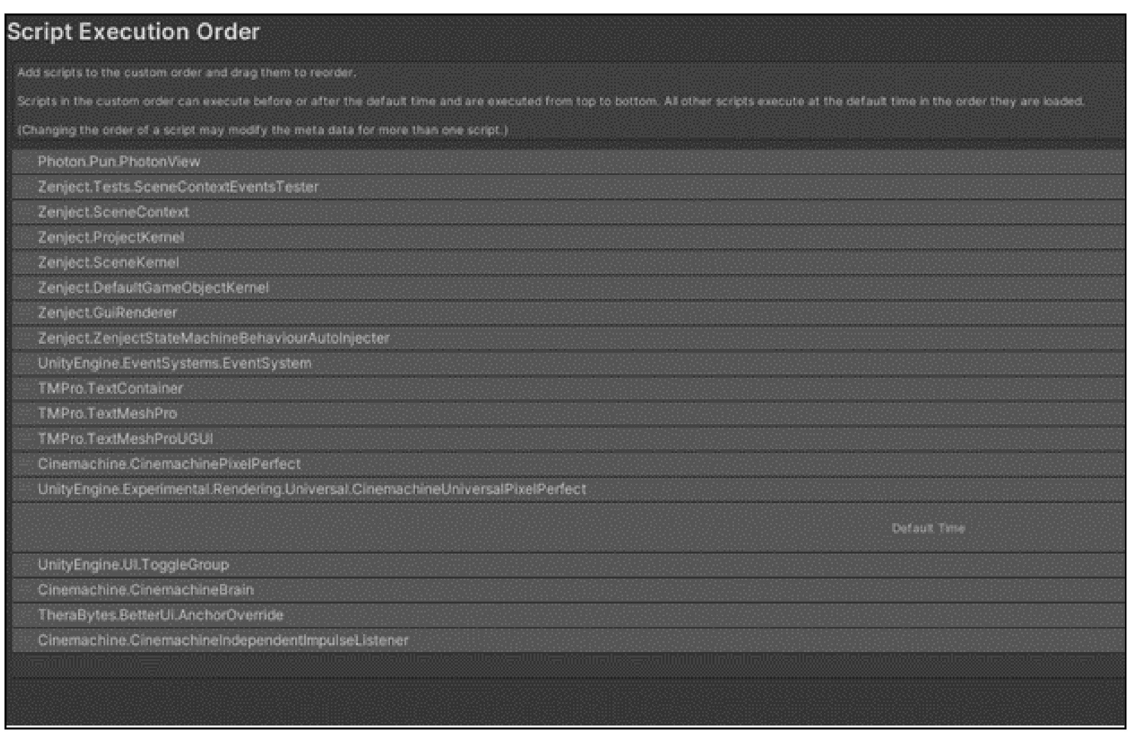


Рисунок 3.2 – Вікно «script execution order» з встановленним порядком виконання скриптів

Також у цьому вікні є скрипти Photon.Pun це клас який відповідає за мережеву составляючу, тому йому теж потрібно ініцілізуватися раніше за всіх інших. Отже, саме в інсталлерах ми вирішуємо залежності сцін, також існує

ProjectContext який відповідає за залежності які доступні вродовж всього часу гри. Приклад інсталеру гри наведено у лістингу 3.1.

Лістинг 3.1 – Метод інсталеру з реєстрацією залежностей

```
public    override                void                InstallBindings()
{
    BindChessBoardData();
    BindGameCameraController();
    BindTimer();
    BindMoveListingData();

    Container
        .Bind<IGameFactory>()
        .To<GameFactory>()
        .AsSingle();

    Container
        .Bind<IBoardFactory>()
        .To<BoardFactory>()
        .AsSingle();

    Container
        .Bind<ChessPieceFactory>()
        .AsSingle();

    Container
        .Bind<PlayerFactory>()
        .AsSingle();

    Container
        .Bind<ISpecialMoveFactory>()
        .To<SpecialMoveFactory>()
        .AsSingle();
}
```

Процес передачі залежностей у клас наведено у лістингу 3.2, де у метод помічений атрибутом Inject передаються залежності після конструювання сцени.

Лістинг 3.2 –Метод інсталеру з реєстрацією залежностей

```
public class WinOrLosePopUp : PopUp
{
     [Header("WinOrLosePopUp")]
     [SerializeField] private TextMeshProUGUI _winOrLoseText;
     [SerializeField] private TextMeshProUGUI _matchResultText;
     [SerializeField] private TextMeshProUGUI[] _playersTexts;
     [SerializeField] private TextMeshProUGUI _scoreText;
```

```
private INetworkService _networkService;
private IAudioSystem _audioSystem;

public override string PopUpKey { get; set; } =
GameConstants.PopUps.WinOrLoseWindow;

 [Inject]
private void Construct(INetworkService networkService, IAudioSystem
audioSystem)
{
	_audioSystem = audioSystem;
	_networkService = networkService;
}
```

Одним із ключових елементів роботи Zenject є ProjectContext. Це спеціальний об'єкт, який автоматично створюється при запуску проєкту та слугує початковою точкою для налаштування залежностей на глобальному рівні. Саме ProjectContext відповідає за пошук і виконання інсталерів — класів, у яких реєструються всі необхідні залежності гри.

На рисунку 3.3 зображено приклад структури ProjectContext-а, який виконує роль "контейнера", а інсталери (Installer) описують логіку реєстрації компонентів.

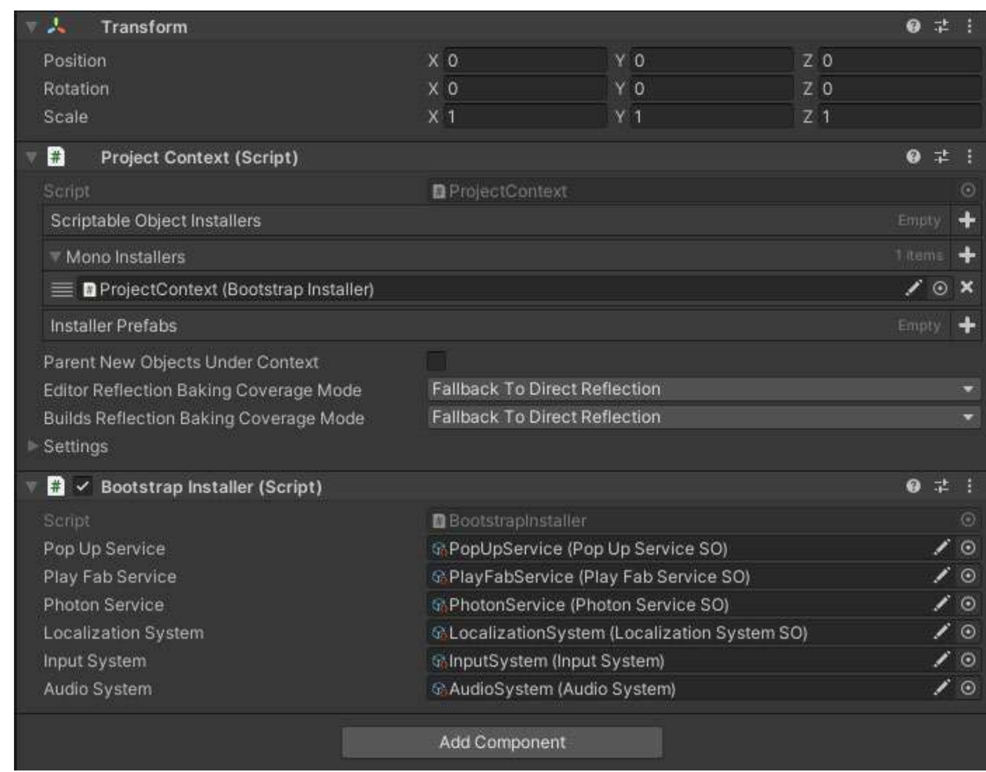


Рисунок 3.3 – Об’єкт впровадження залежностей

Насправді передача залежностей працює завдяки рефлексії, знаходяться всі методи помічені цим атрибутом маркером і в них передаються залежності.

Рефлексія - процес, під час якого програма може відслідковувати та модифікувати власну структуру та поведінку під час виконання. Парадигма програмування, покладена основою відображення, є однією з форм метапрограмуваня і називається рефлексивним програмуванням.

Під час виконання програмних інструкцій (коду) комп'ютери обробляють дані, що призводить до зміни, при цьому комп'ютери не змінюють код. Однак, у більшості сучасних комп'ютерних архітектур код зберігається як дані, і в деяких мовах програмування реалізована можливість обробляти власний код як дані, що призводить до зміни самого коду під час його виконання. Такі самостійні програми в основному створюються за допомогою високорівневих мов програмування, що використовують віртуальні машини.

3.2.2 Менеджмент строк у грі

Строки яки використовуються для загрузки чогось, як імя це завжди слабке місце, тому що у IDE не має можливості перевірки їх правильності. Якщо це метод, IDE шукає цю назву в метадних, якщо не находить тоді видає помилку, але якщо це строка, то такого зробити не можливо. Тому важливо такі строки виділяти в константи, для цього в проекту було створено partial клас GameConstants, тобто клас визначення якого можно розділити на декілька ісходних файлів. Приклад одного такого ісходного файлу наведено у лістингу 3.3.

Лістинг 3.3 – partial клас з константами

```
public static partial class GameConstants
 {
     public static class PopUps
     {
         public const string LogInWindow = "WindowLogIn";
         public const string SignUpWindow = "WindowSignUp";
         public const string ForgotPasswordWindow = 
"WindowForgotPassword";
         public const string RoomPasswordWindow = 
"WindowRoomPassword";
```

```
            public const string ErrorToast = "ErrorToast";
            public const string SuccessToast = "SuccessToast";
            public const string InfoToast = "InfoToast";
            public const string RoomCreationWindow = 
"WindowRoomCreation";
            public const string PlayersOverviewWindow = 
"PlayersOverviewWindow";
            public const string WinOrLoseWindow = "WindowWinOrLoss";
            public const string PromotionWindow = 
"WindowPawnPromotion";
            public const string LoadingPopUp = "LoadingPopUp";
            public const string SettingsPopUp = "SettingsPopUp";
            public const string ExitGamePopUp = "WindowExitTheGame";
        }
}
```

Константи у середовищі .NET поміщаються у метаданні сборки,а при використанні працює механізм інтернування.

У інформатиці інтернування рядків — це метод збереження лише однієї копії кожного окремого рядкового значення, яке має бути незмінним . Інтернування рядків робить деякі завдання обробки рядків більш ефективними в часі або просторі за ціною потреби більше часу під час створення або інтернування рядка. Різні значення зберігаються в інтерн-пулі рядків .

Інтернування рядків прискорює порівняння рядків, які іноді є вузьким місцем у додатках (таких як компілятори та середовища виконання динамічної мови програмування), які значною мірою покладаються на асоціативні масиви з ключами рядків для пошуку атрибутів і методів об’єкта. Без інтернування порівняння двох різних рядків може включати в себе вивчення кожного символу обох. Це повільно з кількох причин: це за своєю суттю O(n) у довжині рядків; зазвичай вимагає зчитування з кількох областей пам'яті , що вимагає часу; а зчитування заповнюють кеш процесора, що означає, що для інших потреб залишається менше кешу. З інтернованими рядками, простийтест на ідентичність об'єкта достатній після початкової операції стажера; це зазвичай реалізується як тест на рівність покажчика, як правило, лише одна машинна інструкція без посилання на пам'ять взагалі.

3.2.3 Локальне зберегання даних

У грі завжди необхідно зберігати якісь дані між сесіями, це можна зробити записуючи їх у файл у Json або XML, для цього проекту локальні дані які потрібно було зберігати, це загалом налаштування тому було вирішено зберігати ці дані за допомогою встроєного в Unity типу PlayerPrefs. PlayerPrefs в залежності від системи зберігає дані в різних місця, наприклад в Windows він зберігає їх у регістрі, в MacOS у папці ~/Library/Preferences folder. Скрипт роботи з PlayerPrefs наведено у лістингу 3.4.

Лістинг 3.4 –Класс для збереження налаштуваннь гри

```
public static class Settings
{
     private const string POST_PROCESSING = "PostProcessing";
     private const string WINDOW_SIZE = "WindowSize";
     private const string TEXTURE_QUALITY = "TextureQuality";
     private const string MUSIC = "Music";
     private const string SOUNDS = "Sounds";
     private const string LANGUAGE = "Language";

     public static bool PostProcessing
     {
         get { return PlayerPrefs.GetInt(POST_PROCESSING, 1) == 1; }
         set { PlayerPrefs.SetInt(POST_PROCESSING,
     Convert.ToInt32(value)); }
     }

     public static int TextureQuality
     {
         get { return PlayerPrefs.GetInt(TEXTURE_QUALITY, 0);  }
         set { PlayerPrefs.SetInt(TEXTURE_QUALITY,
     Convert.ToInt32(value)); }
     }

     public static bool Music
     {
         get { return PlayerPrefs.GetInt(MUSIC, 1) == 1; }
         set { PlayerPrefs.SetInt(MUSIC, Convert.ToInt32(value)); }
     }

     public static bool Sounds
     {
         get { return PlayerPrefs.GetInt(SOUNDS, 1) == 1; }
         set { PlayerPrefs.SetInt(SOUNDS, Convert.ToInt32(value)); }
     }
```

```
        public static string Language
        {
            get { return PlayerPrefs.GetString(LANGUAGE, "en"); }
            set { PlayerPrefs.SetString(LANGUAGE, value); }
        }
    }
```

На рисунку нижче зображено вікно налаштувань гри, яке дозволяє гравцеві змінювати параметри. Зміни, які вносить користувач, одразу зберігаються у PlayerPrefs після закриття та повторного запуску гри.

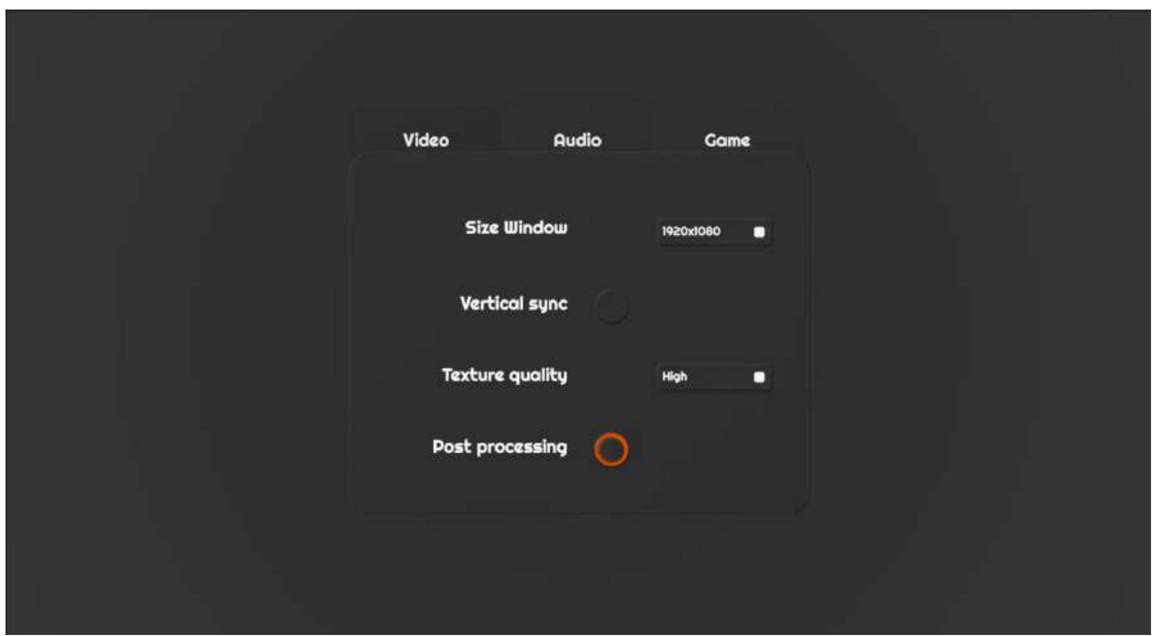


Рисунок 3.4 – Вікно налаштуваннь

3.2.4 Керування пам'яттю

За дефолтом Unity має дуже обмежене керування пам'яттю, усі ресурси на яких є посилання на сцені негайно загружаються в пам'ять при загрузці сцени. Це може призвести до проблем з пам'ятю, тому для ігор яким потрібне більш гнучке керування пам'ятю, є створена юніті система Addressables. Це дуже потужна система, яка також дозволяє розбивати ассети на групи, які потім можуть оновлюватися з інтернету без необхідності оновлювати або скачувати якесь доповнення. Ця система використовувалась у грі для загрузки деяких сцен, а також UI елементів. Також ця система використовується для загрузки локалізації.

Розбиті групи ассетів показано на рисунку 3.5.

Рисунок 3.5 – Групи ассетів у вікні «Addressables Groups»

3.2.5 User Interface

Для реалізації UI використовувався шаблон проектування «Посередник». Усі посилання у сцені були зроблені через окремий клас-медіатор у якому були зібрані всі необхідні методи, головне у цьому підході було не робити у ньому якоїсь логіки, бо це клас лише посередник між самою логікою та елементом на сцені. Приклад класу медіатора наведено у лістингу 3.5.

Лістинг 3.5 –Метод інсталеру з реєстрацією залежностей

```
public class MainMenuMediator : MonoBehaviour
{
     [SerializeField] private MainMenu _mainMenu;

     [DisableInEditorMode, Button] public void OnClick_Play() =>
     _mainMenu.SwitchToLobby();

     [DisableInEditorMode, Button] public void OnClick_Settings() =>
     _mainMenu.SwitchToSettings();

     [DisableInEditorMode, Button] public void OnClick_Exit() =>
     Application.Quit();
}
```

Також методи цього класу помічені атрибутом Button, це дозволяє визивати ці методи через інспектор на цьому об'єкті. Атрибут Button - це атрибут відомого ассета Odin Inspector.

Odin Inspector — це плагін для Unity, який дає змогу насолоджуватися всіма перевагами робочого процесу, пов'язаними з наявністю потужного, налаштованого та зручного редактора, без необхідності писати жодного рядка користувацького коду редактора. Замість того, щоб писати й підтримувати тисячі рядків коду користувацького редактора, розробники можуть анотувати свої структури даних за допомогою 100+ атрибутів будівельних блоків, щоб швидко й легко створювати зручні редактори для всієї своєї команди.

Odin також наповнений широким набором утиліт редактора для багатьох поширених завдань, таких як створення користувацьких вікон редактора та багато, багато іншого.

Вигляд ієрархії UI та вікно реєстрації наведено на рисунках 3.6 та 3.7.

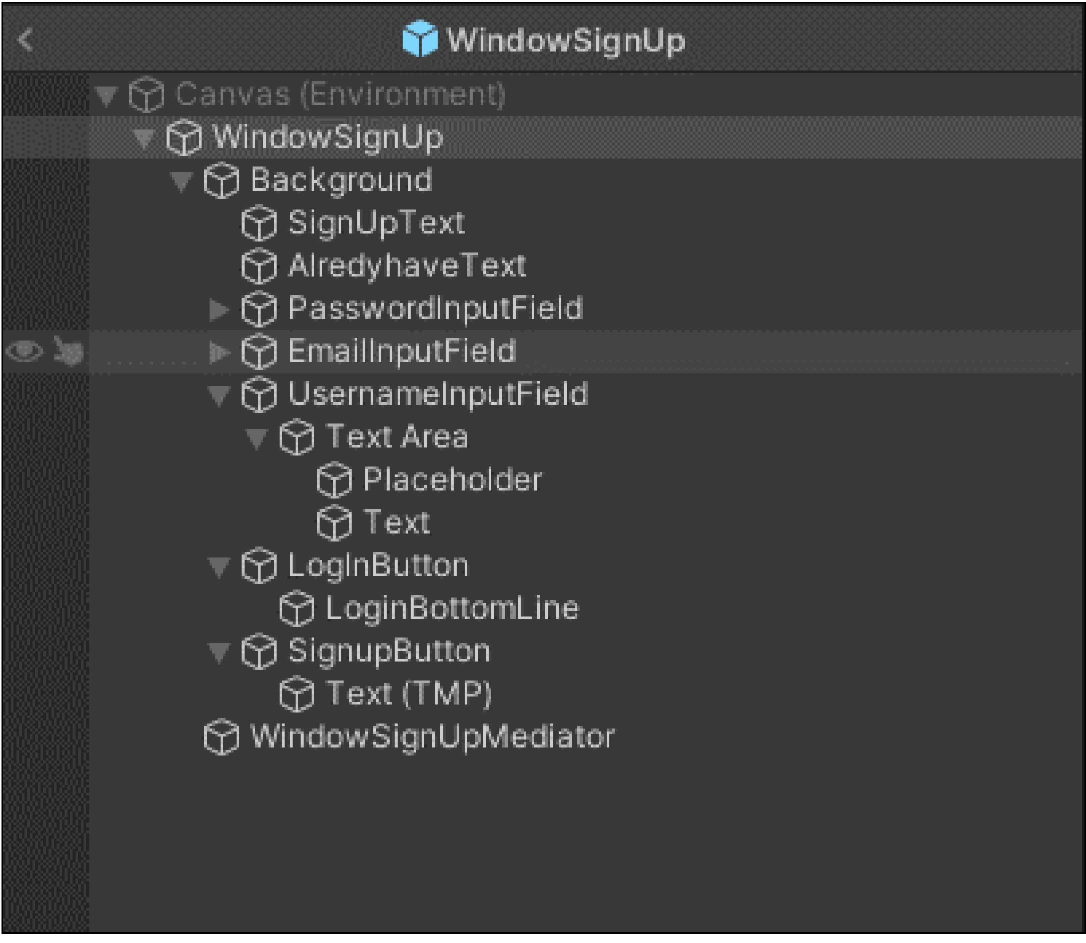


Рисунок 3.6 – Ієрархія UI елементу гри

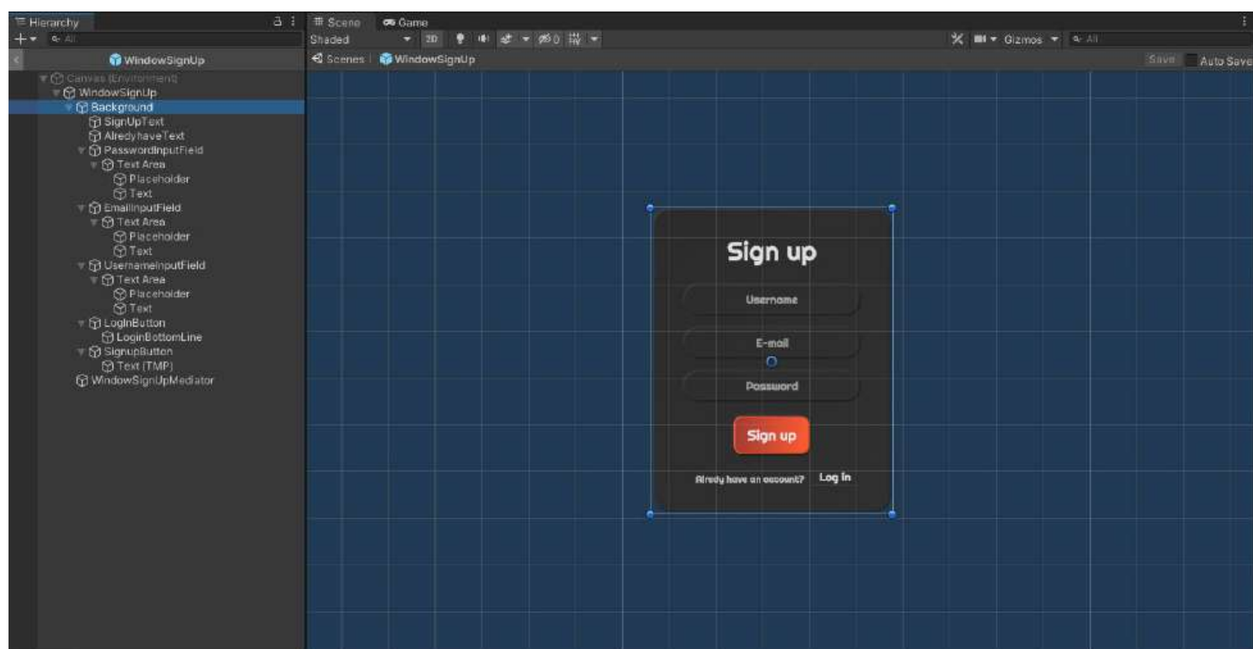


Рисунок 3.7 – Вікно реєстрації

3.2.6 Опис ініціалізації сцени гри

Після запуску сцени, у GameEntryPoint створюється екземпляр класу ChessGame. Оскільки при різних обставинах у майбутньому, можуть створюватися різні види гри, а також через те що при створенні потрібно впроваджувати залежності, логіку створення було абстраговано і використано шаблон проектування «Фабричний метод». Клас GameFactory наведено у лістингу 3.6.

Лістинг 3.6 – Клас «GameFactory» з фабрични методом для створення гри

```
public class GameFactory : IGameFactory
{
 private readonly IInstantiator _instantiator;

 public ChessGame CachedGame { get; private set; }

 public GameFactory(IInstantiator instantiator)
 {
     _instantiator = instantiator;
 }

 public ChessGame Create()
 {
     if (CachedGame == null)
     {
         ChessGame chessGame = _instantiator.Instantiate<ChessGame>();
             CachedGame = chessGame;
```

```
        }

        return CachedGame;
    }
}
}
```

Фабричний метод - визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який саме клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласам.

Слід використовувати шаблон «Фабричний метод» коли:

– класу не відомо заздалегідь, об'єкти яких саме класів йому потрібно створювати;

– клас спроєктовано так, щоб об'єкти, котрі він створює, специфікувалися підкласами;

– клас делегує свої обов'язки одному з кількох допоміжних підкласів, та потрібно локалізувати знання про те, який саме підклас приймає ці обов'язки на себе.

Після, створення екземпляру відбувається його ініціалізація, де створюється потрібний клас ChessBoard, його створення теж абстраговано в окремий клас. Окрім того, в цьому методі створюються гравці, ініціалізується дошка, визначається позиція фігур на дошці, створюється та ініціалізується таймер гри. Загалом тут відбувається ініціалізація подальшої ігрової сессії. Код методу наведено у лістингу 3.7.

Лістинг 3.7 – Метод «Initialize»

```
public void Initialize()
{
    InitialPieceArrangementData                                    =
    _notationString.GameDataFromNotationString();

    ChessBoard chessBoard = _boardFactory.Create();

    CreatePlayers(chessBoard);
    chessBoard.Initialize(this);
    _timer = _timerFactory.Create();
    _timer.InitializeTimer(ChessPlayers[0],          ChessPlayers[1],
```

```
        EndOfGame);
        WhoseTurn = (Team) InitialPieceArrangementData.whoseTurn;
        ActivePlayer = ChessPlayers[(int) WhoseTurn];
        _localPlayer = GetLocalPlayer();
        _gameCameraController.Initialize(_localPlayer.Team);
        SubscribeToLeftGameEvents();
        _timer.Start();
}
```

Клас ChessBoard це абстрактний клас, який містить у собі загальну логіку дошки, а вже MultiplayerBoard це клас який наслідується від нього і реалізує логіку онлайн гри. У цьому класі, заміщуються основні методи які відповідають за рух фігури і робиться RPC виклик, тобто синхронізований виклик метода на всіх клієнтах.

3.3 Реалізація шахових фігур

Кожна фігура успадковається від класу ChessPiece, який визначає всі основні методи та дані фігури. Дані класу наведено у лістингу 3.8.

Лістинг 3.8 – Дані класу «ChessPiece»

```
public abstract class ChessPiece : IDisposable
{
     public int CurrentX;
     public int CurrentY;
     public Team Team;
     public ChessPieceType ChessType { get; set; }
     public Transform PieceTransform { get; set; }
     protected IObjectTweener Tweener;
     public static event Action<ChessPiece> OnPieceDestroyed;
     public static event Action<ChessPiece> OnPieceSpawned;
}
```

CurrentX та CurrentY – це позиція фігури на шаховій дошці, Team – це команда фігури, ChessType – це тип фігури, наприклад король або пішак.

PieceTransform – це компонент фігури на сцені, оскільки фігури це не

MonoBehaviour класи їм потрібно зберігати посилання на компонент фігури у сцені для того щоб переміщувати модельку фігури. Tweener – це клас який відповідає тип переміщення фігури, наприклад у пішака воно лінійне, а у коня за аркою.

Івенти OnPieceSpawned і Destroyed потрібні для додавання та видалення фігур з листу активних фігур гравця.

Метод GetAvailableMoves відповідає за визначення можливих ходів для фігури і заміщаються окрема для кожної фігури.

Метод PositionPiece наведено у лістингу 3.9, відповідає безпосередньо за переміщення фігури, також він асинхроний, а метод твінера MoveTo повертає Task.

Лістинг 3.9 – Метод для переміщення фігури

```
public Task PositionPiece(int x, int y, bool force = false, IObjectTweener tweener = null)
{
    CurrentX = x;
    CurrentY = y;
    Vector3 tile = TileSet.GetTileCenter(x, y);
    if (force)
    {
        PieceTransform.position = tile;
        return Task.CompletedTask;
    }

return tweener == null ? Tweener.MoveTo(PieceTransform, tile) : tweener.MoveTo(PieceTransform,tile);
}
```

3.4 Реалізація спеціальних рухів та використання шаблону «Стратегія»

У шахах є декілька спеціальних рухів – це взяття на проході, перетворення, рокірування.

Ці ходи також прораховуються у методі GetAvailableMoves, який визиває метод GetSpecialMove для фігур які його мають. Оскільки ці рухи потребують додаткових дій, ніж звичайні, спочатку було вирішено зробити перелічення SpecialMove і обробляти рухи в залежності від значення перелічення SpecialMove.

Згодом було проведено рефакторинг і використано шаблон проектування «Стратегія».

Стратегія — шаблон проектування, належить до класу шаблонів поведінки. Його суть полягає у тому, щоб створити декілька схем поведінки для одного об'єкту та винести в окремий клас.

Переваги:

– можливість позбутися умовних операторів;

– клієнт може вибирати найбільш влучну стратегію залежно від вимог щодо швидкодії і пам'яті.

Недоліки:

– збільшення кількості об'єктів;

– клієнт має знати особливості реалізацій стратегій для вибору найбільш вдалої.

Було визначено спільний інтерфейс ISpecialMove з методом ProcessSpecialMove, а обробка кожного руху була винесена в окремий клас, клас. Клас руху взяття на проході зображено у лістингу 3.10, а діаграму класів, що ілюструє його взаємодію з іншими компонентами, — на рисунку 3.8.

Лістинг 3.10 – Інкапсульована логіка руху «Взяття на проході»

```
public class EnPassant : ISpecialMove
{
    private readonly List<Movement> _moveHistory;
    private readonly PieceArrangement _pieceArrangement;

   public EnPassant(List<Movement> moveHistory, PieceArrangement pieceArrangement)
   {
       _moveHistory = moveHistory;
       _pieceArrangement = pieceArrangement;
   }

    public Task ProcessSpecialMove()
    {
        Movement newMove = _moveHistory[_moveHistory.Count - 1];
       ChessPiece myPawn = _pieceArrangement[newMove.Destination.x, newMove.Destination.y];
       Movement targetPawnPos = _moveHistory[_moveHistory.Count - 2];
       ChessPiece targetPawn =
```

```
_pieceArrangement[targetPawnPos.Destination.x,
targetPawnPos.Destination.y];
        if (myPawn.CurrentX == targetPawn.CurrentX)
        {
            if   (myPawn.CurrentY   ==   targetPawn.CurrentY   -   1   ||
myPawn.CurrentY == targetPawn.CurrentY + 1)
            {
                _pieceArrangement[targetPawn.CurrentX,
targetPawn.CurrentY] = null;
                targetPawn.Dispose();
            }
        }
        return Task.CompletedTask;
    }
}
```

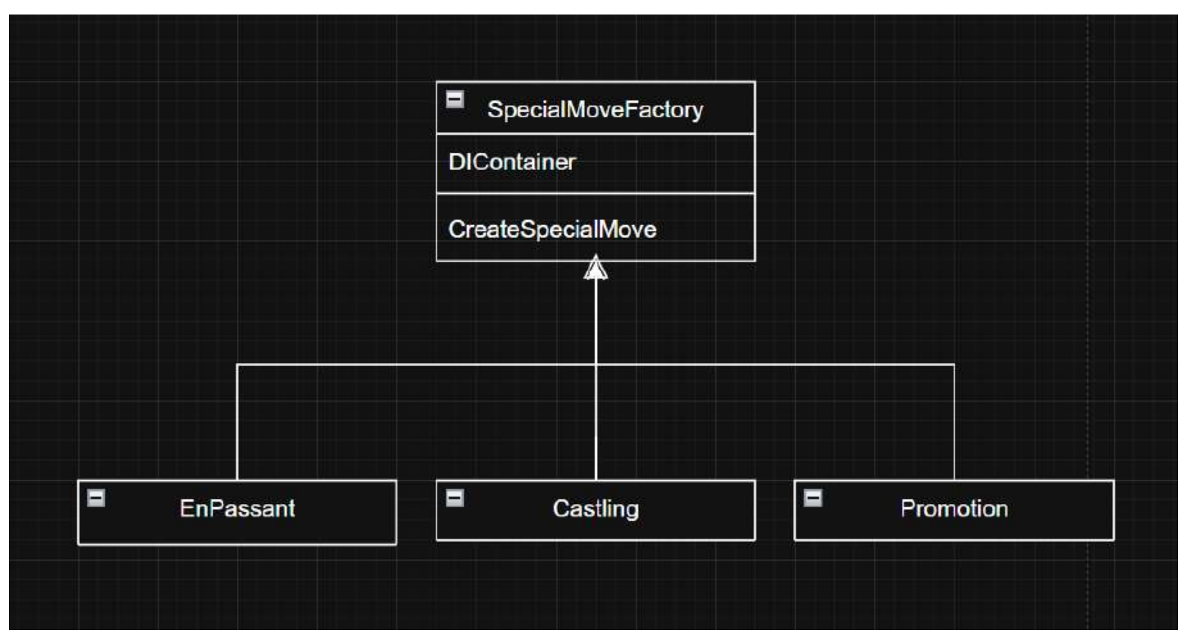


Рисунок 3.8 – Діаграма спеціальних рухів

3.5 Реалізація правил гри

Перевірка чи не стає король під бій при вчиненні хода, перевірка після ходу, чи не мат протилежній команді, ці перевірки визначені в класі ChessBoard.

Це було потрібно тому що цим методам для перевірки потрібен доступ до масиву з фігурами, історії рухів тощо. Також можливо було винести ці методи у окремий статичний клас, тим самим зробити цю частину більш функціональною.

Методи перевірки визиваються перед кожним виділенням можливих рухів і створюють окремий масив для симуляції руху, якщо після симуляції король у безпеці тоді цей рух можливий.

Розрахунок мату визивається після кожного руху і перевіряє схожим образом чи не має мату протилежній команді. Метод симуляції однаковий для цих перевірок і зображений у лістингу 3.11.

Лістинг 3.11 – Метод симуляції руху

```
private void SimulateForSinglePiece(ChessPiece cp, List<Movement>
moves, ChessPiece king)
{
List<Movement> movesToRemove = new List<Movement>();
Team attackingTeam = cp.Team == Team.White ? Team.Black :
Team.White;
for (int i = 0; i < moves.Count; i++)
{
    ChessPiece simulationPiece = cp.ShallowCopy();
    int simX = moves[i].Destination.x;
    int simY = moves[i].Destination.y;
    Vector2Int kingPositionThisSim = new Vector2Int(king.CurrentX,
king.CurrentY);

   if (cp.ChessType == ChessPieceType.King)
       kingPositionThisSim = new Vector2Int(simX, simY);
PieceArrangement simulation = _pieceArrangement.DeepCopy();
var possibleAttackingPieces = new
List<ChessPiece>(_chessGame.ChessPlayers[(int)
attackingTeam].ActivePieces);
simulation[simulationPiece.CurrentX, simulationPiece.CurrentY] =
null;
simulationPiece.CurrentX = simX;
simulationPiece.CurrentY = simY;
simulation[simX, simY] = simulationPiece;

List<Movement> simulatedMoveHistory = new
List<Movement>(_moveHistory);
simulatedMoveHistory.Add(new Movement(new
Vector2Int(simulationPiece.CurrentX, simulationPiece.CurrentY), new
Vector2Int(simX, simY), simulation));

var deadPiece = possibleAttackingPieces.Find(x => x.CurrentX == simX
&& x.CurrentY == simY);

if (deadPiece != null)
possibleAttackingPieces.Remove(deadPiece);

// Get all attacking pieces available moves
```

```
List<Movement> simMoves = new List<Movement>();
for (int a = 0; a < possibleAttackingPieces.Count; a++)
{
    var pieceMoves =
possibleAttackingPieces[a].GetAvailableMoves(simulation,
_chessBoardInfoSO.boardSizeX,_chessBoardInfoSO.boardSizeY,
simulatedMoveHistory);
  for (int b = 0; b < pieceMoves.Count; b++)
  {
    simMoves.Add(pieceMoves[b]);
  }
}
```

3.6 Мережевий код і використання шаблону проектування «Фасад»

Реалізація мережевої взаємодії відбувається за допомогою плагіну для Unity під назвою Photon PUN 2. Це популярний плагін для реалізації мережевої взаємодії який навіть надає безплатний план для 20 гравців чого було достатньо для реалізації, сервіс також надає можливість легко та швидко обновити план без зміни коду.

Цей плагін має дуже багато можливостей і дозволяє зробити як пошагову стратегію так і FPS шутер, тому має багато чого не потрібного для реалізації гри. Потрібно було обмежити використання всіх можливостей тільки для потреб гри, це б звузило можливості використання неправильних методів, та прискорили швидкість розробки бо не потрібно було відволікатися на не потрібні можливості. Тому код мережевої взаємодії було винесено в окремий сервіс і використано шаблон проектування Фасад.

Фасад — шаблон проектування, призначений для об'єднання групи підсистем під один уніфікований інтерфейс, надаючи доступ до них через одну точку входу. Це дозволяє спростити роботу з підсистемами.

Фасад належить до структурних шаблонів проектування.

Переваги:

– приховує реалізацію підсистеми від клієнтів, що полегшує використання

підсистеми;

– сприяє слабкій взаємодії між підсистемою та її клієнтами. Це дозволяє змінити класи, які включають підсистему, не впливаючи на клієнтів;

– зменшує компіляційні залежностей у великих програмних системах;

– спрощує системи перенесення на інші платформи, оскільки менш імовірно, що для побудови однієї підсистеми потрібно побудувати всі інші.

Недоліки:

– не заважає сучасним клієнтам отримувати доступ до базових класів;

– фасад не додає жодної функції, він просто спрощує інтерфейси.

В результаті було створено інтерфейс INetworkService, який став єдиною точкою взаємодії з мережею. Він інкапсулює основну логіку роботи з Photon і надає спрощений, чистий інтерфейс для інших частин проєкту. Інтерфейс наведено у лістингу 3.12.

Лістинг 3.12 – Інтерфейс мережевої взаємодії

```
public interface INetworkService : IService
{
        event Action<(string playerName, string playerScore)[]>
OnReadyToStartGame;

        event Action OnConnectedToMasterEvent;
        bool OfflineMode { get; }
        PhotonServiceSO.NetworkPlayer LocalPlayer { get; }

        public bool AutomaticallySyncScene { set; }
        public LobbyCallbacksDispatcher LobbyCallbacksDispatcher {
get; }
        public RoomCallbacksDispatcher RoomCallbacksDispatcher {
get; }
        public PhotonRPCSender PhotonRPCSender { get; }

        public List<PlayerInfoDTO> PlayersInfo { get; }

        void CreateRoom(string roomName, int timeLimitInSeconds, int
playerTeam, string password = null);

        void LoadGame();

        void LeaveRoom();

        void JoinLobby();
```

```
        void JoinRoom(string roomName = null);
}
```

Усі події, необхідні для обробки станів мережевої взаємодії — такі як підключення до майстер-сервера, приєднання до лобі, створення або приєднання до кімнати — були винесені в окремі колбеки, що значно спростило логіку, зокрема в UI для відображення цих подій.

3.7 Налагодження та аналіз продуктивності гри

Під час розробки гри важливим етапом була налагодження (debugging) та оптимізація продуктивності. Для цього використовувалися інструменти, що допомогли виявити помилки в коді та підвищити ефективність гри. Особлива увага приділялася стабільності гри під час багатокористувацької взаємодії та у стресових сценаріях.

Для моніторингу роботи гри в реальному часі та збору логів було використано SR Debugger. Це консольний інструмент, що дозволяє отримувати докладні логи про виконання гри та її взаємодію з різними підсистемами. Його інтеграція не потребує багато ресурсів, що дозволяє використовувати його навіть під час активної сесії гри без помітного впливу на продуктивність. За допомогою SR Debugger було можливим:

– запис та перегляд логів про стан гри, події та помилки;

– відстеження ключових подій, таких як рухи гравця, зміна стану гри, виконання команд;

– фільтрація та пошук у логах для виявлення аномалій чи помилок у реальному часі.

На рисунку 3.9 можна побачити консольне вікно SR Debugger-a.

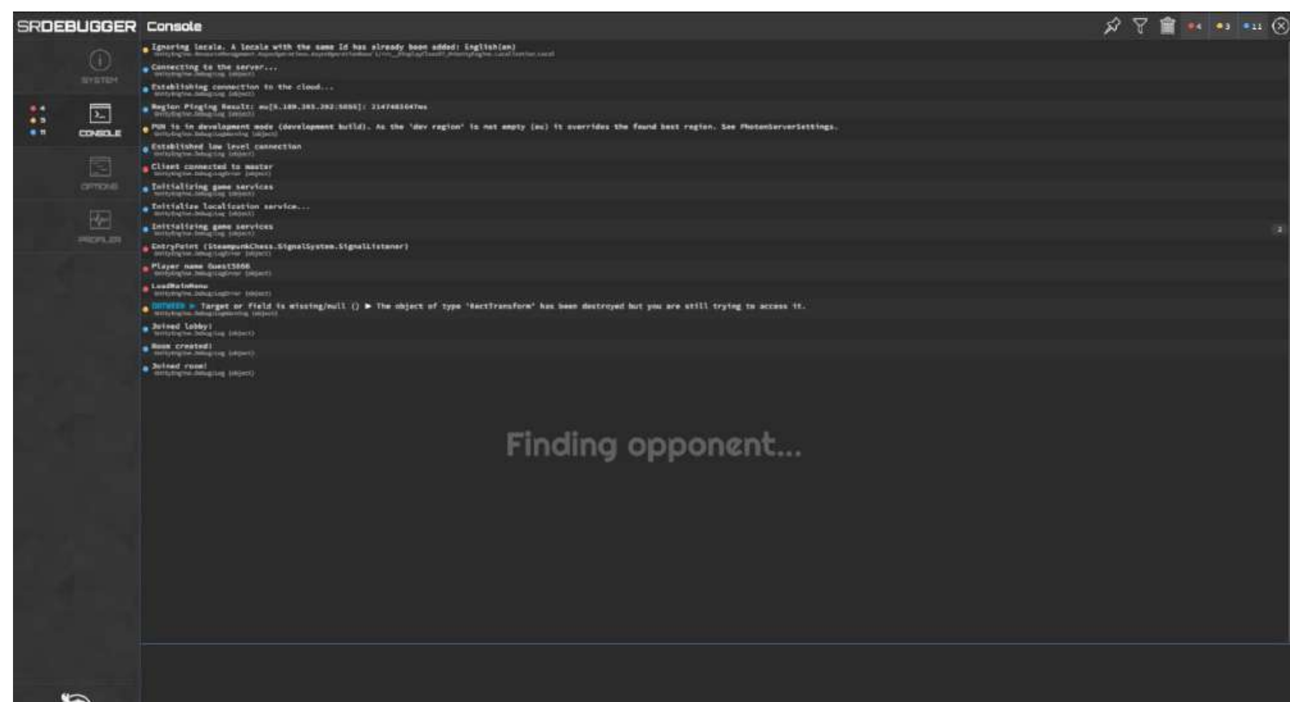


Рисунок 3.9 – SR Debugger

3.8 Локалізація

Локалізація була зроблена за допомогою офіційного плагіну від Unity. Для зручної загрузки та керування пам'яттю цей плагін використовує Addressables. Файл локалізації — це Google-таблиця, дані з якої завантажуються в Unity. Такий підхід має багато переваг, основна з них — це те, що для перекладу не потрібно взаємодіяти з Unity, а лише працювати з таблицею. Це дуже зручно, особливо якщо переклад замовляється на фрилансі. Таблиця складається з чотирьох стовпчиків: перший — це ключ строки, за допомогою якого можливо знайти фразу, решта стовпчиків — це переклади на мови.

Код взаємодії з локалізацією також було відокремлено в окремий сервіс, що значно спрощує масштабування та підтримку. Додатково, завдяки автоматичному оновленню даних із таблиці, можна швидко вносити правки або додавати нові мови без перекомпіляції гри. Така гнучкість особливо корисна під час підтримки проєкту після релізу.Частка таблиці перекладу зображена на рисунку 3.10.

	English(en)	Ukrainian(uk)
hello_text	Hello	Привіт
login_text	Log In	Увійти
signup_text	Sign Up	Реєстрація
username_text	Username	Логін
password_text	Password	Пароль
email_text	E-mail	Електронна пошта
rememberme_text	Remember me?	Запам'ятати мене?
forgotpassword_text	Forgot password?	Нагадати пароль?
alreadyhaveanaccount_text	Already have an account?	Вже є аккаунт?
send_text	Send	Відправити
play_text	Play	Грати
settings_text	Settings	Налаштунки
exit_text	Exit	Вихід
continue_as_guest_text	Continue as Guest	Продовжити як гість
incorrectpassword_text	Incorrect password	Невірний пароль
connectingtotheserver_text	Connecting to the server...	Підключення до сервера...
establishingconnectiontothecloud_	Establishing connection to the cloud...	Встановлення з'єднання з мережою...
initializinggameservices_text	Initializing game services...	Ініціалізація ігрових сервісів...
youhavesuccessfullylogged_text	You have successfully logged in	Ви ввійшли в аккаунт
youhavesuccessfullysigned_text	You have successfully signed in	Ви зареєструвалися
enterroompassword_text	Enter room password	Введіть пароль від кімнати
joingame_text	Join game	У гру
nametext	Name:	Ім'я:
timetext	Time:	Час:
jointext	Join	У гру
createtext	Create game	Створити гру
randomgame_text	Random game	Випадкова гра
nameroom_text	Name	Назва
team_text	Team	Команда
time_text	Time	Час
createroom_text	Create room	Створити кімнату
gametiming_text	{time} min	{time} хв
gametimenolimit_text	No limit	Безліміту
exitthegame_text	Exit the game?	Вийти з гри?

Рисунок 3.10 – Таблиця локалізації

3.9 Реалізація стандартів позначення позицій FEN та PGN

Fen або Нотація Форсайта - Едвардса — один із видів шахової нотації, яку зазвичай застосовують, щоб описати поточну позицію (наприклад, для текстового подання шахової задачі). Також лаконічний запис початкової позиції важливий для такого різновиду гри як «шахи 960» (шахи Фішера) [13].

Portable Game Notation (PGN) — формат файлу для збереження шахових партій. Він був розроблений Стівеном Едвардсом (англ. Steven J. Edwards) в 1994 році, щоб полегшити обмін партіями (наприклад через інтернет) між шаховими програмами [13].

Формат PGN використовує символи з ASCII-кодування і складається з двох частин: метадані і нотація партії. У першій частині міститься інформація про партію: турнір, тур, імена гравців, результат тощо. Друга частина складається з алгебраїчної нотації. Коментарі розташовують між {} дужок [13].

Більшість шахових програм підтримує цей формат. Обробка файлів може відбуватися і за допомогою звичайної програми для редагування тексту. В одному

файлі можна зберігати більше, ніж одну партію [13].

PGN використовувався у грі для відображення зроблених ходів користувачу, а FEN використовувався для зберігання позицій. Саме з нього створюється початкова позиція. На рисунку 3.11 зображено конфіг з FEN строкою, яка використовується для завантаження позицій фігур у грі.

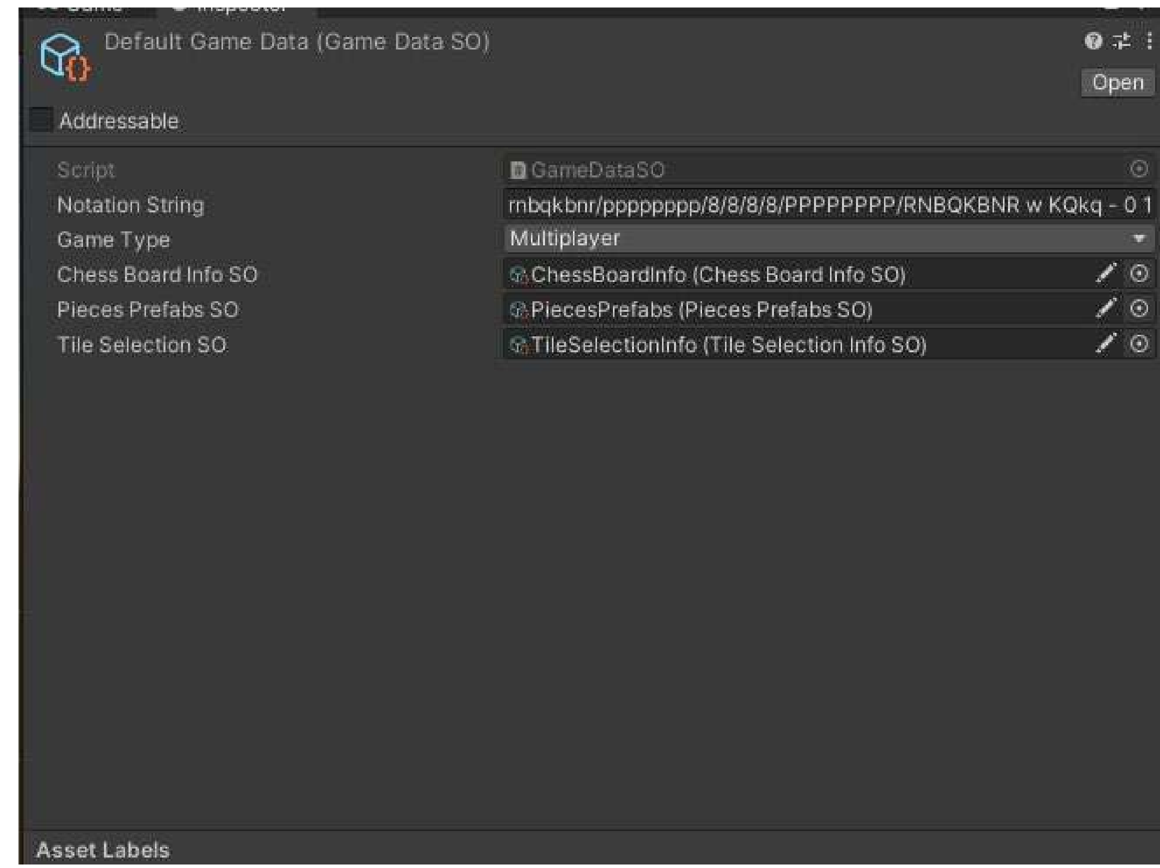


Рисунок 3.11 – Конфігурація гри

3.10 Висновки

Завдяки чітким стандартам, ітеративному підходу к розробці і увагу до деталей вдалось зробити гнучку архітектуру гри, код гри прозорий і при приєднанні до розробки іншого програміста буде швидко зрозумілий. Це дозволило мінімізувати технічний борг та забезпечити стабільність під час подальшого

розвитку проєкту. Впровадження чітких принципів кодування також допомогло зменшити кількість помилок і полегшити їх виявлення на ранніх етапах. Завдяки правильному використанню шаблонів проектування вдалося позбутися деяких умовних конструкцій, що теж зробило код більш зрозумілим, а за допомогою інверсії залежностей вдалося чітко контролювати залежності і не використовувати антипатерн «Одинак».

Також, використання готових рішень, таких як Addressables, Unity Localization, SR Debugger та інших, суттєво прискорило розробку. Це дозволило зосередитися на реалізації геймплею та логіки гри, не витрачаючи час на створення технічних рішень з нуля.

Такий підхід значно спростив тестування і підтримку коду, а також сприяв більш ефективній командній роботі. В результаті архітектура стала більш масштабованою і готовою до впровадження нових функцій без суттєвих змін у базі коду. Готова версія гри зображена на рисунку 3.12.

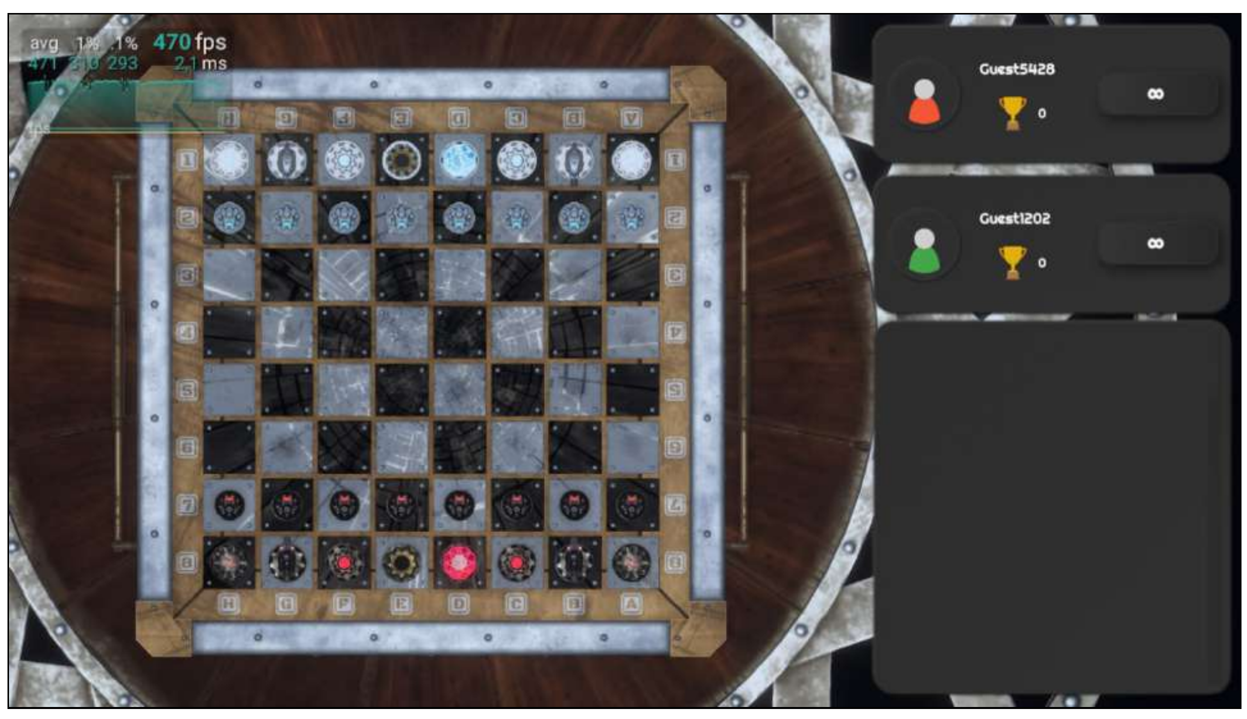


Рисунок 3.12 – Ігрова сессія, вид з верху

4 ВИПРОБОВУВАННЯ СИСТЕМИ

Забезпечення якості є важливим компонентом розробки ігор, хоча індустрія відеоігор не має стандартної методології. Натомість розробники та видавці мають власні методи. Маленькі розробники, як правило, не мають персоналу з контролю якості; однак великі компанії можуть наймати команди QA на повний робочий день. Високі комерційні ігри професійно та ефективно перевіряються відділом контролю якості видавців.

У ході розробки гри кожне вікно ретельно тестувалося, всі помилки було документовано і вирішено у порядку серйозності.

4.1 Огляд головних вікон гри

4.1.1 Заставка

Заставка — це графічний елемент керування, що складається з вікна , що містить зображення , логотип і поточну версію програмного забезпечення. Зазвичай з'являється під час запуску гри.

Заставки зазвичай використовуються особливо великими додатками для сповіщення користувача про те, що програма завантажується. Вони повідомляють про те, що триває тривалий процес. Іноді на екрані заставки вказує хід завантаження . Заставка зникає, коли з'являється головне вікно програми. Заставки можна додати на деякий час, а потім замінити заново.

У грі, гравця зустрічає заставка з логотипом яка зображена на рисунку 4.1. При натисканні будь-якої кнопки починають грузитися ігрові сервіси, та відбувається перевірка чи є у гри дані про гравця, відбувається спроба увійти в аккаунт, якщо вони успішна, тоді гравця переносить у головне меню, якщо ні то гравцю пропонують зареєструватися або увійти в аккаунт.

Рисунок 4.1 – Заставка гри

4.1.2 Вікна реєстрації та авторизації

Авторизація − керування рівнями та засобами доступу до певного захищеного ресурсу, як в фізичному розумінні, так і в галузі цифрових технологій (наприклад, автоматизована система контролю доступу) та ресурсів системи залежно від ідентифікатора і пароля користувача або надання певних повноважень (особі, програмі) на виконання деяких дій у системі обробки даних.

Гравець може вільно переміщуватися між трьома вікнами реєстрації, авторизації та вікна «Забули пароль?». Також є можливість увійти у гру як гість, при цьому ви зможете грати з іншими гравцями, але дані не будуть збережені. Вікно авторизації зображене на рисунку 4.2.

Механізм авторизації був реалізований самостійно, без використання вбудованих рішень, що дало більше контролю над логікою входу та реєстрації. Для збереження даних користувача, таких як прогрес гри або налаштування, використовувався сервіс PlayFab, який також забезпечує хмарне зберігання та зручний API для доступу до профілів гравців. Завдяки використанню PlayFab вдалося забезпечити надійне та швидке збереження даних, що важливо для підтримки мультиплеєрного режиму та синхронізації між гравцями.

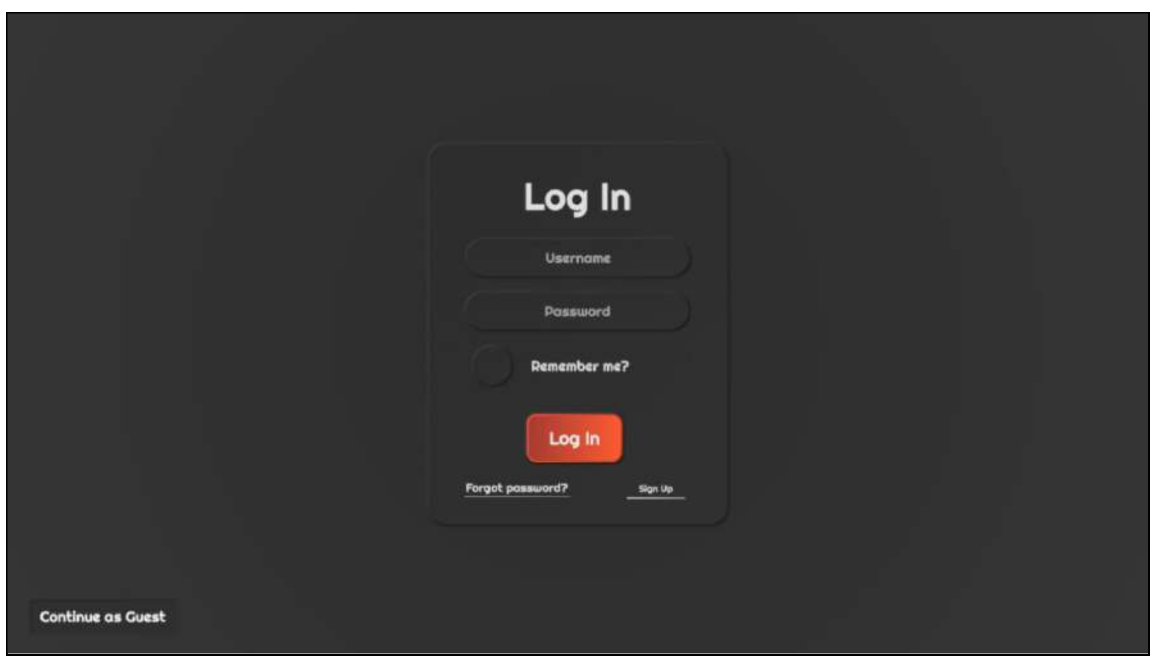


Рисунок 4.2 – Вікно авторизації

4.1.3 Головне меню

Ігрове меню — це елемент графічного інтерфейсу користувача, який надає доступ до різноманітних функцій гри. Меню іноді організовано ієрархічно, що дозволяє здійснювати навігацію між різними рівнями структури меню. Вибір пункту меню за допомогою стрілки розширює його, показуючи друге меню (підменю) з параметрами, пов'язаними з вибраним записом.

У грі меню скаладається з трьох кнопок. Кнопка «Play» переносить користувача у лобі, де він може бачити активні кімнати, вступати в них або створювати нові. Кнопка «Налаштування» відкриває вікно графічних, аудіо та інших налаштувань гри. Кнопка «Вихід» дозволяє вийти з гри. Переміщення між вікнами відбувається за допомогою кнопки Escape. Головне меню зображено на рисунку 4.3.

Дизайн меню виконаний у стилі, який відповідає загальній атмосфері гри, забезпечуючи гармонійний візуальний досвід. Також передбачено адаптивне масштабування інтерфейсу, що дозволяє коректно відображати меню на екранах різних розмірів та роздільної здатності та співвідношення сторін, завдяки чому користувачі отримують зручний і зрозумілий інтерфейс як на великих моніторах, так і на компактних пристроях.

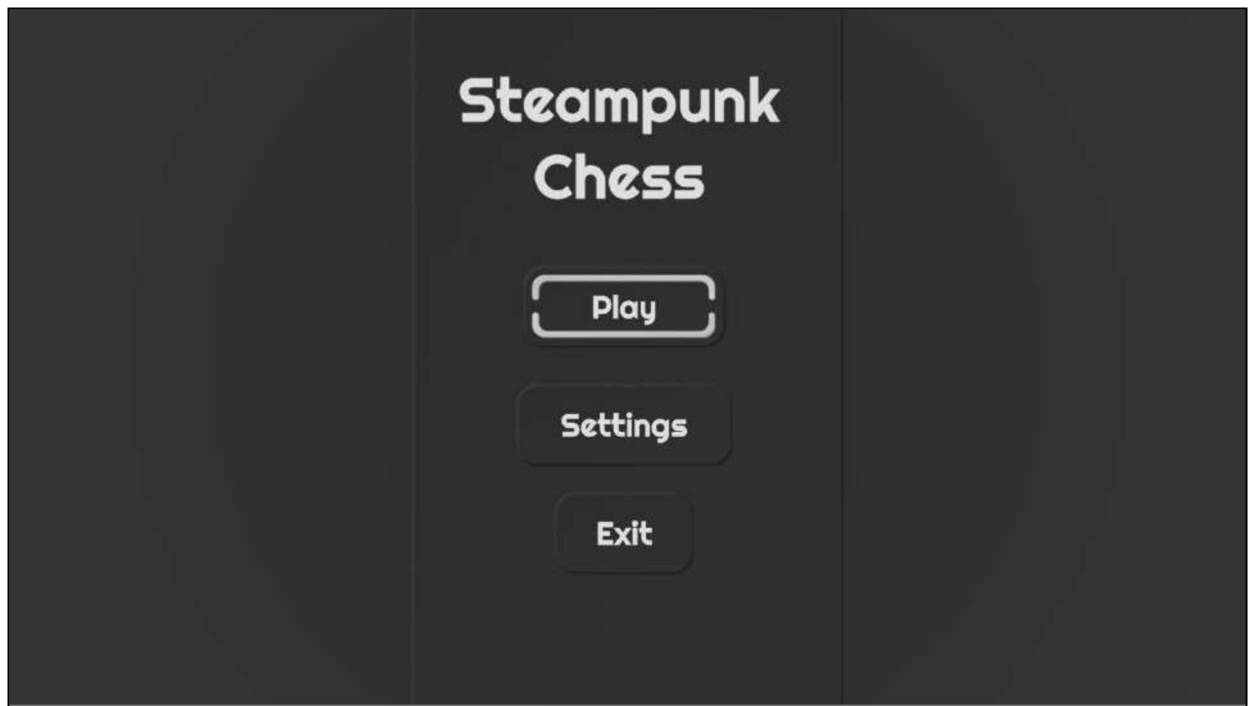


Рисунок 4.3 – Головне меню

4.1.4 Лобі

Лобі — це екрани меню, де гравці можуть переглядати майбутній ігровий сеанс, переглядати результати останнього. Лобі гри зображено на рисунку 4.4.

У багатьох іграх гравці повертаються у лобі в кінці кожного сеансу. У деяких випадках гравці, які приєднуються до сеансу, що вже розпочався, розміщуються у лобі до початку наступного. Оскільки у лобі споживається дуже мало ресурсів, вони іноді додатково використовуються як меню завантаження для гравців, поки знайдеться відповідний хост для майбутнього сеансу.

Лобі, створені списками відтворення, часто мають таймер зворотного відліку перед початком сеансу, тоді як лобі, створені гравцем, зазвичай змінюються на розсуд гравця.

У лобі гри гравець може створити нову кімнату або вступити у існуючу. При створенні нової кімнати відкривається вікно, де можна налаштувати час гри, команду гравця, назву кімнати та встановити пароль для захисту. Крім того, у лобі передбачені базові функції комунікації між гравцями, що дозволяє обговорювати майбутній матч або координувати стратегію. Це створює зручне та дружнє середовище, яке сприяє кращій взаємодії між учасниками перед початком гри.

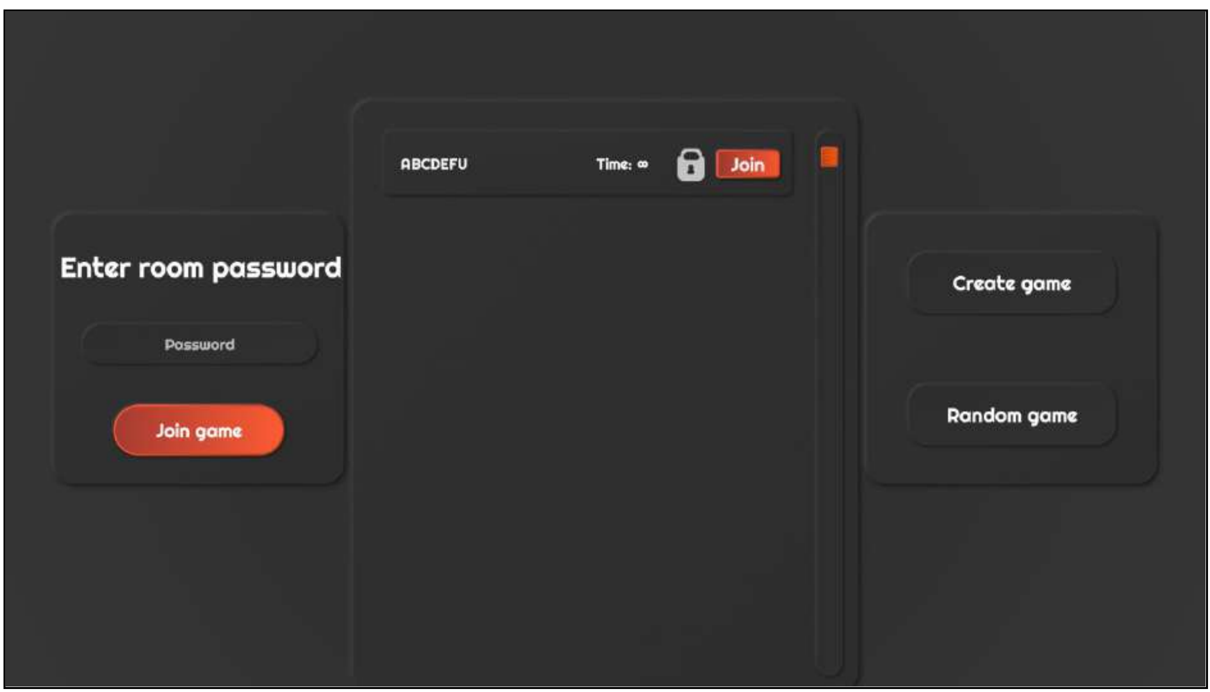


Рисунок 4.4 – Лобі гри

4.2 Модульне тестування

Модульне тестування — це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яка може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру.

Модульні тести, або unit-тести, розробляють в процесі розробки програмісти та, іноді, тестувальники білої скриньки (white-box testers).

Зазвичай unit-тести застосовують для того, щоб упевнитися, що код відповідає вимогам архітектури та має очікувану поведінку.

Модульне тестування дозволяє програмісту, коли він буде змінювати код (проводити рефакторинг) бути впевненим, що модуль працює вірно (це — регресивне тестування). Оскільки модульне тестування вимагає написання тестів для всіх функцій та методів у програмі, помилки швидко локалізуються та виправляються.

Модульне тестування може бути застосоване в інтеграційному тестуванні, тестування окремих модулів та сукупності цих модулів робить інтеграційне тестування легшим. Однак модульне тестування знизу вгору не є інтеграційним тестуванням. Інтеграція з зовнішніми модулями має включатися до інтеграційних тестів, а не до модульних.

Модульні тести являють собою специфічний вид документації до системи. Розробники можуть подивитися на модульний тест, щоб дізнатися про функції, що виконує модуль, та як його застосовувати.

Unit-тест перевіряє критичні характеристики модуля. Відповідність чи невідповідність цим характеристикам демонструє коректність модуля. Модульний тест «документує» ці критичні характеристики, але не треба покладатися лише на код в документуванні ПЗ під час розробки.

Слід відзначити, що звичайна письмова документація дуже повільно реагує на зміни в коді, тоді як модульні тести завжди відображають поточний стан модуля. Крім того, тести допомагають швидко виявляти регресії та забезпечують безпеку під час рефакторингу. Вони також сприяють підвищенню довіри до якості коду серед усієї команди розробників. Використання юніт-тестів дозволяє автоматизувати перевірку ключових функціональностей і зменшити час на ручне тестування. Завдяки цьому процес розробки стає більш ефективним і контрольованим. Під час розробки програмного забезпечення методом TDD, модульний тест стає частиною дизайну. Кожен тест визначає потрібні класи та методи, їх очікувану поведінку.

У грі були написані тести для багатьох функцій і класів програми, це дозволило бути впевненим у змінюванні коду, і що це не приведе до якихось помилок. Такий підхід значно підвищив стабільність проєкту та спростив процес внесення нових функціональних можливостей. Для написанння юніт-тестів використовувалася поширена практика Arrange-Act-Assert, що робить тести значно зрозумільшими. Приклад unit-тесту та огляд робочих тестів зображено на рисунках 4.5 та 4.6.

```
[Test]
quizandpuzzle *
public void WhenKingAndRookH1DidntMove_AndThereIsNoObstacleToThem_ThenAvailableMovesShouldIncludeCastling()
{
    //Arrange
    var notationString = new FenNotationString("rnbqkbnr/pp1p1ppp/2p5/1B2p3/8/4PN2/PPPP1PPP/RNBQK2R b KQkq - 1 3");
    var chessBoardInfoSO = Container.Resolve<ChessBoardInfoSO>();
    var piecesPrefabsSO = Container.Resolve<PiecesPrefabsSO>();
    var chessPieceFactory = Container.Resolve<ChessPieceFactory>();
    var pieceArrangement = new PieceArrangement(notationString, chessBoardInfoSO, piecesPrefabsSO, chessPieceFactory);
    var tileSet = new TileSet(chessBoardInfoSO);
    pieceArrangement.Initialize();
    tileSet.Initialize();
    var moveHistory = new List<Movement>()
    {
        new Movement( start: new Vector2Int( x: 5, y: 6), destination: new Vector2Int( x: 5, y: 4),pieceArrangement),
    };
    var whiteKing :ChessPiece = pieceArrangement[4, 0];
    var expectedCastlingMove = new Movement( start: new Vector2Int( x: 4, y: 0),
        destination: new Vector2Int( x: 6, y: 0),
        pieceArrangement,
        specialMove: new Castling(moveHistory, pieceArrangement));

    //Act
    var availableMoves :List<Movement> = whiteKing.GetAvailableMoves(pieceArrangement, tileCountX: 8, tileCountY: 8, moveHistory);

    //Assert
    availableMoves.Should().ContainEquivalentOf(expectedCastlingMove);
}
```

Рисунок 4.5 – Unit-тест

```
✓ Steampunk Chess
  ✓ Tests.dll
    ✓ SteampunkChess
      ✓ Tests
        ✓ ChessBoardTests
          ✓ FindAllIndexesListExtensionTest
          ✓ WhenCreatingMoveWithKnightOnB1ToC3_AndGetMovePGN_ThenMovePGNShouldBeC3
          ✓ WhenCreatingTiles_AndLookupTileIndex_ThenTileIndexShouldBeAsStated
          ✓ WhenGivenFenAndParsedToGameData_AndParseBackToFen_ThanFenTextShouldBeTheSame
          ✓ WhenGivenStartPositionFen_AndParseToGameData_ThanPieceArrangementShouldBeAsExpected
          ✓ WhenKingAndRookH1DidntMove_AndThereIsNoObstacleToThem_ThenAvailableMovesShouldIncludeCastling
          ✓ WhenSpawningChessPiece_AndPositionPieceXY55_ThenXYShouldBe55
          ✓ WhenWhitePawnOnE5_AndBlackPawnOnF5IsPreviousMove_ThanAvailableMovesShouldIncludeEnPassant
          ✓ WhenWhitePawnOnG7_AndCanMoveG8_ThanAvailableMovesShouldContainPromotion
        ✓ ParseCLIArgsTests
          ✓ WhenGameArgsAreEmpty_AndAddSkipUserValidationArg_ThenGameArgsShouldContainSkipUserValidationArg
  ✓ Zenject-UnitTests-Editor.dll
    ✓ Zenject
      ✓ Tests
        ✓ AbstractFactory
          ✓ TestFactory
            ✓ TestToSelf
        ✓ BindFeatures
          ✓ TestConcreteIdentifier
            ✓ Test1
          ✓ TestIfNotBound
            ✓ Test1
            ✓ Test2
          ✓ TestMultipleContractTypes
            ✓ TestMultiBind1
            ✓ TestMultiBindListInjection
          ✓ TestMultipleContractTypes2
            ✓ Test1
            ✓ TestAllInterfaces
            ✓ TestAllInterfacesAndSelf
            ✓ TestAllInterfacesAndSelfMistake
            ✓ TestAllInterfacesMistake
            ✓ TestInterfaces
          ✓ TestMultipleContractTypes3
            ✓ TestMultiBind2
          ✓ TestRebind
            ✓ Run
          ✓ TestUnbind
            ✓ Run
```

Рисунок 4.6 – Вікно з працюючими тестами

4.3 Деплой проекту

Деплой проекту — це процес підготовки гри до запуску на цільових платформах та її розповсюдження серед користувачів.

Для зручності роботи та економії часу в проекті був створений пайплайн автоматизації збірок за допомогою бібліотеки Unity.Cake. Ця бібліотека дозволяє налаштувати процес збірки гри, включаючи вибір цільової платформи, оптимізацію ресурсів та генерацію виконуваних файлів, у вигляді скриптів, які запускаються автоматично.

Налаштований build.cake скрипт можна побачити у лістингу 4.1.

Лістинг 4.1 – Build.Cake скрипт для автоматизації збірки гри

```
var target = Argument("target", "Build-Dev");

Task("Clean-Artifacts")
    .Does(() =>
{
    CleanDirectory($"./artifacts");
});

Task("Build-Release")
    .IsDependentOn("Clean-Artifacts")
    .Does(() =>
{
        UnityEditor(2020,3,
      new UnityEditorArguments{
           BatchMode = true,
           Quit = true,
           ExecuteMethod                                              =
 "SteampunkChess.Editor.GameBuilder.BuildRelease",
           BuildTarget = BuildTarget.Win64,
           LogFile = "./artifacts/unity.log",

      },
      new UnityEditorSettings{
           RealTimeLog = true,
      });

});

RunTarget(target);
```

Для запуску процесу збірки гри достатньо виконати команду dotnet cake у терміналі або командному рядку. Ця команда активує скрипт автоматизації, створений за допомогою Unity.Cake, який виконує всі необхідні дії для збірки гри. Після запуску скрипт автоматично налаштовує параметри збірки, генерує виконувані файли для вибраної платформи та зберігає їх у визначеній директорії. Завдяки цьому підходу процес білду стає максимально простим і зручним.

Крім того, такий метод дозволяє легко підтримувати і оновлювати процес збірки без необхідності вручну змінювати налаштування, що значно підвищує ефективність розробки. Автоматизація також сприяє зниженню кількості помилок, пов’язаних з людським фактором, забезпечуючи стабільність і передбачуваність результатів. Результат збірки наведено на рисунку 4.7.

Name	Date modified	Type	Size
MonoBleedingEdge	1/8/2025 9:36 PM	File folder	
Steampunk Chess_Data	1/8/2025 10:18 PM	File folder	
Steampunk Chess.exe	1/8/2025 10:18 PM	Application	639 KB
UnityCrashHandler64.exe	10/26/2021 5:31 PM	Application	1,204 KB
UnityPlayer.dll	10/26/2021 5:31 PM	DLL File	27,495 KB

Рисунок 4.7 – Файли збірки гри

У створеній директорії знаходяться всі необхідні файли, включаючи виконуваний файл гри, який можна запустити для перевірки її роботи. Згенеровані файли містять усі ресурси та налаштування, необхідні для коректного функціонування гри, що дозволяє розпочати тестування або використовувати їх для розповсюдження серед користувачів.

ВИСНОВКИ

В ході представленої роботи було досліджено місце комп’ютерних та мобільних ігор в сьогоднішньому світі.

Була обгрутована економічна доцільність проекту та отриманий розмір економії, який і є підтвердженням економічної вигоди та доцільності виконання даних робіт власними спеціалістами.

Були розглянуті такі питання по охороні праці:

– вимоги до безпеки робочих місць;

– вимоги безпеки під час роботи з екранними пристроями;

– вимоги безпеки до екранних пристроїв;

– дотримання вимог електробезпеки під час роботи.

В ході роботи було розібрано існуючі ігрові двигуни – платформи для розробки ігор для комп’ютерів, телефонів, консолей і тд. Було виявлено найбільш поширені та найбільш потужні двигуни, якими стали Unity, Unreal Engine 4, CryEngine, Godot. Було проаналізовано кожен із них, його сильні та слабкі сторони. В результаті було отримано аналіз можливостей кожного з двигунів і його абстрактний рейтинг по відношенню до поставленої задачі гри.

Було створено гру на вибраному двигуні Unity. За жанром гра є логіческою грою з можливістю гри через мережу Інтернет. Освоєння рушія Unity несе не маловажний характер, так як у світі індустрія розробки ігор дедалі більше поширюється у суспільстві. Ігри перестали бути лише предметом для розваг, і тепер використовуються і в інших областях, наприклад, у науці чи навчання користувачів. Окрім, того ігровий рушій Unity використовується не тільки для створення ігор, а й для створення фільмів та моделювання архітектури, 3D-візуалізації. Тому розвиток у цьому напрямі можна вважати одним із найважливіших у суспільстві.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Комп'ютерні ігри у навчанні історії. URL: https://pidru4niki.com/2015082665979/informatika/kompyuterni_igri_navchanni_istoriyi (дата звернення: 02.05.2025).

2 History of video games. URL: https://www.history.com/articles/history-of-video-games (date of access: 03.05.2025).

3 Game development stages. URL: https://pinglestudio.com/blog/full-cycle-development/game-development-stages (date of access: 06.05.2025).

4 Game Engine. URL: https://gameopedia.com/blogs/game-engines-all-you-need-to-know (date of access: 07.05.2025).

5 Unity 3D. URL: https://support.unity.com/hc/en-us/articles/206336795-What-platforms-are-supported-by-Unity (date of access: 08.05.2025).

6 Unreal Engine. URL: https://www.unrealengine.com/en-US/faq (date of access: 08.05.2025).

7 CryEngine (серія рушіїв). URL: https://www.cryengine.com/support/view/general (дата звернення: 08.05.2025).

8 Godot – Особливості різних платформ. URL: https://docs.godotengine.org/uk/4.x/tutorials/platform/ (date of access: 08.05.2025).

9 Коноваленко І. В. Програмування мовою C# 6.0. Львів : ТНТУ, 2016. 46 с.

10 Model-View-ViewModel (MVVM). URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (date of access: 10.05.2025).

11 Zenject – Dependency Injection Framework. URL: https://github.com/modesttree/Zenject (date of access: 10.05.2025).

12 Одинак (шаблон проєктування). URL: https://refactoring.guru/uk/design-patterns/singleton (date of access: 10.05.2025).

13 Forsyth-Edwards Notation. URL: https://www.chess.com/terms/fen-chess (date of access: 11.05.2025).

ДОДАТОК А
ЛІСТИНГИ ПРОГРАМИ

Лістинг А.1 – Основні класи програми

```
public abstract class ChessBoard
    {
        private readonly TileSet _tileSet;
        private readonly PieceArrangement _pieceArrangement;
        private readonly ChessBoardInfoSO _chessBoardInfoSO;
        private readonly List<Movement> _moveHistory;
        private readonly MoveListing _moveListing;
        private readonly TileSelection _tileSelection;
        private readonly IAudioSystem _audioSystem;

        private ChessPiece _activePiece;
        private IChessGame _chessGame;
        private List<Movement> _availableMoves;
        private bool _processingMove;

        protected ChessBoard(ChessBoardData chessBoardData, MoveListingData
moveListingData,    ChessPieceFactory    chessPieceFactory,    IAudioSystem
audioSystem)
        {
            _audioSystem = audioSystem;
            _chessBoardInfoSO = chessBoardData.ChessBoardInfoSO;
            _moveHistory = new List<Movement>();
            _moveListing = new MoveListing(moveListingData, _moveHistory);
            _tileSelection                      =                      new
TileSelection(chessBoardData.TileSelectionSO);
            _tileSet = new TileSet(_chessBoardInfoSO);
            _pieceArrangement                      =                      new
PieceArrangement(chessBoardData.NotationString,          _chessBoardInfoSO,
chessBoardData.PiecesPrefabsSO, chessPieceFactory);
        }

        public ChessPiece this[int x, int y] => _pieceArrangement[x, y];
```

```
        public virtual void Initialize(IChessGame chessGame)
        {
            _chessGame = chessGame;
            InitializeBoardComponents();
        }

        private void InitializeBoardComponents()
        {
            _tileSet.Initialize();
            _pieceArrangement.Initialize();
            _tileSelection.Initialize();
        }

        protected abstract void MoveTo(Vector2 moveTo);
        protected  abstract  void  SelectPieceAndShowAvailableMoves(Vector2
hitPosition);

        public void OnTileHover(GameObject tile)
        {
            if (_processingMove || !_chessGame.CanPerformMove())
                return;

            Vector2Int hitPosition = _tileSet.LookupTileIndex(tile);
            ChessPiece cp = _pieceArrangement[hitPosition.x, hitPosition.y];

            bool onTileClicked = Input.GetMouseButtonUp(0);

            if (onTileClicked)
            {
                if (_activePiece != null)
                {
                    if (cp != null && _activePiece.IsFromSameTeam(cp))
                    {
                        SelectPieceAndShowAvailableMoves(hitPosition);
                    }
                    else    if    (SearchForMoveDestination(_availableMoves,
hitPosition, out Movement move))
                    {
```

```
                    MoveTo(move.Destination);
                }
            }
            else
            {
                if (cp != null && _chessGame.IsTeamTurnActive(cp.Team))
                    SelectPieceAndShowAvailableMoves(hitPosition);
            }
        }
    }

    private bool SearchForMoveDestination(List<Movement> moves,
Vector2Int moveDestination, out Movement move)
    {
        for (int i = 0; i < moves.Count; i++)
        {
            if (moves[i].Destination == moveDestination)
            {
                move = moves[i];
                return true;
            }
        }

        move = null;
        return false;
    }

    protected void OnSelectPieceAndShowAvailableMoves(Vector2Int
hitPosition)
    {
        _activePiece = _pieceArrangement[hitPosition.x, hitPosition.y];
        _availableMoves =
_activePiece.GetAvailableMoves(_pieceArrangement,
_chessBoardInfoSO.boardSizeX,
            _chessBoardInfoSO.boardSizeY, _moveHistory);
        RemoveMovesToPreventCheck();
        ShowAvailableMoves();
    }
```

```
        private void ShowAvailableMoves()
        {
            List<(Vector3, bool)> tileData = new List<(Vector3, bool)>();
            for (int i = 0; i < _availableMoves.Count; i++)
            {
                Vector3 tileCenter =
                    TileSet.GetTileCenter(_availableMoves[i].Destination.x,
_availableMoves[i].Destination.y);
                tileData.Add((tileCenter,
_availableMoves[i].IsAttackMove));
            }

            _tileSelection.ShowSelection(tileData);
        }


        protected async void OnMoveTo(Vector2Int moveTo)
        {
            if (SearchForMoveDestination(_availableMoves, moveTo, out
Movement move))
            {
                _processingMove = true;
                _tileSelection.ClearSelection();
                _moveHistory.Add(move);
                _moveListing.UpdateMoveListing();
                await move.Process();
                _audioSystem.PlaySound(Sounds.PieceMoveSound);

                if (IsCheckmated())
                {
                    _chessGame.EndOfGame(move.MovePiece.Team);
                }

                _activePiece = null;
                _chessGame.ChangeActiveTeam();
                _processingMove = false;
            }
        }
```

```
        private void RemoveMovesToPreventCheck()
        {
            ChessPiece targetKing = _chessGame.ActivePlayer.GetPiecesOfType<King>().First();
            SimulateForSinglePiece(_activePiece, _availableMoves, targetKing);
        }

        private void SimulateForSinglePiece(ChessPiece cp, List<Movement> moves, ChessPiece king)
        {

            List<Movement> movesToRemove = new List<Movement>();
            Team attackingTeam = cp.Team == Team.White ? Team.Black : Team.White;


            for (int i = 0; i < moves.Count; i++)
            {
                ChessPiece simulationPiece = cp.ShallowCopy();
                int simX = moves[i].Destination.x;
                int simY = moves[i].Destination.y;

                Vector2Int kingPositionThisSim = new Vector2Int(king.CurrentX, king.CurrentY);

                if (cp.ChessType == ChessPieceType.King)
                    kingPositionThisSim = new Vector2Int(simX, simY);


                PieceArrangement simulation = _pieceArrangement.DeepCopy();
                var possibleAttackingPieces = new List<ChessPiece>(_chessGame.ChessPlayers[(int)attackingTeam].ActivePieces);


                // Simulate that move
                simulation[simulationPiece.CurrentX, simulationPiece.CurrentY] = null;
```

```
                simulationPiece.CurrentX = simX;
                simulationPiece.CurrentY = simY;
                simulation[simX, simY] = simulationPiece;


                List<Movement>       simulatedMoveHistory       =       new
List<Movement>(_moveHistory);
                simulatedMoveHistory.Add(new                   Movement(new
Vector2Int(simulationPiece.CurrentX,     simulationPiece.CurrentY),     new
Vector2Int(simX, simY), simulation));

                // Did one of the pieces got taken down during  simulation
                var deadPiece = possibleAttackingPieces.Find(x => x.CurrentX
== simX && x.CurrentY == simY);

                if (deadPiece != null)
                    possibleAttackingPieces.Remove(deadPiece);

                // Get all attacking pieces available moves
                List<Movement> simMoves = new List<Movement>();
                for (int a = 0; a < possibleAttackingPieces.Count; a++)
                {
                    var                    pieceMoves                    =
possibleAttackingPieces[a].GetAvailableMoves(simulation,
_chessBoardInfoSO.boardSizeX,
                        _chessBoardInfoSO.boardSizeY,
simulatedMoveHistory);
                    for (int b = 0; b < pieceMoves.Count; b++)
                    {
                        simMoves.Add(pieceMoves[b]);
                    }
                }

                //Is that spot safe for king?
                if (SearchForMoveDestination(simMoves, kingPositionThisSim,
out Movement move))
                {
                    movesToRemove.Add(moves[i]);
                }
```

```
            }

            //Remove  some moves from availableMoves
            for (int i = 0; i < movesToRemove.Count; i++)
            {
                moves.Remove(movesToRemove[i]);
            }
        }

        private bool IsCheckmated()
        {
            var lastMove = _moveHistory[_moveHistory.Count - 1];
            Team attackingTeam = _pieceArrangement[lastMove.Destination.x,
lastMove.Destination.y].Team;
            Team targetTeam = (attackingTeam == Team.White) ? Team.Black :
Team.White;

            IReadOnlyList<ChessPiece>           attackingPieces           =
_chessGame.ChessPlayers[(int) attackingTeam].ActivePieces;
            IReadOnlyList<ChessPiece>           defendingPieces           =
_chessGame.ChessPlayers[(int) targetTeam].ActivePieces;
            ChessPiece    targetKing    =    _chessGame.ChessPlayers[(int)
targetTeam].GetPiecesOfType<King>().First();



            List<Movement> attackingMoves = new List<Movement>();
            for (int a = 0; a < attackingPieces.Count; a++)
            {
                var                         pieceMoves                     =
attackingPieces[a].GetAvailableMoves(_pieceArrangement,
_chessBoardInfoSO.boardSizeX, _chessBoardInfoSO.boardSizeY, _moveHistory);
                for (int b = 0; b < pieceMoves.Count; b++)
                {
                    attackingMoves.Add(pieceMoves[b]);
                }
            }

            if        (SearchForMoveDestination(attackingMoves,         new
```

```
Vector2Int(targetKing.CurrentX, targetKing.CurrentY), out _))
            {
                //King is under attack, can we defend him?
                for (int i = 0; i < defendingPieces.Count; i++)
                {
                    var                  defendingMoves                  =
defendingPieces[i].GetAvailableMoves(_pieceArrangement,
_chessBoardInfoSO.boardSizeX, _chessBoardInfoSO.boardSizeY, _moveHistory);
                    //Delete moves that give us check
                    SimulateForSinglePiece(defendingPieces[i],
defendingMoves, targetKing);

                    if (defendingMoves.Count != 0)
                        return false;
                }
                return true;
            }
            return false;
        }
    }

public class PieceArrangement : IInitializable
    {
        private ChessPiece[,] _chessPieces;

        private readonly NotationString _notationString;
        private readonly ChessBoardInfoSO _chessBoardInfoSO;
        private readonly PiecesPrefabsSO _piecesPrefabsSO;
        private readonly ChessPieceFactory _chessPieceFactory;

        private const string PiecesParentName = "Pieces";
        private Transform _piecesParent;

        public       PieceArrangement(NotationString       notationString,
ChessBoardInfoSO    chessBoardInfoSO,    PiecesPrefabsSO    piecesPrefabsSO,
ChessPieceFactory chessPieceFactory)
        {
            _notationString = notationString;
            _chessBoardInfoSO = chessBoardInfoSO;
```

```
            _piecesPrefabsSO = piecesPrefabsSO;
            _chessPieceFactory = chessPieceFactory;
            _chessPieces = new ChessPiece[chessBoardInfoSO.boardSizeX,
chessBoardInfoSO.boardSizeY];
        }


        public ChessPiece this[int x, int y]
        {
            get => _chessPieces[x, y];
            set => _chessPieces[x, y] = value;
        }

        public PieceArrangement DeepCopy()
        {
            PieceArrangement pieceArrangement = new
PieceArrangement(_notationString, _chessBoardInfoSO, _piecesPrefabsSO,
_chessPieceFactory);

            for (int x = 0; x < _chessBoardInfoSO.boardSizeX; x++)
            {
                for (int y = 0; y < _chessBoardInfoSO.boardSizeY; y++)
                {
                    if (this[x, y] != null)
                    {
                        pieceArrangement[x, y] = this[x, y];
                    }
                }
            }

            return pieceArrangement;
        }

        public void Initialize()
        {
            _chessPieces = SpawnAllPieces(_notationString,
(_chessBoardInfoSO.boardSizeX, _chessBoardInfoSO.boardSizeY));
            PositionPieces();
        }
```

```
        private ChessPiece[,] SpawnAllPieces(NotationString notationString,
(int boardSizeX, int boardSizeY) chessBoardSize)
        {
            PieceArrangementData data =
notationString.GameDataFromNotationString();
            ChessPiece[,] chessPieces = new
ChessPiece[chessBoardSize.boardSizeX, chessBoardSize.boardSizeY];
            _piecesParent = new GameObject(PiecesParentName).transform;

            for (int x = 0; x < chessBoardSize.boardSizeX; x++)
            for (int y = 0; y < chessBoardSize.boardSizeY; y++)
                if (data.piecesInfo[x, y] != null)
                    chessPieces[x, y] = SpawnSinglePiece(data.piecesInfo[x,
y].type, data.piecesInfo[x, y].team);
            return chessPieces;
        }

        public ChessPiece SpawnSinglePiece(ChessPieceType pieceType, Team
team)
        {
            var prefabIndex = ((int)pieceType - 1) + ((int)team * 6);
            var pieceGO =
Object.Instantiate(_piecesPrefabsSO.piecesPrefabs[prefabIndex],
_piecesParent);

            ChessPiece cp = _chessPieceFactory.CreatePiece(pieceType);

            cp.ChessType = pieceType;
            cp.Team = team;
            cp.PieceTransform = pieceGO.transform;
            cp.Initialize();

            return cp;
        }

        private void PositionPieces()
        {
            for (int x = 0; x < _chessBoardInfoSO.boardSizeX; x++)
```

```
            for (int y = 0; y < _chessBoardInfoSO.boardSizeY; y++)
                if (this[x, y] != null)
                    this[x, y].PositionPiece(x, y, true);
        }
    }
public class ChessGame : IInitializable, IChessGame
    {
        private readonly NotationString _notationString;
        private readonly IBoardFactory _boardFactory;
        private readonly INetworkService _networkService;
        private readonly GameCameraController _gameCameraController;
        private readonly PlayerFactory _playerFactory;
        private readonly TimerFactory _timerFactory;
        private readonly IPopUpService _popUpService;
        private readonly ICloudService _cloudService;
        private readonly IInputSystem _inputSystem;

        public ChessPlayer ActivePlayer { get; private set; }

        public ChessPlayer[] ChessPlayers { get; }

        private ChessPlayer _localPlayer;

        private GameTimer _timer;


        public PieceArrangementData InitialPieceArrangementData { get;
private set; }

        public bool IsActivePlayer => ActivePlayer == _localPlayer;


        public Team WhoseTurn { get; private set; }

        public bool WaitingForUserInput { get; set; }


        public ChessGame(NotationString notationString, IBoardFactory
boardFactory, INetworkService networkService,
```

```
            GameCameraController gameCameraController
            ,   PlayerFactory   playerFactory,   TimerFactory   timerFactory,
IPopUpService popUpService,
            ICloudService cloudService, IInputSystem inputSystem)
        {
            _inputSystem = inputSystem;
            _notationString = notationString;
            _boardFactory = boardFactory;
            _networkService = networkService;
            _gameCameraController = gameCameraController;
            _playerFactory = playerFactory;
            _timerFactory = timerFactory;
            _popUpService = popUpService;
            _cloudService = cloudService;
            ChessPlayers = new ChessPlayer[2];
        }


        public void Initialize()
        {
            InitialPieceArrangementData                                       =
_notationString.GameDataFromNotationString();

            ChessBoard chessBoard = _boardFactory.Create();

            CreatePlayers(chessBoard);
            chessBoard.Initialize(this);


            _timer = _timerFactory.Create();
            _timer.InitializeTimer(ChessPlayers[0],        ChessPlayers[1],
EndOfGame);


            WhoseTurn = (Team) InitialPieceArrangementData.whoseTurn;
            ActivePlayer = ChessPlayers[(int) WhoseTurn];
            _localPlayer = GetLocalPlayer();

            _gameCameraController.Initialize(_localPlayer.Team);
```

```
            SubscribeToLeftGameEvents();

            _timer.Start();
        }

        private void SubscribeToLeftGameEvents()
        {
            _inputSystem.OnBackButtonPressed = () => {
_popUpService.ShowPopUp(GameConstants.PopUps.ExitGamePopUp); };

            _networkService.RoomCallbacksDispatcher.OnPlayerLeftRoomEvent
+= _ =>
            {
                _popUpService.ShowPopUp(GameConstants.PopUps.ErrorToast,
"Player left the room!");
                EndOfGame(_localPlayer.Team);
            };

            _networkService.RoomCallbacksDispatcher.OnLeftRoomEvent += ()
=>
            {
                _inputSystem.OnBackButtonPressed = null;
                _inputSystem.OnCameraViewChanged = null;
            };
        }


        private ChessPlayer GetLocalPlayer()
        {
            int team = _networkService.LocalPlayer.PlayerTeam;
            Logger.DebugError($"Local player is from team {team}");
            return ChessPlayers[team];
        }

        public bool CanPerformMove()
        {
            if (IsActivePlayer && !WaitingForUserInput)
                return true;
```

```
            return false;
        }

        public void ChangeActiveTeam()
        {
            WhoseTurn = WhoseTurn == Team.White ? Team.Black : Team.White;
            ActivePlayer = ChessPlayers[(int) WhoseTurn];
            _timer.SwitchPlayer();
        }

        public bool IsTeamTurnActive(Team team)
        {
            return ActivePlayer.Team == team;
        }

        private void CreatePlayers(ChessBoard chessBoard)
        {
            for (int i = 0; i < 2; i++)
            {
                ChessPlayers[i] = _playerFactory.Create(chessBoard, (Team)
i, InitialPieceArrangementData,
                    _networkService.LocalPlayer.MatchTimeLimitInSeconds);
                ChessPlayers[i].Initialize();
            }
        }

        public void EndOfGame(Team winTeam)
        {
            _timer.Stop();
            Debug.LogError(_localPlayer.Team == winTeam ? "You win" : "Game
over");
            MatchResult result = _localPlayer.Team == winTeam ?
MatchResult.Win : MatchResult.Lose;
            if (result == MatchResult.Win && _cloudService.IsLoggedIn)
            {
                int newPlayerScore =
_networkService.LocalPlayer.PlayerScore + 20;
                _cloudService.UpdateUserData(new Dictionary<string,
string>()
```

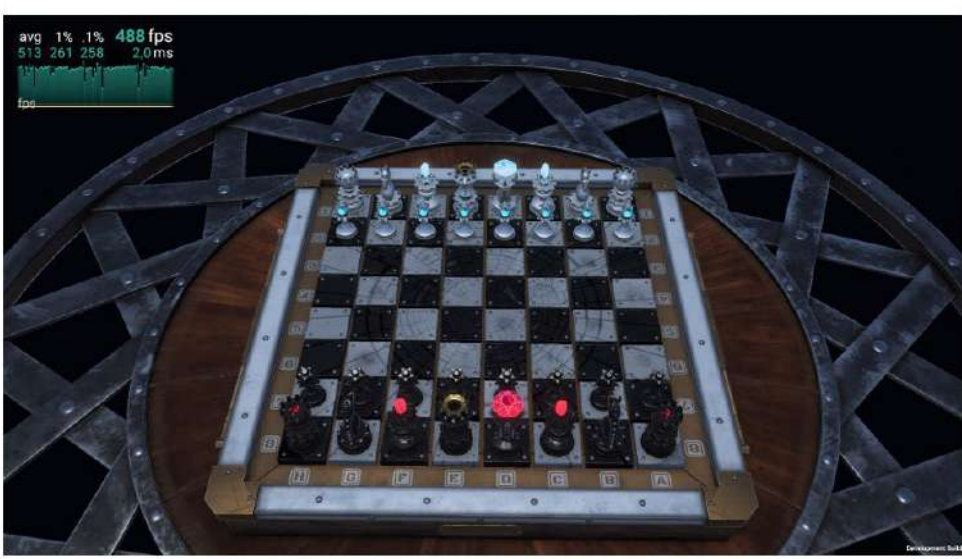
```
                {
                    [GameConstants.PlayerDataKeys.PlayerScoreKey]          =
newPlayerScore.ToString(),
                });
            }
_popUpService.ShowPopUp(GameConstants.PopUps.WinOrLoseWindow, result);
        }

        public PieceArrangementData AssembleCurrentGameData()
        {
            PieceArrangementData      pieceArrangementData      =      new
PieceArrangementData();
            pieceArrangementData.whoseTurn = (int) WhoseTurn;
            pieceArrangementData.canBlackCastleKingSide                  =
ChessPlayers[(int) Team.Black].CanRightSideCastle;
            pieceArrangementData.canBlackCastleQueenSide                 =
ChessPlayers[(int) Team.Black].CanLeftSideCastle;
            pieceArrangementData.canWhiteCastleKingSide                  =
ChessPlayers[(int) Team.White].CanRightSideCastle;
            pieceArrangementData.canWhiteCastleQueenSide                 =
ChessPlayers[(int) Team.White].CanLeftSideCastle;
            return pieceArrangementData;
        }
    }
```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Розробка комп'ютерної системи для гри в шахи у реальному часі

ст.групи КНТ-512сп: Рогач О.В.
к.т.н.,доцент : Скрупський С.Ю.

Проєкт



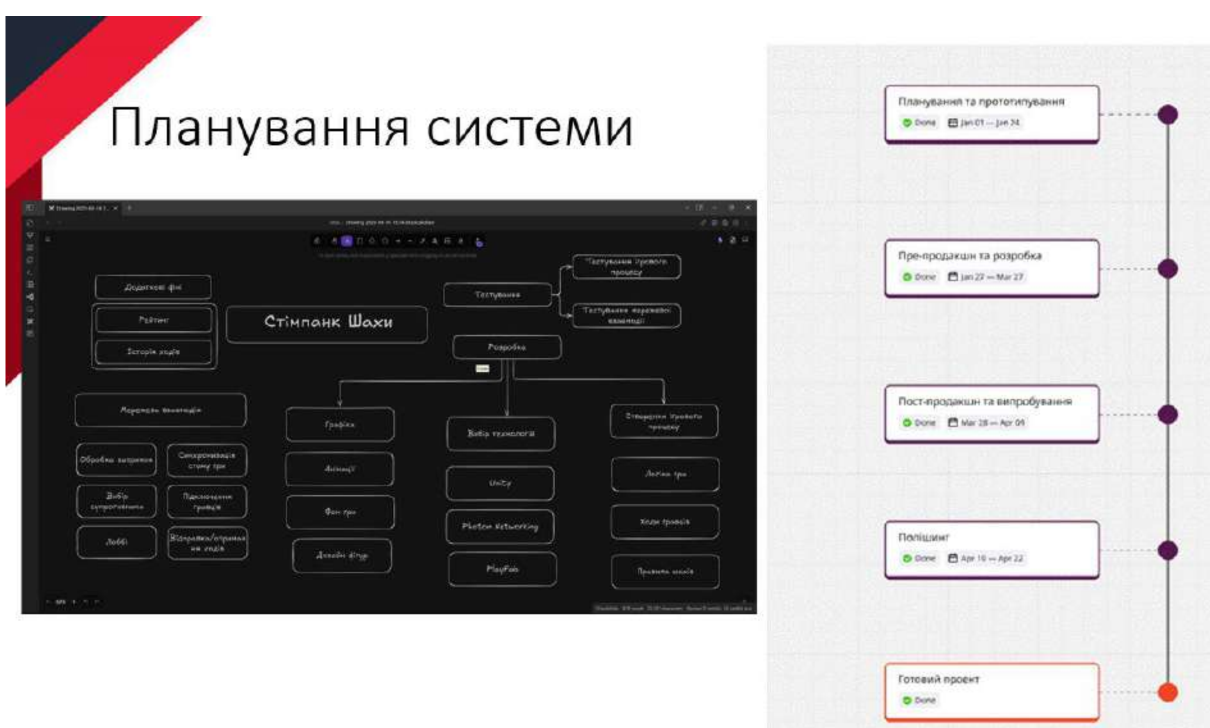
Існуючі аналоги. Порівняльна таблиця

	Steampunk Chess	Lichess	Chess.com	3D Chess Online
Візуальна подача	3D графіка та анімації + стімпанк дизайн + камера	Мінімалізм, 2D	Класичний, 2D	3D, без стилізації
Простота гри	Можливість гри без акаунту, у режимі "Гість"	Можливість гри без акаунту	Можливість гри без акаунту	Потрібна реєстрація
Історія рухів	FEN	FEN, PGN...	FEN, PGN...	Відсутня
Гра з другом	Можлива через лоббі, створена кімната може містити пароль	Окрема фіча "Друзі"	Окрема фіча "Друзі"	Лоббі
Система рейтингу	👍	👍	👍	👎

Системні вимоги

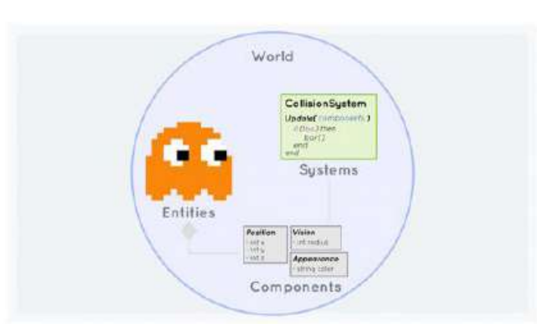
КОМПОНЕНТ	МІНІМАЛЬНІ ВИМОГИ
Процесор	Двоядерний процесор 2.4 ГГц
Оперативна пам'ять	4 ГБ
Операційна система	Windows
Графічний процесор	Radeon HD 7000 або Nvidia GeForce GTX 500
Місце на диску	2 гб
Підключення до Інтернет (тільки при мультиплеєрній грі)	Стабільне підключення, швидкість 2 Мбіт/с

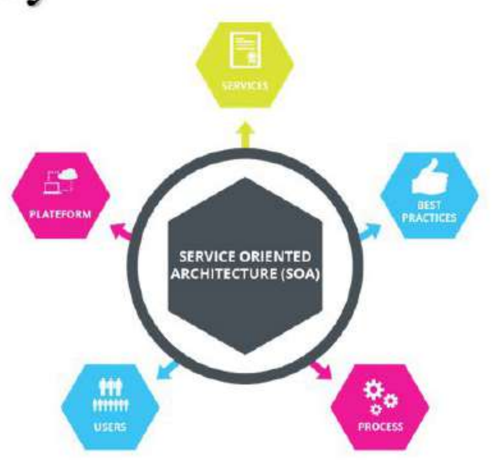
КОМПОНЕНТ	РЕКОМЕНДОВАНІ ВИМОГИ
Процесор	Intel Core i5-4460 або AMD Ryzen 3 1200
Оперативна пам'ять	8 ГБ
Операційна система	Windows
Графічний процесор	Radeon RX 500 або Nvidia GeForce GTX 900
Місце на диску	4 гб
Підключення до Інтернет (тільки при мультиплеєрній грі)	Стабільне підключення, швидкість 2 Мбіт/с



Використані програми

Вибір архітектури проекту



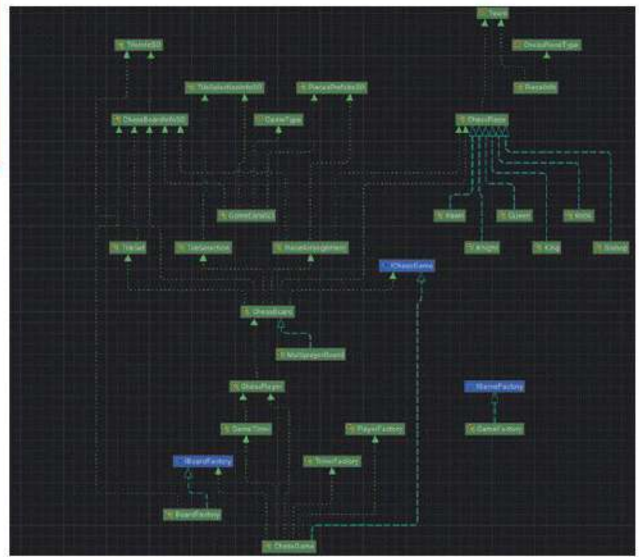


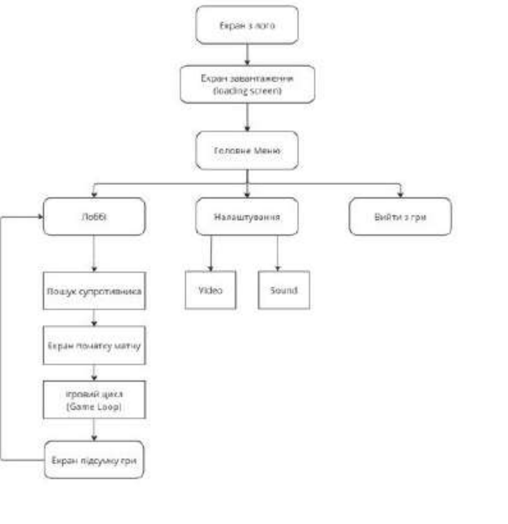
Використані інструменти

	Опис
Zenject	Фреймворк для dependency injection у Unity, що допомагає структурувати код, зменшити зв'язність між об'єктами та полегшити тестування.
Addressabless	Система для асинхронного завантаження та управління ресурсами в Unity, яка дозволяє ефективно розподіляти контент у великі проєкти, зокрема для оновлень і завантаження даних з сервера.
Unity Localization	Офіційний пакет Unity для підтримки багатомовності, який дозволяє перекладати текст, аудіо та інші ресурси для різних регіонів і мов.
UniTask	Легковажна альтернатива Task для Unity, оптимізована для використання з async/await без GC allocations і з кращою продуктивністю в ігрових сценаріях.
Hot Reload	Інструмент, який дозволяє змінювати C# код у реальному часі без перезавантаження сцени чи повного запуску гри, що значно пришвидшує процес розробки.
Cake.Unity	Надбудова над Cake (C# Make), яка дозволяє автоматизувати збірку Unity-проєктів через скрипти, полегшуючи CI/CD інтеграцію та рутинні задачі.

miro

Проектування системи

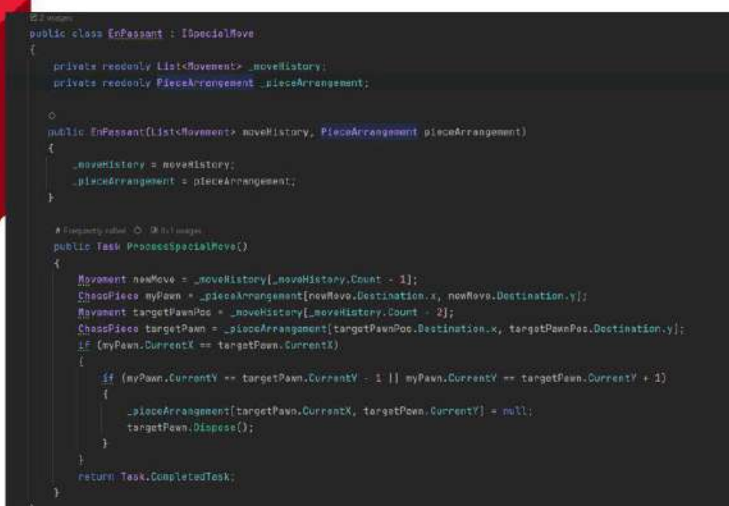


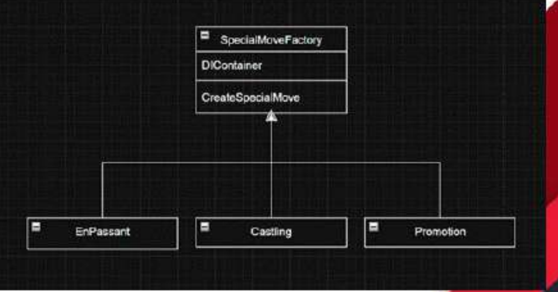


Симуляція руху фігури

```
private void SimulateForSinglePiece(ChessPiece piece, List<Movement> moves, ChessPiece king)
{
    List<Movement> movesToRemove = new List<Movement>();
    Team attackingTeam = piece.Team == Team.White ? Team.Black : Team.White;

    for (int i = 0; i < moves.Count; i++)
    {
        ChessPiece simulationPiece = piece.ShallowCopy();
        int simX = moves[i].Destination.x;
        int simY = moves[i].Destination.y;

        Vector2Int kingPositionThisSim = new Vector2Int(king.CurrentX, king.CurrentY);

        if (piece.ChessType == ChessPieceType.King)
            kingPositionThisSim = new Vector2Int(simX, simY);

        PieceArrangement simulation = _pieceArrangement.DeepCopy();
        var possibleAttackingPieces = new List<ChessPiece>(_chessGame.ChessPlayers[(int) attackingTeam].ActivePieces);

        simulation[simulationPiece.CurrentX, simulationPiece.CurrentY] = null;
        simulationPiece.CurrentX = simX;
        simulationPiece.CurrentY = simY;
        simulation[simX, simY] = simulationPiece;

        List<Movement> simulatedMoveHistory = new List<Movement>(_moveHistory);
        simulatedMoveHistory.Add(new Movement(new Vector2Int(simulationPiece.CurrentX, simulationPiece.CurrentY), new Vector2Int(simX, simY), simulation));

        var deadPiece = possibleAttackingPieces.Find(x => x.CurrentX == simX && x.CurrentY == simY);

        if (deadPiece != null)
            possibleAttackingPieces.Remove(deadPiece);
```

Спеціальний рух "Взяття на проході"



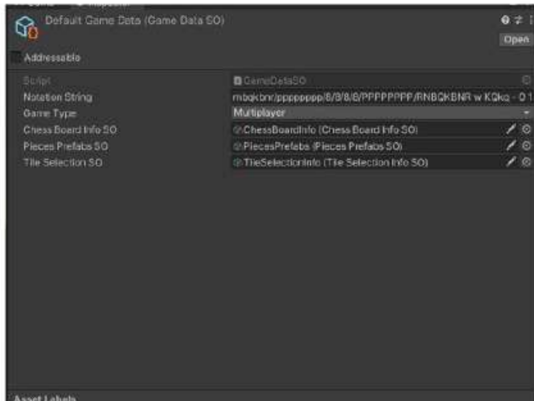


PGN та FEN

Для зберігання шахових позицій використовувався FEN,а для відображення рухів PGN.

Portable Game Notation (PGN) — формат файлу для збереження шахових партій.

Нотація Форсайта — Едвардса (Forsyth–Edwards Notation) — один із видів шахової нотації, яку зазвичай застосовують, щоб описати поточну позицію (наприклад, для текстового подання шахової задачі).



Мережева взаємодія

- Віддалений сервер
- RPC
- PlayFab
- Photon PUN 2

Висновок

- Розроблена комп'ютерна система для гри в шахи у реальному часі.
- Застосовано ігровий рушій Unity у поєднанні з сучасними бібліотеками для реалізації інтерфейсу та мережевого функціоналу;
- Реалізовано основний функціонал системи:
  - створення облікового запису та авторизація користувачів;
  - пошук супротивника та підключення до онлайн-матчу;
  - онлайн матчі у реальному часі;
  - система рейтингу;
  - перегляд шахових партій у вигляді нотацій;