МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет інформаційних технологій

Кафедра інтелектуальних технологій

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

НА ТЕМУ:

«Стратегічна веб-гра жанру 4X

з автоматичною генерацією поля»

Галузь знань **12 «Інформаційні технології»**

Спеціальність **122 «Комп'ютерні науки»**

Освітня програма **«Комп'ютерні науки»**

Освітній рівень: **бакалавр**

Виконала: студентка 4 курсу, групи КН-42
спеціальності – 122 «Комп'ютерні науки»

Козелько Софія Єгорівна
(прізвище та ініціали)

Керівник к.т.н., доц. Кіктєв М.О.
(наук. ступінь, звання, прізвище та ініціали)

Випускна кваліфікаційна робота бакалавра допущена до захисту
рішенням кафедри інтелектуальних технологій
Протокол № 13 від 05.06.2023 р.

зав. кафедри_____ доц. Іларіонов О.Є.

Київ 2023

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет інформаційних технологій

Кафедра інтелектуальних технологій

Спеціальність 122 «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
Зав. кафедри інтелектуальних технологій

к.т.н., доц. Іларіонов О.Є.
(звання, прізвище та ініціали)

(підпис)

« ____ » _____ 2023 р.

З А В Д А Н Н Я
НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ
Козелько Софії Єгорівни

1. Тема роботи: "Стратегічна веб-гри жанру 4X з автоматичною генерацією поля"

затверджена наказом ректора від "11" листопада 2022 року № 4

2. Термін виконання проекту (роботи): 31 травня 2023 року

3. Вихідні дані до роботи: розробити стратегічну гру з можливістю розширення ігрового простору

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):
1) аналіз предметної області: стратегічних ігор жанру 4X;
2) проектування веб-гри жанру 4X;
3) програмна реалізація стратегічної гри жанру 4X.

5. Перелік презентаційного матеріалу (з точним зазначенням обов'язкових презентацій):
Мета, об'єкт та предмет кваліфікаційної роботи (1 слайд), Завдання кваліфікаційної роботи (1 слайд), Особливості жанру 4X (1 слайд), Аналіз історії розвитку ігор жанру 4X (1 слайд), Етапи розробки ігор жанру 4X (1 слайд), Моделювання основних процесів гри жанру 4X (2 слайди), Діаграми послідовності дій (2 слайди), Порядок завантаження ігрового додатку (2 слайд), Робочі вікна розробленого ігрового додатку (1 слайд), Розроблені алгоритми (2 слайди), Висновки (1 слайд).

6. Консультанти з випускної кваліфікаційної роботи із зазначенням її розділів, що їх стосуються

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Кіктєв М.О.		
2	Кіктєв М.О.		
3	Кіктєв М.О.		

7. Дата видачі завдання 15 лютого 2023 року

Керівник випускної кваліфікаційної роботи___ ___ / М.О. Кіктєв /
(підпис) (ініціали та прізвище)

Завдання прийняв до виконання ___ / С.Є. Козелько /
(підпис) (ініціали та прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів випускної кваліфікаційної роботи	Термін виконання етапів	Примітка
1	Обговорення постановки завдання та змісту пояснювальної записки	25.01.2023 -22.02.2023	
2	Вибір та формування теми	23.02.2023 - 27.02.2023	
3	Аналіз предметної області	28.02.2023 – 11.03.2023	
4	Вибір методів рішення задачі	12.03.2023 –20.04.2023	
5	Створення програмного модулю	21.04.2023 -10.05.2023	
6	Оформлення пояснювальної записки	11.05.2023 – 29.05.2023	

Керівник випускної кваліфікаційної роботи ___ ___ / М.О. Кіктєв /
(підпис) (ініціали та прізвище)

Студент ____ / С.Є. Козелько /
(підпис) (ініціали та прізвище)

Анотація

Козелько Софія Єгорівна виконала випускну кваліфікаційну роботу на тему “Стратегічна веб-гра жанру 4X з автоматичною генерацією поля” за спеціальністю 122 – «Комп’ютерні науки».

Дана кваліфікаційна робота присвячена аналізу предметної області розробки стратегічних ігор жанру 4X, моделюванню їх основних процесів та розробці програмної реалізації гри. У першому розділі проведений аналіз літературних джерел за тематикою ігрового дизайну, огляд жанру 4X та його особливостей, історія розвитку та аналіз популярних ігор жанру 4X, тенденції виходу на ринок та основні складності при розробці ігор цього жанру. У другому розділі проведений аналіз сучасних тенденцій та напрямків розвитку веб-ігор жанру 4X, моделювання основних процесів гри жанру 4X, проектування інтерфейсу, побудова діаграм. У третьому розділі розглянуті етапи розробки стратегічної гри жанру 4X, розроблено графічні компоненти гри, описано класи об’єктної моделі гри, розгорнуто ігровий сервер та проведено тестування ігрового додатку. В результаті роботи розроблено функціонуючий ігровий додаток, який відповідає всім вимогам до стратегічних ігор жанру 4X.

Ключові слова: ігровий дизайн, жанр 4X, веб-ігри, моделювання, програмна реалізація.

Abstract

Kozelko Sofiia Yegorivna has completed a graduation thesis on the topic "Strategy web game of the 4X genre with automatic map generation" in the specialty 122 – "Computer Science".

This qualification work is devoted to the analysis of the subject area of development of strategic games of the 4X genre, modeling of their main processes, and development of game software implementation. The first chapter provides an analysis of literary sources on game design, an overview of the 4X genre and its characteristics, history and analysis of popular 4X games, trends in market entry and major challenges in game development. The second chapter analyzes modern trends and directions in the development of web games of the 4X genre, models the main processes of 4X games. The third chapter covers the stages of developing a strategic game of the 4X genre, the development of graphic game components, a description of object model classes, the deployment of a game server, and testing of the game application. As a result of the work, a functional game application was developed that meets all requirements for strategic games of the 4X genre.

Keywords: game design, 4X genre, web games, modeling, software implementation.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ І СКОРОЧЕНЬ

RTS – скорочення від Real-Time Strategy (стратегії в режимі реального часу), що означає жанр ігор, в яких гравець управляє військовими одиницями та веде битви у режимі реального часу.

TBS – скорочення від Turn-Based Strategy (стратегії поштовхового типу), що означає жанр ігор, в яких гравець та його супротивник по черзі роблять свої ходи.

4X – скорочення від Explore, Expand, Exploit, Exterminate (Досліджувати, Розширюватися, Експлуатувати, Знищувати), що означає жанр стратегічних ігор, в яких гравець керує розвитком своєї цивілізації, займає нові території, експлуатує ресурси та веде війну з іншими цивілізаціями.

AI – скорочення від Artificial Intelligence (штучний інтелект), що означає програмний код, що дозволяє ігровим персонажам діяти самостійно і приймати рішення.

NPC – скорочення від Non-Playable Character (нестерпна постать), що означає персонажа, яким керує комп'ютер, а не гравець.

DLC – скорочення від Downloadable Content (завантажуваний контент), що означає додатковий матеріал до гри, який можна завантажити після випуску гри.

MMORPG – скорочення від Massively Multiplayer Online Role-Playing Game (онлайн-гра з багатьма гравцями).

ВСТУП

Актуальність кваліфікаційної роботи на тему "Стратегічна веб-гра жанру 4X" полягає в тому, що в останні роки веб-ігри стали дедалі більш популярними, а саме жанр 4X є одним з найбільш захоплюючих та цікавих для гравців. Тому розробка нових та покращення існуючих стратегічних веб-ігор є актуальним завданням для розвитку індустрії комп'ютерних ігор.

Крім того, веб-ігри жанру 4X є популярними серед гравців у всьому світі, що робить їх привабливими для розробників, які можуть залучати широку аудиторію та отримувати прибуток від продажу внутрішньої валюти та реклами. Розробка нової та покращення існуючих стратегічних веб-ігор жанру 4X є актуальним завданням для розвитку індустрії комп'ютерних ігор та задоволення потреб гравців.

Одним з критичних питань, які можуть викликати суперечки серед розробників, є баланс геймплею [1]. Жанр 4X включає в себе багато елементів гри, таких як війна, дипломатія, ресурси та інше, що потребує від розробників збалансованості геймплею та механік гри. Недостатній баланс може спричинити незадоволення серед гравців, які можуть відмовитися від гри. З іншого боку, занадто складний баланс може зменшити виклик та цікавість гри.

Іншим критичним питанням може бути оптимізація гри під різні пристрої та браузери. Веб-ігри мають бути доступні для гравців з будь-якого пристрою та браузера, що може вимагати зусиль від розробників для підтримки різних платформ. Якщо гра не працює на певних пристроях або браузерах, це може зменшити аудиторію гри.

Також, ще одним можливим критичним питанням є монетизація гри. Ігри жанру 4X зазвичай мають глибокий геймплей та довгі часові проміжки, що може зменшити готовність гравців до витрат на гру. Розробники повинні знайти баланс між довжиною гри та монетизацією, щоб забезпечити рентабельність проекту.

Крім того, інтернет-зв'язок є необхідним для гри в веб-ігри, що може створювати проблеми для гравців, які мають поганий зв'язок з Інтернетом [2].

Розробники повинні забезпечувати якісний та стабільний інтернет-зв'язок, щоб гравці могли насолоджуватися грою без перерв та завад.

Ігровий дизайн є однією з ключових областей індустрії цифрових розваг [3]. З появою нових технологій виробничі інструменти значно розвинулися, але мало було зроблено для підтримки дизайну ігор. Дизайнери все ще використовують ті самі інструменти, що існували на початку цієї галузі, і, хоча галузь поспіль досягла успіху в продажах, дослідники та професіонали сходяться на думці, що відсутність інструментів, концептуальних чи програмних, створює перешкоду для будь-яких спроб стандартизації та заважає передача знань між поколіннями дизайнерів. Вони запропонували концептуальні та конкретні інструменти, які могли б доповнити або замінити проектний документ, щоб покращити процес створення ігор [5].

Мета кваліфікаційної роботи – розробка та тестування стратегічної веб-гри жанру 4X, а також проведення аналізу характеристик та механік гри з метою забезпечення її балансу та підвищення цікавості геймплею.

Об'єкт дослідження – стратегічна веб-гра жанру 4X.

Предмет дослідження – процес розробки, тестування та аналізу характеристик стратегічної веб-гри жанру 4X.

Для досягнення мети використовується ReactJs, PixiJs для анімації та Firebase для роботи на багатьох девайсах. Середовище розробки Visual Studio Code.

Таким чином, кваліфікаційна робота має значення як для індустрії комп'ютерних ігор, так і для наукового дослідження в галузі інформаційних технологій та інтернет-маркетингу. Результати дослідження можуть бути використані для розвитку інших веб-ігор жанру 4X, а також для побудови маркетингових стратегій на ринку комп'ютерних ігор.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ РОЗРОБКИ СТРАТЕГІЧНИХ ІГОР ЖАНРУ 4X

1.1 Аналіз проблематики дослідження і літературних джерел за темою ігрового дизайну

Предметна область створення ігор має різнобічні напрями і в кожному з них проводяться власні наукові дослідження.

Можна виділити п’ять основних напрямів, які постійно досліджуються не тільки на рівні ігрових студії, а і в наукових колах:

– проєктування та розробка гри жанру 4X: розробка гри цього жанру вимагає детального проєктування геймплею, механік гри, інтерфейсу користувача та ігрового середовища. Для досягнення високої якості гри, потрібно ретельно продумати баланс геймплею, розвиток персонажів, стратегічні елементи, генерацію світу та інші аспекти;

– управління ресурсами: ігри жанру 4X мають складну систему управління ресурсами, такими як золото, населення, наука, культура тощо. Одна з основних проблем полягає в балансуванні та оптимізації цих ресурсів.

– використання штучного інтелекту: ігри жанру 4X мають багатошарову систему штучного інтелекту, яка контролює поведінку комп'ютерних гравців. Розробка ефективного інтелекту, який здатен приймати рішення, прогнозувати стратегію та реагувати на дії гравця, може бути викликом;

– балансування геймплею: одна з важливих проблем полягає в балансуванні геймплею, щоб забезпечити рівновагу між різними аспектами гри, такими як розвиток, бойова система, економіка тощо. Недостатній баланс може призвести до несправедливості або нудоти в геймплеї.

– графіка та анімація: ігри жанру 4X часто мають велику кількість об'єктів, персонажів та ігрових елементів, які потребують якісної графіки та анімації. Розробка інтерактивних та привабливих візуальних ефектів може бути

складною та часомісткою задачею.

Ці проблеми вимагають від розробників гри ретельного планування, дизайну, реалізації та тестування, а також постійного вдосконалення та оновлення гри для забезпечення якісного геймплею та задоволення гравців.

Незважаючи на те, що ігровий дизайн широко прийнятий як основний інструмент у галузі, у проектній документації існує значний брак формалізації та стандартизації. Незважаючи на очевидний успіх галузі, дослідники та професіонали погоджуються, що брак інструментів, концептуальних чи програмних, перешкоджає передачі знань між поколіннями дизайнерів та розвитку самого процесу проектування [1].

Протягом багатьох років практики та дослідники визначили потребу у формальних моделях та інструментах для підтримки дизайну ігор [2]. У цьому контексті «формальний» стосується не математичних моделей, а організованих, стандартизованих і структурованих моделей і інструментів, які допомагають методам розробки ігор.

Як техніка проектування, шаблони дизайну ігор представляють собою еволюцію в порівнянні з FADT і найбільш значний внесок у спробі створити базу даних концепцій дизайну. Його структура, натхненна шаблонами дизайну, дозволяє підкреслити відповідні аспекти кожного візерунка та їхні взаємозв’язки. З іншого боку, проект має недоліки, які перешкоджають його використанню. Немає достатньої відповідності між шаблонами та іграми або жанрами, які їх використовують, щоб забезпечити аналіз, орієнтований на ігри.

Леблан [8] підкреслював важливість розгляду точки зору як дизайнера (виробника), так і гравця (споживача) при розробці ігор. Робота пропонує структуру для аналізу ігор у трьох різних компонентах: механіка, яка описує статичні правила ігрової системи, описані в проектному документі; динаміки, які представляють поведінку механіки під час виконання ігрового сеансу, присутні в іграх і прототипах, а також; естетика, яка охоплює бажані емоційні реакції гравців, отримані в результаті випробувань і експериментів з прототипом або грою.

Деякі автори припускають, що не всі ігри можуть бути оформлені в ієрархічну систему категорій і підкатегорій [9]. Серед них запропонував ортогональну таксономію для ігор, яка визначає просторову систему характеристик, де ігри та жанри позиціонуються відповідно до їхньої близькості до цих характеристик. Як приклад, автор представляє простори одного, двох чи трьох вимірів із застосуванням різних характеристик, таких як симуляція. Результати досліджень показали, які елементи є бажаними для конкретного ігрового проекту, що відповідає класифікації. Подальші дослідження могли б пов'язати податкову економіку в іграх і колекції концепцій дизайну, щоб дослідити, які кількісні та якісні відносини можна встановити між елементами, ринком і перевагами користувачів.

1.2 Принципи побудови ігор жанру 4X

1.2.1 Визначення жанру ігор 4X

Жанр 4X (англійською 4X, від англ. eXplore, eXpand, eXploit, eXterminate) – це піджанр стратегічних ігор, який включає в себе групу ігор, де гравці відтворюють роль лідера цивілізації, яка розвивається від початку історії до сучасності.

Основні особливості жанру 4X:

1. Експлорація: гравець досліджує світ гри, збирає ресурси, досліджує нові технології та території.

2. Розширення: гравець займає нові території, розбудовує нові міста та володіння, збільшує свої можливості та ресурси.

3. Експлуатація: гравець використовує свої ресурси для побудови нових будівель, створення армії, дослідження нових технологій та забезпечення потреб своїх громадян.

4. Екстермінація: гравець змагається з іншими цивілізаціями за гегемонію на світовій карті, використовуючи свою армію та технології для перемоги над

ворогами.

У жанрі 4X гравець зазвичай має можливість вибирати, яким чином розвивати свою цивілізацію, які технології досліджувати, як розбудовувати свої міста та володіння, які союзники шукати та з ким воювати. Головна мета гравця полягає в досягненні світової домінанти, але шлях до неї може бути різним в залежності від виборів, які зробив гравець.

Однією з головних переваг жанру 4X є його відкритий світ, який дозволяє гравцям відчути себе лідерами своєї цивілізації та відчути незалежність у вирішенні питань розвитку. Крім того, жанр 4X дозволяє гравцям розвиватись та досягати своїх цілей у своєму темпі, що робить його доступним для гравців з різним рівнем досвіду.

Однак, жанр 4X має свої виклики та обмеження. Однією з головних складнощів є час, який потрібно вкладати в гру. Жанр 4X вимагає від гравця великої кількості часу та терпіння, щоб зробити кожен крок у своєму розвитку. Крім того, жанр 4X може бути складним для новачків, оскільки він має велику кількість функцій та можливостей, які можуть здатися заплутаними та незрозумілими.

Ще однією проблемою є баланс гри. У жанрі 4X може бути складно забезпечити баланс між різними цивілізаціями, технологіями та ресурсами. Це може призвести до того, що деякі цивілізації стануть переважаючими та домінуючими на карті.

Огляд жанру 4X показує, що це складний та цікавий піджанр стратегічних ігор, який вимагає від гравців розуміння та планування, але в той же час дозволяє відчути себе лідером та досягати своїх цілей у своєму темпі.

1.2.1 Сучасні тенденції розвитку ігор жанру 4X

Жанр 4X з'явився в 90-х роках XX століття і відразу ж здобув популярність серед гравців та розробників ігор. Перші ігри цього жанру, такі як Sid Meier's Civilization (1991) та Master of Orion (1993), відкрили нові

можливості для стратегічних ігор та змусили гравців звернути увагу на жанр 4X [14, 15, 16].

Sid Meier's Civilization став однією з перших відомих ігор в жанрі 4X та забезпечив популярність жанру наступних десятиліть [16]. Головною метою гравця в цій грі було розвиток своєї цивілізації від початку історії до сучасності. Гравці повинні були виконувати різні завдання, такі як забезпечення потреб своїх громадян, дослідження нових технологій, дипломатичні відносини з іншими цивілізаціями та боротьба за контроль над світовою картою.

Master of Orion – це ще одна відома гра у жанрі 4X, яка вийшла в 1993 році. У грі гравці брали на себе роль керівника цивілізації на просторах космосу. Головна мета гравця полягала в досягненні світової домінанти в космосі шляхом дослідження нових технологій, розвитку власного флоту та побудови нових планет [17].

За останні 30 років жанр 4X пройшов довгий шлях та зазнав багатьох змін. Сьогодні існують сотні ігор у цьому жанрі, які варіюються від класичних героїчних фентезі-світів до науково-фантастичних ігор про космічну експансію. До найвідоміших ігор у жанрі 4X також входять Civilization VI, Endless Legend, Stellaris, Age of Wonders III, Galactic Civilizations III та багато інших.

Нинішні ігри у жанрі 4X [18-21] дозволяють гравцям відтворити роль керівника своєї цивілізації та відчути себе лідерами на світовій карті. Головні завдання гравців полягають у дослідженні світу гри, збиранні ресурсів, розбудові міст та володінь, створенні армії та веденні війни з іншими цивілізаціями. Головна мета полягає в досягненні світової домінанти та забезпеченні успіху своєї цивілізації.

Однією з головних особливостей сучасних ігор у жанрі 4X є розширення можливостей гравців. Сьогодні гравці можуть створювати свої власні міста та володіння, використовувати різні ресурси, налагоджувати дипломатичні відносини з іншими цивілізаціями та боротьбу з ними. Багато ігор дозволяють гравцям вибирати свій стиль гри, вибирати свої ресурси та технології

дослідження, а також розвивати свої міста та володіння відповідно до власних уявлень та стратегій.

Огляд жанру 4X показує, що цей жанр не стоїть на місці і продовжує розвиватися та еволюціонувати. Нові можливості та технології дозволяють розробникам створювати більш складні та захоплюючі ігри, які дозволяють гравцям відчувати себе справжніми лідерами своєї цивілізації.

На основі аналізу Steam сформовано список усіх ігор з тегом 4X strategy з 1990 до 2021 року [22]. Графік на рис. 1.1 відображають тренди та порядки, але можуть суттєво відрізнятись від реальності, особливо в рамках одного конкретного продукту (не за всіма іграми можна знайти публічну статистику продажів).

При аналізі кількості ігор, що виходять на рік, бачимо цікаві тренди. До 2006-го кількість релізів щороку була приблизно та сама. Далі починається різке зростання. Він викликаний як зростаючим інтересом гравців до жанру, так і комерційними успіхами серії Civilization.

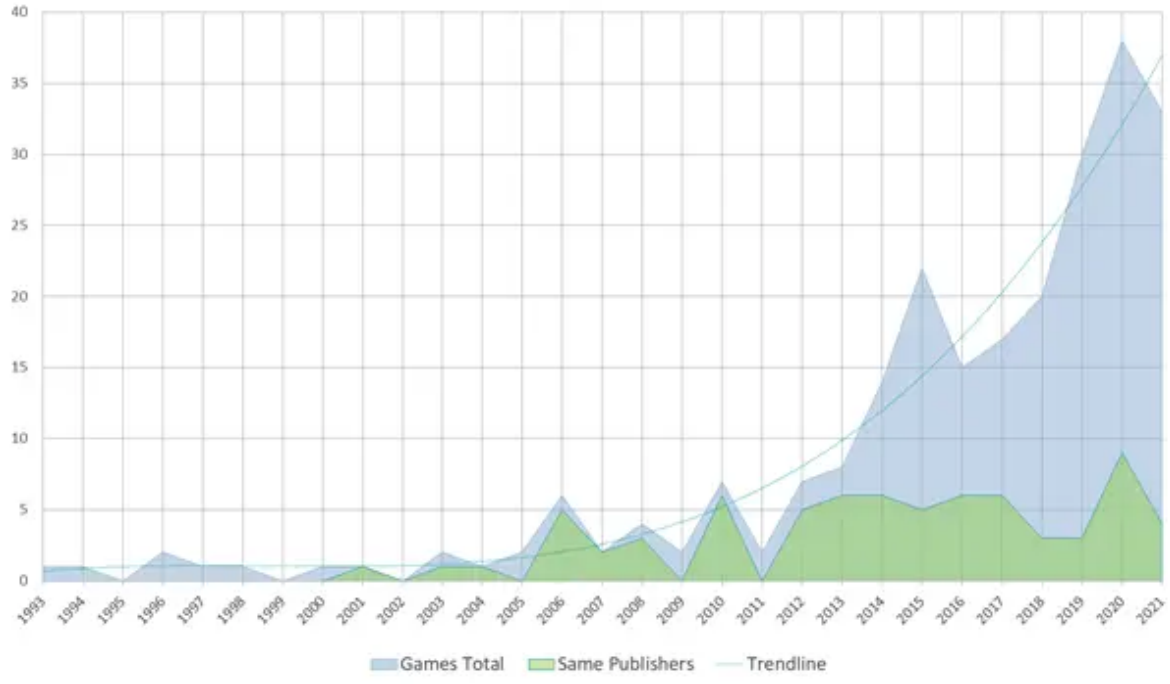

Рисунок 1.1 – Кількість 4X стратегій на рік [22]

Для оцінки ринку ігор було проведено порівняльний аналіз самих популярних ігор за останні 20 років в жанрі 4X, щоб з’ясувати їх сильні і слабкі сторони і визначити чи впливали вони на успішність ігор.

Побудовано порівняльну таблицю (табл. 1.1), де в стовбчиках назви ігор, а в рядках + чи - навпроти параметра. Список параметрів:

1) зручний інтерфейс;

2) 3d графіка;

3) потужний штучний інтелект;

4) можливість швидких битв;

5) браузерна версія;

6) мобільна версія;

7) високі вимоги до комп'ютер;

8) розмір світу;

9) глибина гри;

10) гнучкість налаштувань;

11) модифікації;

12) історична атмосфера;

13) музика та звук;

14) підтримка та оновлення.

В порівняльному аналізі прийняли наступні ігри:

1. Civilization I;

2. Master of Orion;

3. Alpha Centauri;

4. Master of Magic;

5. Sid Meier's Alpha Centauri;

6. Galactic Civilizations;

7. Age of Wonders;

8. Endless Legend;

9. Stellaris;

10. Civilization VI;

11. Endless Space 2.

В табл. 1.1 назви ігор замінено на номери, щоб зменшити розміри таблиці.

Таблиця 1.1

Порівняльна таблиця популярних ігор жанру 4X

Параметр	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Зручний інтерфейс	+	-	+	-	+	+	+	+	+	+	+
3D графіка	-	-	+	-	+	+	+	+	+	+	+
Потужний штучний інтелект	-	-	+	-	+	+	-	-	+	+	+
Можливість швидких битв	+	+	+	+	+	+	+	+	+	+	+
Браузерна версія	-	-	-	-	-	-	-	-	-	+	+
Мобільна версія	-	-	-	-	-	-	-	-	-	+	+
Високі вимоги до комп'ютера	-	-	-	-	-	-	-	-	-	+	+
Розмір світу	Medium	Medium	Large	Small	Large	Large	Medium	Large	Large	Large	Large
Глибина гри	High	High	High	High	High	High	High	High	High	High	High
Гнучкість налаштувань	Medium	Medium	High	Medium	High	High	High	High	High	High	High
Модифікації	-	-	-	-	-	-	-	-	+	+	-
Історична атмосфера	+	-	-	-	-	-	+	+	-	-	-
Музика та звук	+	-	+	-	+	+	+	+	+	+	+
Підтримка та оновлення	+	-	-	-	+	+	+	+	+	+	+

На основі порівняльної таблиці можна зробити такі висновки:

1. Зручний інтерфейс: Більшість ігор мають зручний інтерфейс, окрім Master of Orion та Master of Magic.

2. 3D графіка: Більшість ігор мають 3D графіку, за винятком Civilization I та Master of Orion.

3. Потужний штучний інтелект: Більшість ігор мають потужний штучний інтелект, за винятком Age of Wonders та Endless Legend.

4. Можливість швидких битв: Усі ігри мають можливість швидких битв.

5. Браузерна версія: Тільки Civilization VI та Endless Space 2 мають

браузерну версію.

6. Мобільна версія: Тільки Civilization VI та Endless Space 2 мають мобільну версію.

7. Високі вимоги до комп'ютера: Тільки Civilization VI та Endless Space 2 мають високі вимоги до комп'ютера.

8. Розмір світу: Більшість ігор мають великий розмір світу.

9. Глибина гри: Усі ігри мають високу глибину гри.

10. Гнучкість налаштувань: Усі ігри мають високу гнучкість налаштувань.

11. Модифікації: Тільки Civilization VI та Stellaris мають підтримку модифікацій.

12. Історична атмосфера: Тільки Civilization I та Age of Wonders мають історичну атмосферу.

13. Музика та звук: Більшість ігор мають хорошу музику та звукове супроводження.

14. Підтримка та оновлення: Більшість ігор мають підтримку та оновлення, за винятком Master of Orion, Master of Magic та Alpha Centauri.

1.2.2 Основні складності при розробці ігор жанру 4X

Розробка ігор у жанрі 4X є складним та трудомістким процесом. Основні складності, які можуть виникнути при розробці ігор цього жанру, включають наступне:

1. Баланс гри: Один з найскладніших аспектів розробки ігор у жанрі 4X полягає в забезпеченні балансу гри. Розробники повинні враховувати взаємодію різних аспектів гри, таких як економіка, військова стратегія, дипломатія, дослідження та технології, щоб забезпечити рівні умови для всіх цивілізацій, які з'являються в грі.

2. Розробка штучного інтелекту: У грі у жанрі 4X, штучний інтелект (ШІ) повинен бути достатньо розвиненим, щоб забезпечити гравцям виклики та

створити різноманітні ситуації в грі. Розробники повинні враховувати різноманітність стратегій та тактик, які можуть використовувати ШІ, щоб забезпечити складну та захоплюючу гру.

3. Геймплей: Ігри у жанрі 4X повинні мати цікавий та захоплюючий геймплей, який пропонує гравцям різні завдання та виклики. Розробники повинні враховувати велику кількість можливостей та варіацій у грі, які дозволяють гравцям відчувати себе справжніми лідерами своєї цивілізації.

4. Графіка та звук: Важливим аспектом гри у жанрі 4X є якість графіки та звуку. Розробники повинні забезпечити детальну та реалістичну графіку, яка дозволяє гравцям відчути себе частинкою цивілізації та створити атмосферу гри. Звуковий дизайн також має велике значення, оскільки він допомагає створити настрій та відчуття у гравця.

5. Технічні складності: Розробка ігор у жанрі 4X може зіткнутися з технічними складностями, пов'язаними з обробкою великих обсягів даних та графіки, збереженням гри та іншими технічними аспектами. Розробники повинні забезпечити оптимальну продуктивність гри, щоб гравці могли насолоджуватися грою без затримок та зависань.

Розробка ігор у жанрі 4X є складним та трудомістким процесом. Розробники повинні уважно вивчати ринок, аналізувати попередні ігри цього жанру та добре розуміти потреби гравців, щоб створити популярну гру.

1.2.3 Огляд базових алгоритмів при розробці ігор жанру 4X

При розробці гри жанру 4X можуть знадобитися наступні математичні алгоритми:

1. Алгоритми генерації світу: для створення гри з випадковою генерацією світу можуть використовуватися алгоритми, такі як алгоритм Diamond-Square або Perlin Noise для створення рельєфу і ландшафту.

2. Алгоритм пошуку шляху: для переміщення одиниць по світу гри можна використовувати алгоритми пошуку шляху, наприклад, алгоритм A* або

Dijkstra, які дозволяють знаходити найкоротший шлях між двома точками. Алгоритм A* ефективно знаходить найкоротший шлях від початкової точки до пункту призначення, враховуючи як вартість досягнення вузла.

3. Алгоритми розміщення ресурсів: для розміщення різних ресурсів (наприклад, руди, дерева, води) на карті гри можуть використовуватися алгоритми, такі як алгоритм Вороного або алгоритм Пуанкаре.

4. Алгоритми штучного інтелекту: для програмування поведінки комп'ютерних противників можуть використовуватися алгоритми штучного інтелекту, такі як алгоритми машинного навчання (наприклад, нейронні мережі або генетичні алгоритми) або алгоритми прийняття рішень, що базуються на правилах.

5. Алгоритми економічної моделі: для моделювання економічних процесів в грі можуть використовуватися алгоритми, такі як алгоритм розподілу ресурсів або алгоритм визначення цін на ринку.

6. Алгоритми бойової системи: для симуляції битв і бойових взаємодій між одиницями можуть використовуватися алгоритми, такі як алгоритм розрахунку пошкоджень, алгоритм визначення успішності атаки та алгоритм прийняття рішень для стратегічного планування в бою.

1.3 Концепція розроблюваної гри, вимоги до ігрового додатку і завдання до випускної кваліфікаційної роботи

Вимоги до ігрового додатку можна формувати тільки після визначення основних правил гри чи хоча б напряму розвитку сценарію. Саме для цього було проведено аналіз основних тенденцій ігор і вибору самих популярних.

З урахуванням популярності ігор формату Цивілізація прийнято за основу власної гри жанру 4X взяти базові правила цієї гри, які визначаються за напрямами:

– режими гри (мультіплей, реальний час або покрокова, 2D або 3D);

– будова ігрового світу;

– економіка;

– бойові дії;

– взаємодія з іншими учасниками гри.

Всі ігри жанру 4X мають єдину мету – експансія (захоплення всього ігрового простору), хоча в різних модифікаціях ігор допускаються режими експансії через наукові відкриття, але сутність експансії від цього не змінюється.

Сформуємо основний концепт гри у вигляді кроків і розкриття кожного етапу гри.

1. Гра починається з запуску ігрового лобі:

– підключення до сервера гри після її запуску;

– приєднання до доступного лобі (якщо вже є створена гра чи збережена гра), використовується кнопка "Join".

– якщо немає збережених даних або лобі пусте, то необхідно створити нову гру;

2. Етап гри:

– взаємодія з грою відбувається через виконання різних дій на ігровому полі;

– кожен елемент поля має свою властивість (не відходячи від загального жанру – пусте поле, поле ресурсу, поле з будівлею), щоб переглянути інформацію про кількість юнітів, тип поля або будівлі необхідно його вибрати;

– взаємодія з юнітами: натискання на поле з юнітом для виконання дій, таких як переміщення, припинення переміщення або перегрупування, вказуючи на нове місце, клікнувши на поле, на яке потрібно їх перемістити (в даному випадку гра автоматично побудує шлях переміщення);

– можливість перегрупування юнітів, розділяючи або об'єднуючи їх у різні групи;

– взаємодія з будівлями: на полях, які є власністю гравця можна будувати будівлі (клавіша "Construct" у меню вибору будівель);

– кожна будівля має властивості: що надає, скільки будується, скільки

коштує;

– для спрощення базового геймплею будівництва всіх будівель займатимуть три ходи для завершення;

– після побудови, можливість взаємодіяти з будівлею та виконувати різні дії в ній (виробництво нових юнітів);

3) Економіка:

– золото є ресурсом, який використовується для підтримки юнітів та будівель;

– кожен юніт споживає визначену кількість золотих на хід (для спрощення геймплею будемо брати 1 золото за хід).

– золото можна добувати через будівлі, кожна з яких має свою швидкість виробництва (з огляду на аналіз ігор прийнято наступні базові налаштування (які з часом розвитку гри можуть коригуватись для балансування розвитку): основний замок приносить 4 золота за хід, хижина приносить 2 золота за хід);

– кількість золота та кількість зроблених ходів повинна бути постійно перед очима гравця (наприклад, відображення у лівому верхньому кутку екрана);

4) Битви:

– якщо юніти протилежних команд зустрічаються на полі, запускається режим битви (є ігри де цей режим переходить в формат реального часу, але даний елемент є окремим в будові гри і може бети замінено);

– битва проводиться на окремому полі битви (передбачається формат автоматичного бою);

5) Взаємодія з іншими гравцями:

– в грі передбачено використання боту, який може виконувати прості дії за правилами;

– з іншими противниками в іграх 4X вести різні відносини, але для спрощення формат дипломатії краще впроваджувати вже на нових етапах розвитку гри.

З огляду на проведений аналіз популярних ігор в жанрі 4X та

особливостей розробки ігор жанру 4X визначено наступні загальні функціональні та нефункціональні вимоги до ігрових додатків даного класу:

А) Функціональні вимоги:

1. Керування територіями: гравці повинні мати можливість розширювати свої території, здобувати нові землі та управляти ними. Це включає заселення територій, розбудову і вдосконалення будівель, а також управління ресурсами.

2. Економічний менеджмент: гравці повинні мати можливість управляти своєю економікою, займатися добуванням золота, створювати та управляти фінансами для створення нових юнітів.

3. Ігрова стратегія: гравці повинні мати можливість вести битви з іншими арміями, планувати проведення битв, використовувати юнітів, стратегічні ходи та тактику для досягнення своїх цілей.

Б) Нефункціональні вимоги:

1. Продуктивність та оптимізація: гра повинна бути оптимізована для різних пристроїв та операційних систем, забезпечуючи плавну роботу та швидку відгуки на дії гравців.

2. Візуальна привабливість: гра повинна мати графіку, яка забезпечує візуальні ефекти, анімацію та інші візуальні складові, які роблять геймплей кращим для гравців.

3. Збереження та завантаження: гра повинна підтримувати можливість збереження та завантаження прогресу гри, щоб гравці могли продовжити гру з моменту, на якому вони зупинилися.

Враховуючи ці функціональні та нефункціональні вимоги до ігрового додатку можна сформувати завдання до кваліфікаційної роботи:

1. Описати характеристики та особливості жанру 4X та веб-ігор загалом;

2. Розробити концепцію гри та дизайн геймплею з врахуванням потреб від функціоналу гри та можливостей технічної реалізації;

3. Розробити програмний код гри та забезпечити його функціональність та стабільну роботу;

4. Провести тестування гри;

5. Проаналізувати механіки гри та забезпечити баланс геймплею для підвищення цікавості гри.

6. Сформулювати висновки та рекомендації щодо подальшого розвитку гри.

1.4 Висновки до першого розділу

Проведене в розділі дослідження показало стале становлення геймдизайну та розкрило принципи розробки ігор, що відносяться до 4X стратегій.

Аналіз 4X стратегій показав, що цей жанр є дуже стабільним і розвивається вже багато десятиліть і не зараз виходить на пік. У цьому сегменті є продукти для всіх гравців: від глибокої і складної в освоєнні пісочниці Stelleris до надпотужної графіки Endless Space II. Від всесвітньо відомої Civilization до Age of Wonders, що втрачає популярність.

Сформована концепція гри та проведений аналіз інших ігор жанру 4X дозволив визначити функціональні та нефункціональні вимоги до ігрового додатку і завдання до випускової кваліфікаційної роботи.

РОЗДІЛ 2 МОДЕЛЮВАННЯ РОБОТИ ВЕБ-ІГОР ЖАНРУ 4X

2.1 Моделювання основних процесів гри жанру 4X

2.1.1 Формування дерева функцій гри жанру 4X

Описання повного дерева функцій передбачає більше 100 елементів керування, тому окремі модулі можуть включати в себе десятки підмодулів. На основі прототипів однакові дії не дублювались для кожного класу об’єктів (дерево функцій гри жанру 4X представлено на рис. 2.1):

1. Функції запуску гри

1.1. Підключення до сервера і генерація сторінок:

– підкючення до БД;

– обробка бекенду;

– обробка фронтенду;

1.2. Запуск лоббі:

– сканування відкритих ігор;

– підключення до гри;

– створення гри;

2. Функції ігрового процесу:

2.1. Переміщення:

– вибір юніта;

– вибір нової точки;

– розрахунок маршруту;

– запуск автоматичного оновлення переміщення за кроками;

– напад на противника;

– сканування поля для пошуку противників;

2.2. Функції бою:

– розрахунок ураження;

Рисунок 2.1 – Дерево функцій гри жанру 4X

– дія у разі перемоги;

– дія у разі поразки;

2.3. Функції економіки:

– побудова будівель;

– генерація юнітів;

– накопичення ресурсів за хід;

3. Функції відображення в графічному інтерфейсі:

3.1. Візуалізація компонентів:

– візуалізація ігрового поля;

– візуалізація юнітів та будівель;

– візуалізація властивостей юнітів та будівель;

– візуалізація вибору юніта або будівлі;

– візуалізація переміщення;

– управління іформаційними вікнами;

4. Функції бота:

4.1 Розрахунок ходів:

– сканування поля для пошуку противників;

– оцінка армій;

– напад на противника.

2.1.2 Розробка контекстних діаграм гри жанру 4X

У розробці веб-ігор жанру 4X важливо враховувати ці вимоги та процеси, щоб створити гру, яка буде привабливою та цікавою для гравців. Розробка веб-ігор жанру 4X вимагає великої уваги до деталей та готовності до постійного вдосконалення, щоб забезпечити гравцям високу якість гри та найкращий досвід.

Контекстна діаграма "ЯК БУДЕ" для стратегічної гри 4X в нотації IDEF0 передбачає описання гри жанру 4X і зображена на рис. 2.2.

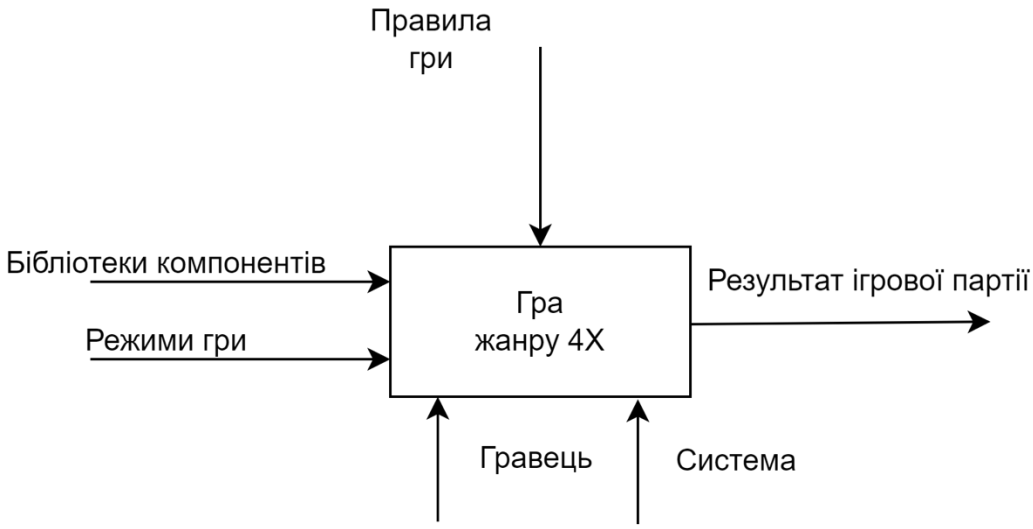

Рисунок 2.2 – Контекстна діаграма ЯК БУДЕ гри жанру 4X

Вхідні дані для проведення гри включають бібліотеки ігрових компонентів режими гри, правила гри. Результатом проведення гри є результат ігрової партії. Управління процесом розробки включає дотримання правил гри. Механізмом гри використовуються система (ігровий додаток) та гравець.

Вихідні дані можуть включати результат партії, тобто стан гри після її завершення. Це може бути повідомлення про перемогу або поразку. Вихідні дані можуть бути використані як для оцінки результатів гри, так і для подальшого аналізу та вдосконалення геймплею.

Для реалізації процесу проведення ігрової партії в залежності від її тривалості, можна виділити чотири етапи: завантаження модулів ігрового процесу, завантаження моделей компонентів, проведення ігрової партії та підведення підсумків партії. Кожен етап має свої вхідні дані, вихідні дані, управління та механізми.

На рис. 2.3 можна побачити контекстну діаграму, яка представлена чотирма процесами, що утворились після декомпозиції процесу проведення ігрової партії. Для подальшої декомпозиції необхідно враховувати, що результат попереднього процесу є вхідними даними для наступного.

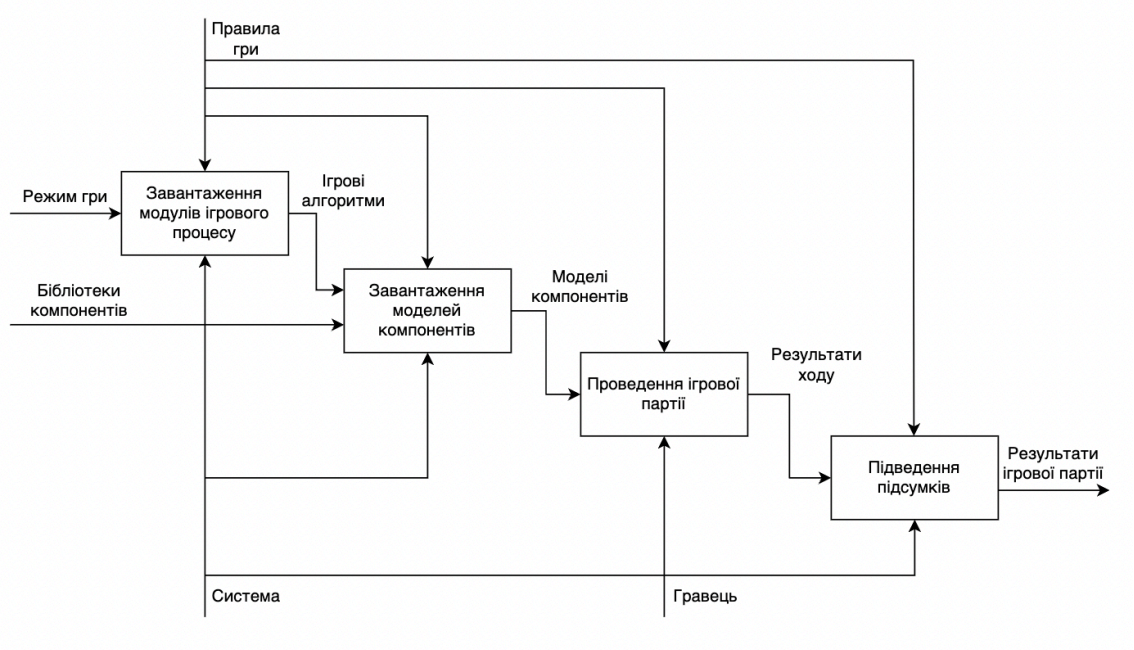

Рисунок 2.3 – Декомпозиція гри в жанрі 4X у форматі IDEF0

Було проведено декомпозицію процесів:

– «Завантаження модулів ігрового процесу» (рис. 2.4);

– «Завантаження моделей компонентів» (рис. 2.5);

– «Проведення ігрової партії» (рис. 2.6);

– «Підведення підсумків» (рис. 2.7).

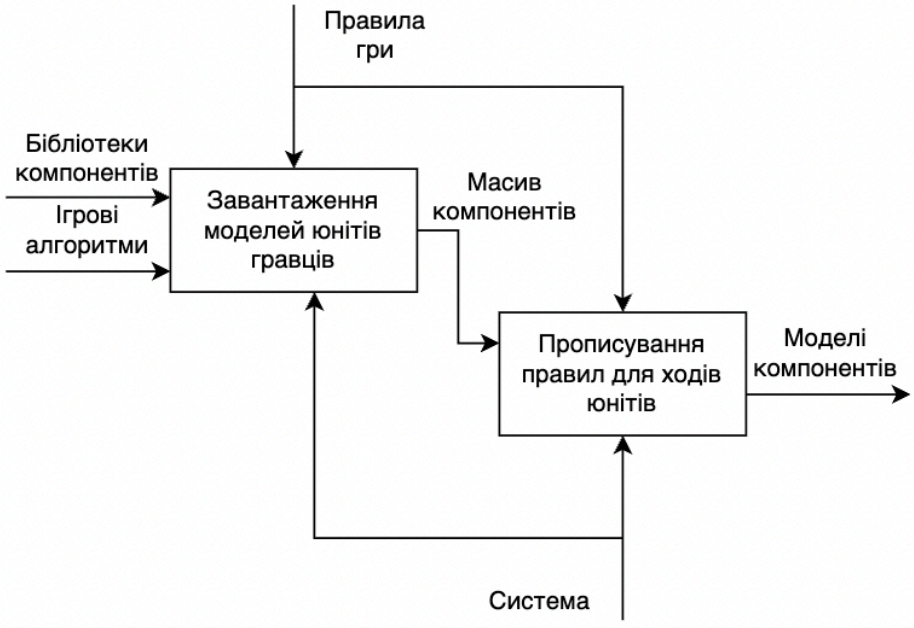

Рисунок 2.4 – Декомпозиція процесу «Завантаження модулів ігрового процесу»

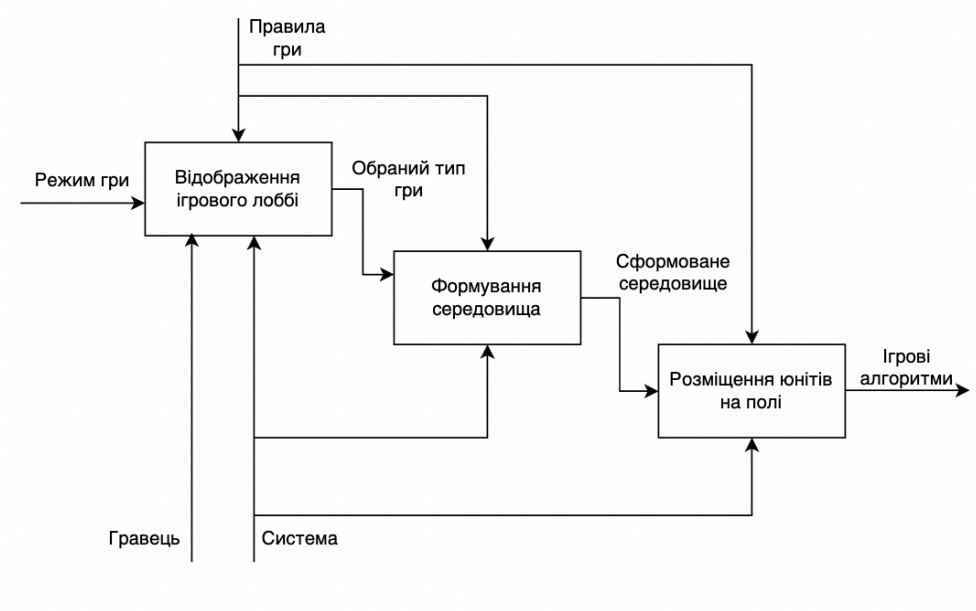

Рисунок 2.5 – Декомпозиція процесу «Завантаження моделей компонентів»

Процес «Проведення ігрової партії» представлений чотирма етапами: проведення дій на ігровому полі, формування юнітів для бою, проведення бою та закінчення ходу. Даний процес зображено на рисунку 2.6.

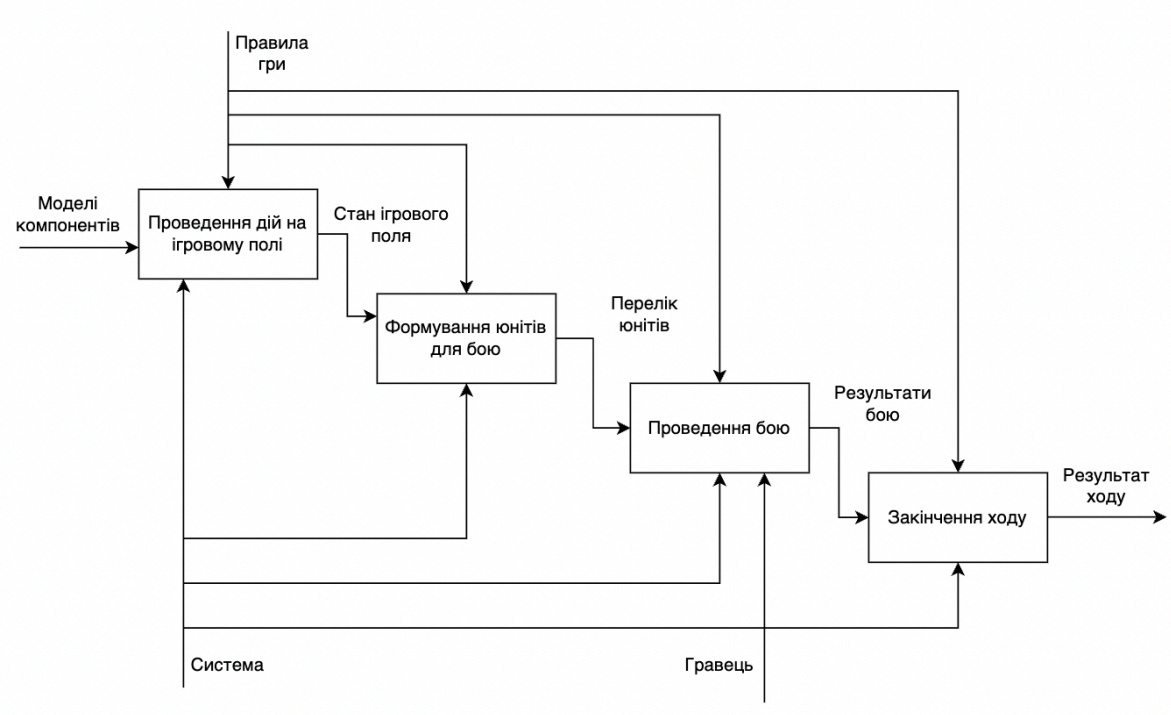

Рисунок 2.6 – Декомпозиція процесу «Проведення ігрової партії»

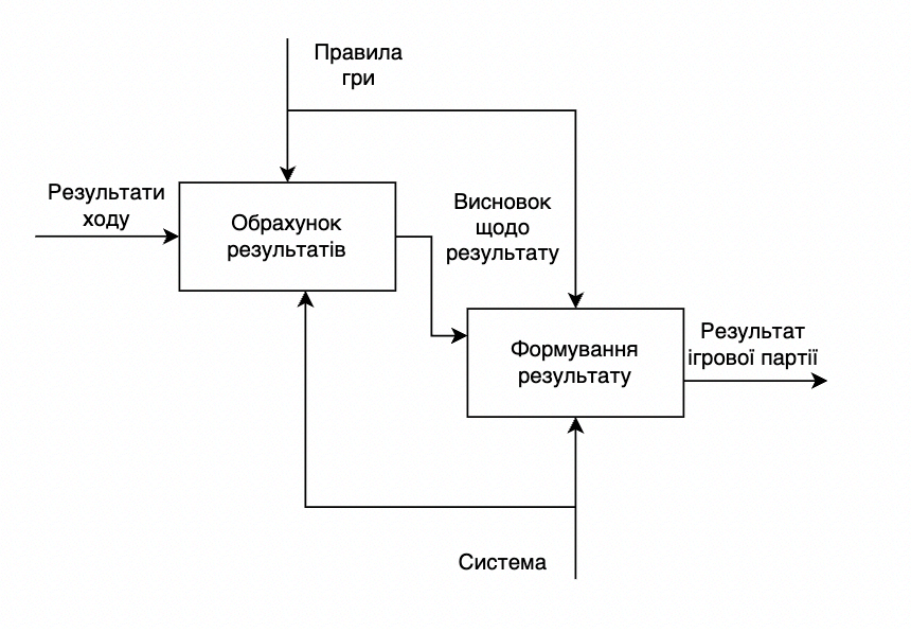

Рисунок 2.7 – Декомпозиція процесу «Підведення підсумків»

Діаграма діяльності приймається як форма технічного завдання, що містить інформацію для подальшого проєктування та реалізації будь-яким іншим методом. Вона відображає взаємодію об’єктів з плином часу, включаючи послідовність відправлення повідомлень, відкриття вікон. Діаграма діяльності відображає часові особливості передачі та прийому повідомлень об’єктами. Для ретельного проєктування та реалізації програмного продукту детально промальована діаграма послідовності є корисним інструментом, за яким і формується код.

На діаграмах наведено візуальне представлення графу діяльності для наступних процесів:

– вхід до системи (рис. 2.8);

– реалізація процесу ігрової партії (рис. 2.9);

– діяльність гравця під час бою (рис. 2.10).

Вихідний контекст для діаграми включає вхідну точку в систему, де користувач може ввести свої дані для входу. Першою подією є відправлення запиту на сервер для аутентифікації користувача.

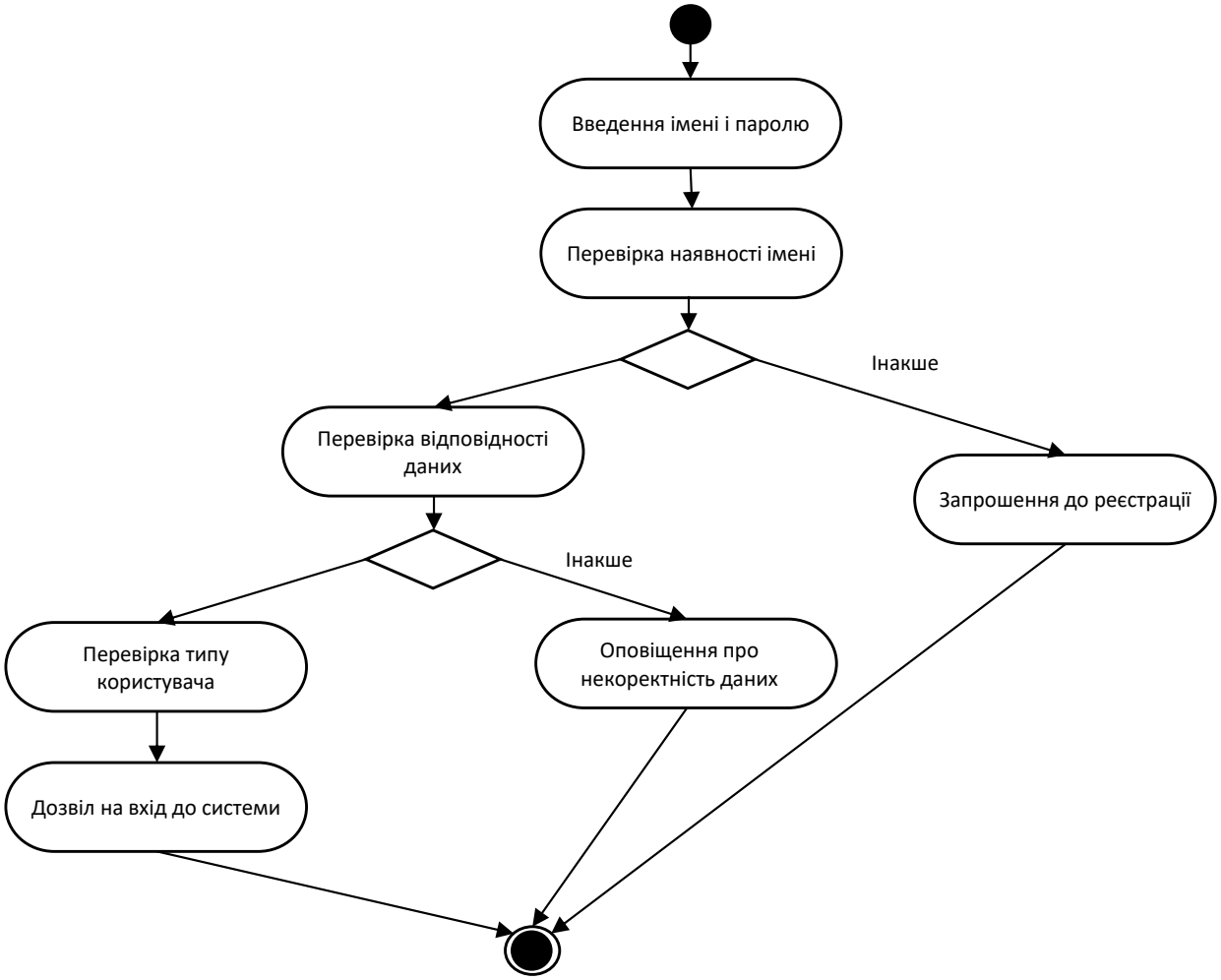

Рисунок 2.8 – Діаграма діяльності при вході в систему

Дії гравця під час партії описуються наступною діаграмою діяльності (рис. 2.9).

1. Перегляд інформації:

– Гравець може переглядати інформацію про своїх юнітів, будівлі, золото та інші параметри гри. Це дозволяє гравцю отримати уявлення про поточний стан своєї цивілізації.

2. Рух юнітів:

– Гравець може вибрати своїх юнітів і перемістити їх на інше поле на ігровій карті. Це дозволяє гравцю розширювати свої території, взаємодіяти з іншими гравцями та ресурсами.

Рисунок 2.9 – Діаграма діяльності гравця на кожному ході гри

3. Будівництво:

– Гравець може збудувати нові будівлі на своїх територіях. Кожна будівля може мати свою функціональність, яка покращує економіку або добування золота гравця. Для будівництва споруд гравець повинен мати достатньо ресурсів.

4. Виробництво юнітів:

– Гравець може створювати нових юнітів у своїх будівлях. Для виробництва юнітів гравець повинен мати достатньо ресурсів та відповідну будівлю.

5. Проведення битв:

– Гравець може здійснювати битви проти інших гравців або бота. Це включає атаку на юнітів противника.

Дії гравця для одного юніта під час бою описаються наступною діаграмою діяльності (рис. 2.10).

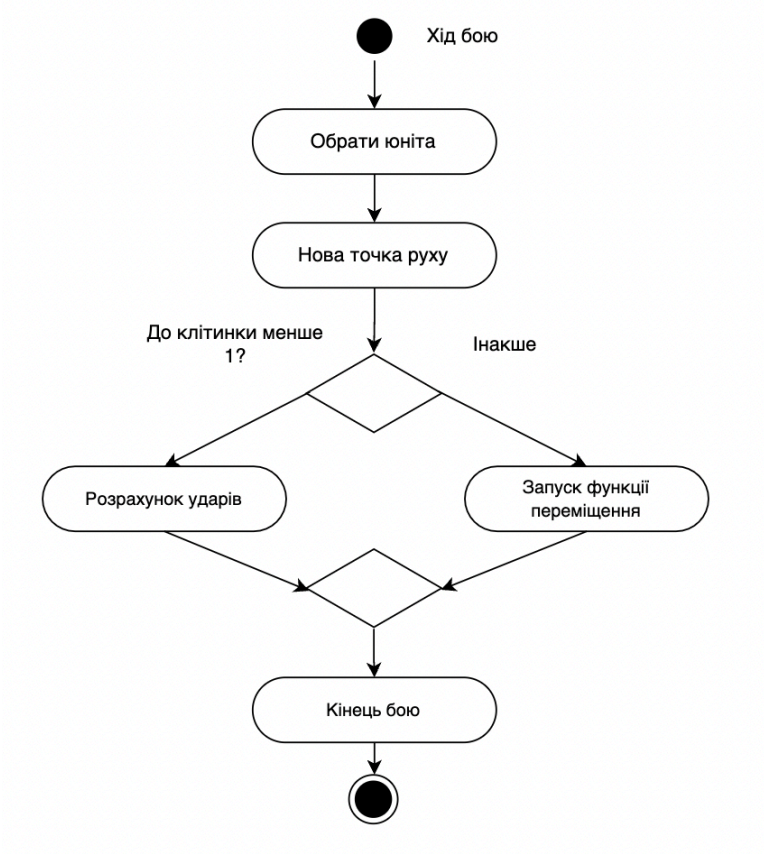

Рисунок 2.10 – Діаграма діяльності гравця на кожному юніті під час бою

Під час бою в грі гравець здійснює дії з метою перемоги над військами противника:

1. Вибір дій:

– Гравець обирає дії для своїх юнітів, які можуть включати атаку або захист.

– Кожен тип юніта може мати свої унікальні дії та можливості під час бою.

2. Атака:

– Гравець може атакувати противників, натискавши на ціль або вказуючи напрямок атаки.

– Атака може виконуватися різною зброєю в залежності від типу юніта.

– Після атаки відбувається обрахунок пошкоджень та можливих втрат для обох сторін.

3. Обрахунок результатів:

– Після того, як гравець здійснив свої дії, проводиться обрахунок результатів бою.

– Пошкодження, втрати та інші наслідки дій враховуються, і результати відображаються у вигляді змін у стані юнітів та ходів бою.

4. Повторення ходу:

– Після закінчення ходу гравця, ігровий двигун може переключитися на іншого гравця або противника для проведення їх ходу.

– Процес бою може тривати кілька ходів, поки не буде досягнуто перемоги або не буде досягнуто певних умов для закінчення бою.

На основі прописаних компонентів є можливість спроєктувати діаграму класів (рис. 2.11). Особливість обраної мови програмування дозволила більш детально виписати назви класів і їх взаємодію.

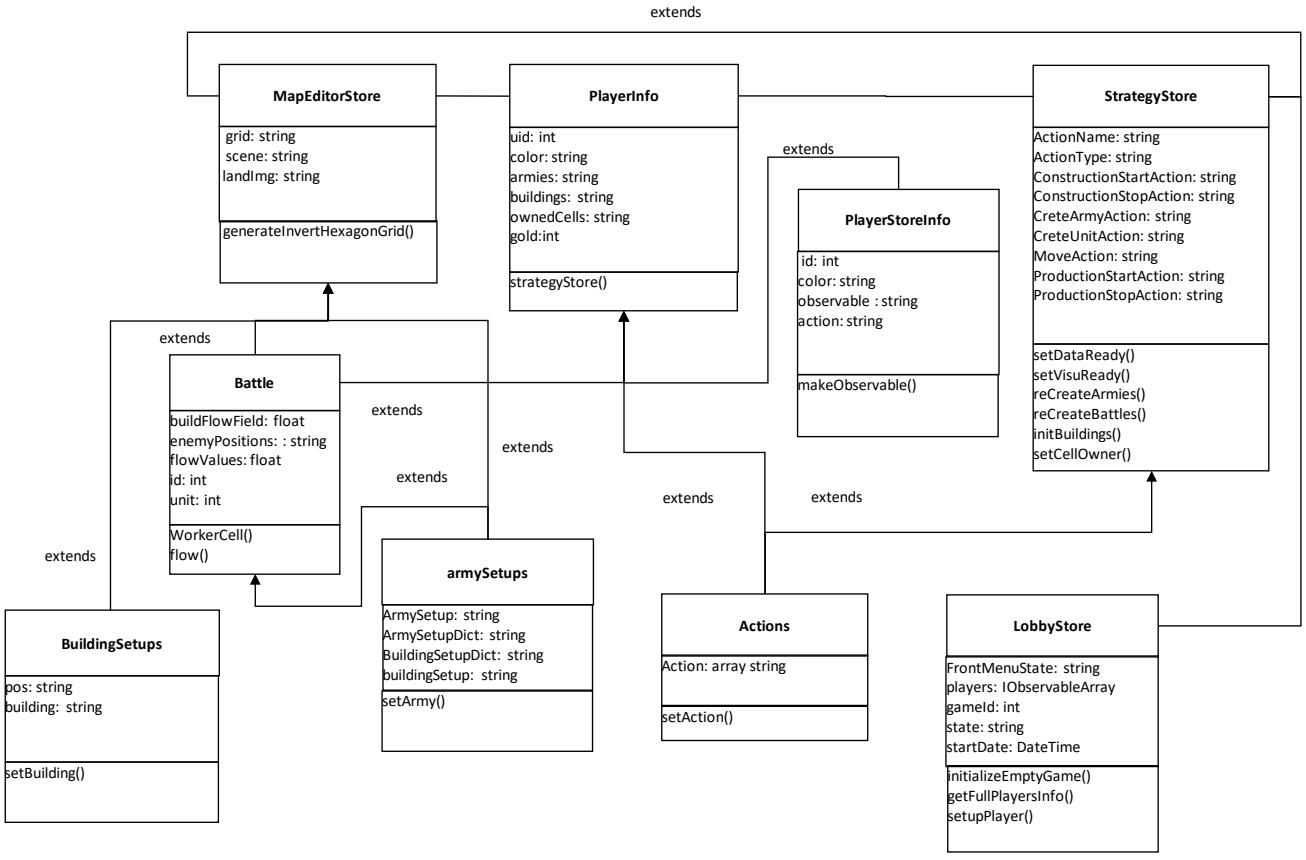

Рисунок 2.11 – Діаграма класів гри жанру 4X

Нижче представлено зміст основних класів системи з вказуванням змінних і методів:

1) LobbyStore використовується для управління лобі гри. Він містить різні методи та властивості, які дозволяють налаштувати та керувати лобі, зберігати інформацію про гравців, реагувати на події та здійснювати взаємодію з базою даних Firebase. Основні методи та властивості класу LobbyStore:

Властивості:

– fire: Об'єкт FirebaseManager, який використовується для взаємодії з Firebase.

– players: Масив IObservableArray, який містить інформацію про гравців в лобі.

– gameId: Рядок, який містить ідентифікатор гри.

– state: Стан лобі, представлений перелічувальним типом FrontMenuState.

– startDate: Об'єкт Date, який визначає початкову дату гри.

Методи:

– updateStartDate(): Оновлює початкову дату гри на основі поточного часу.

– listenForNewGame(onNewGame): Слухає нові ігри, викликаючи onNewGame з ідентифікатором нової гри.

– isMaster(): Перевіряє, чи є поточний користувач майстром гри.

– gp(append): Повертає шлях до гри з доданим суфіксом append.

– setupPlayer(): Налаштовує гравця і повертає його ідентифікатор.

– setState(menuState): Встановлює стан лобі.

– initializeEmptyGame(initStrategyData): Ініціалізує порожню гру з вказаною інформацією про стратегію.

– removeGame(): Видаляє гру.

– getStrategyInfo(): Отримує інформацію про стратегію.

– getFullPlayersInfo(): Отримує повну інформацію про гравців.

– getColors(): Отримує доступні кольори.

– setPlayerInfo(uid, playerInfo): Встановлює інформацію про гравця.

– onPlayers(callback): Слухає зміни у гравцях та викликає callback з

оновленою інформацією про гравців.

– onPlayerAdded(callback): Слухає додавання гравця та викликає callback з ідентифікатором та інформацією про гравця.

– onPlayerRemoved(callback): Слухає видалення гравця та викликає callback з ідентифікатором та інформацією про гравця.

– onPlayerChanged(callback): Слухає зміни у гравцях та викликає callback з ідентифікатором та оновленою інформацією про гравця.

– saveStrategyInfo(info): Зберігає інформацію про стратегію.

– addAction(action): Додає дію до гри.

– onAction(callback): Слухає додавання дії до гри та викликає callback з новою дією.

– onTurn(callback): Слухає зміни в номері поточного ходу та викликає callback з номером поточного ходу.

– incrementTurn(): Збільшує номер поточного ходу та додає дію завершення ходу.

– onDisconnect(): Виконує дії при відключенні гравця.

– getGames(): Отримує список доступних ігор для поточного користувача.

– newGame(): Створює нову гру.

– joinLobby(): Долучає користувача до лобі.

– startListenForLobbyPlayers(): Починає слухати події додавання та видалення гравців у лобі.

– stopListenForLobbyPlayers(): Зупиняє слухання подій додавання та видалення гравців у лобі.

– onConnect(): Слухає подію підключення гравця та виконує відповідні дії.

2) PlayerStore – це сховище (store) для зберігання та керування даними гравця в грі. Основні властивості та методи класу включають:

Властивості:

– id (observable): унікальний ідентифікатор гравця.

– isOnline (observable): прапорець, що позначає, чи є гравець онлайн.

– isAI: прапорець, що позначає, чи є гравець штучним інтелектом.

– color: колір гравця.

– armies: масив об'єктів ArmyStore, який містить армії гравця.

– buildings: масив об'єктів BuildingStore, який містить будівлі гравця.

– ownedCells: набір (set) ідентифікаторів тайлів, що належать гравцю.

– gold (observable): кількість золота гравця.

Методи:

– Конструктор: приймає об'єкт strategyStore (екземпляр класу StrategyStore), playerId (ідентифікатор гравця) та об'єкт playerInfo (інформація про гравця). Встановлює властивості об'єкта з наданої інформації.

– setID: встановлює ідентифікатор гравця.

– get goldIncome: обчислює дохід гравця від золота. Використовується сума кількостей підрозділів армій, будівель та додаткового доходу від столиці.

– getGoldIncomeDetailed: повертає докладну інформацію про дохід гравця.

– addArmy: додає армію до масиву армій гравця.

– removeArmy: видаляє армію з масиву армій гравця.

– setGold: встановлює кількість золота гравця.

– applyIncome: застосовує дохід гравця, збільшуючи кількість золота на величину goldIncome.

– setIsOnline: встановлює прапорець `

3) PlayerInfoStore використовується для зберігання та керування інформацією про гравця в грі. Основні властивості та методи класу включають:

Властивості:

– id: унікальний ідентифікатор гравця.

– color: колір гравця.

– isOnline (observable): прапорець, що позначає, чи є гравець онлайн.

Методи:

– Конструктор: приймає параметр color (колір гравця) і встановлює відповідні властивості.

– setID: встановлює ідентифікатор гравця.

– setColor: встановлює колір гравця.

– setIsOnline: встановлює прапорець isOnline для позначення статусу онлайн/офлайн гравця.

– data: повертає об'єкт PlayerInfo з поточними значеннями властивостей color, isOnline та id. Цей об'єкт використовується для передачі інформації про гравця іншим частинам програми.

4) MapEditorStore використовується для зберігання та керування даними редактора карти гри. Основні властивості та методи класу включають:

Властивості:

– grid: об'єкт GridStore, який представляє сітку гри.

– scene: об'єкт MapEditorScene для відображення графічного інтерфейсу редактора карти.

– landImg (observable): рядок, що містить посилання на зображення для покриття землі на карті.

– paletteActiveTool (observable): активний інструмент палітри - null, 'unit' або 'land'.

– paletteTile (observable): тип покриття землі, вибраний на палітрі.

– paletteUnitType (observable): тип одиниці, вибраний на палітрі.

– paletteUnitColor (observable): колір одиниці, вибраний на палітрі.

– Методи:

– Конструктор: ініціалізує властивості класу та створює об'єкт GridStore з використанням generateInvertHexagonGrid.

– setLandImage: встановлює зображення для покриття землі на карті.

– setPaletteActiveTool: встановлює активний інструмент палітри.

– setPaletteTile: встановлює тип покриття землі на палітрі.

– setPaletteUnitColor: встановлює колір одиниці на палітрі.

– setPaletteUnitType: встановлює тип одиниці на палітрі.

– isPassable: перевіряє, чи можна проходити через заданий шестигранник hex.

5) StrategyStore – використовується для зберігання та керування даними та логікою стратегічної гри. Основні властивості та методи класу включають:

Властивості:

– gridSize: розмір сітки гри.

– grid: об'єкт GridStore, який представляє сітку гри.

– activeBattles: словник, що містить активні битви в грі.

– currentBattleStore (observable): поточний об'єкт BattleStore, що відповідає за поточну битву в грі.

– mapDatas: словник, що містить дані про карти гри.

– players: ObservableMap для зберігання гравців у грі.

– currentPlayer: об'єкт PlayerStore, який представляє поточного гравця.

– armies: IObservableArray для зберігання армій в грі.

– buildings: IObservableArray для зберігання будівель в грі.

– ticks (observable): лічильник ходів в грі.

– playing (observable): прапорець, що позначає, чи відтворюється гра.

– scene: об'єкт StrategyScene для відображення графічного інтерфейсу гри.

– battleWindow: об'єкт, що відповідає за вікно битви.

– stays: словник, що містить дані про перебування армій.

– colors: масив кольорів, що використовуються для гравців.

– uiState (observable): поточний стан користувацького інтерфейсу гри.

– selectedArmy (observable): поточна вибрана армія в грі.

– dataProvider: об'єкт FireStrategyDataProvider для отримання та зберігання даних стратегії.

– dataReady (observable): прапорець, що позначає, чи готові дані гри.

– visuReady (observable): прапорець, що позначає, чи готовий візуальний інтерфейс гри.

Методи:

– Конструктор: приймає параметри grid (об'єкт GridStore) і playerInfo (об'єкт PlayerInfoStore) і ініціалізує властивості класу.

– setDataReady: встановлює прапорець dataReady для позначення готовності даних гри.

– setVisuReady: встановлює прапорець visuReady для позначення готовності

візуального інтерфейсу гри.

– reCreateArmies: перетворює дані про армії з параметра armiesSetup у нові об'єкти ArmyStore та оновлює список армій у грі.

– reCreateBattles: перетворює дані про битви з параметра battles у нові об'єкти BattleStore та оновлює активні битви у грі.

– initBuildings: ініціалізує будівлі у грі з даних buildingsSetup.

– setCellOwner: встановлює гравця-власника для комірки з вказаним шестигранником.

– addNewBuilding: додає нову будівлю в гру з інформацією з параметра info і оновлює потрібні властивості комірки.

– onCellClick: обробляє клік на комірку гри.

– applyAction: застосовує виконану дію action до гри.

– applyMoveAction: застосовує дію переміщення до армії з вказаним шляхом переміщення.

– reselectArmy: змінює вибрану армію на найближчу до поточної комірки.

– setSelectedArmy: встановлює вибрану армію.

– userArmyAction: виконує дію користувача на армію з вказаним типом actionType.

– setCurrentBattleStore: встановлює поточний об'єкт BattleStore на основі комірки cell.

– getArmiesOn: повертає список армій, що перебувають на заданому шестиграннику.

– setUIState: встановлює стан користувацького інтерфейсу гри.

– setScene: встановлює сцену гри для візуалізації.

– addArmy: додає нову армію до г

6) armySetup і buildingSetup – два словника armySetup і buildingSetup, які містять налаштування армій і будівель для гри.

ArmySetup:

– pos: координати позиції армії на карті, представлені у форматі [x, y].

– units: масив з ключами типів одиниць, що належать до армії.

– goingTo (опціонально): цільові координати, куди армія збирається рухатись. Може бути представлено як окремими координатами [x, y], так і масивом координат [[x1, y1], [x2, y2], ...].

– ArmySetupDict: словник, де ключами є рядки, які відповідають кольорам армій (наприклад, 'red' і 'green'), а значеннями є масиви об'єктів ArmySetup. Кожен об'єкт представляє армію певного кольору з її налаштуваннями.

BuildingSetup:

– pos: координати позиції будівлі на карті, представлені у форматі [x, y].

– building: тип будівлі з використанням значень з BUILDING_TYPES (наприклад, BUILDING_TYPES.CASTLE).

– BuildingSetupDict: словник, де ключами є рядки, які відповідають кольорам гравців, а значеннями є масиви об'єктів BuildingSetup. Кожен об'єкт представляє будівлю певного типу із відповідними координатами для гравця певного кольору.

2.1.3 Побудова діаграми EPC для розроблюваної гри жанру 4X

Діаграми EPC (Event-driven Process Chain) – це графічний метод моделювання бізнес-процесів, що використовується для опису послідовності подій та функцій у бізнес-процесах. Вони дозволяють візуалізувати залежності між подіями, функціями та учасниками процесу.

Основні елементи діаграм EPC включають:

1. Події (Events): Події представляють важливі моменти часу або стану, які спричиняють запуск певних функцій у бізнес-процесі. Вони позначаються круглими об'єктами і мають назву, що описує подію, наприклад, "Замовлення отримано" або "Платіж здійснений".

2. Функції (Functions): Функції виконують дії або операції у бізнес-процесі. Вони позначаються прямокутниками і мають назву, яка описує виконувану дію, наприклад, "Перевірити наявність товару" або "Сформувати звіт".

3. Учасники (Participants): Учасники представляють ролі або структурні

компоненти, які беруть участь у виконанні бізнес-процесу. Вони можуть бути людьми, системами, відділами або зовнішніми структурами. Учасники позначаються зовнішніми об'єктами, які мають назву, наприклад, "Клієнт", "Складський відділ" або "Бухгалтерія".

4. Зв'язки (Connections): Зв'язки відображають залежності між подіями, функціями та учасниками в бізнес-процесі. Вони показують напрямок потоку даних або керування від одного елемента до іншого. Зв'язки можуть мати стрілки, що показують напрямок потоку, а також мітки, які вказують тип зв'язку (наприклад, "виконується після" або "генерує").

Діаграми EPC дозволяють чітко моделювати бізнес-процеси, визначати послідовність подій та функцій, а також ідентифікувати учасників процесу та залежності між ними. Вони допомагають покращити розуміння та комунікацію щодо бізнес-процесів у команді або організації.

Основні компоненти діаграми EPC для гри наступні (рис. 2.12):

1. Події:

– запуск ігрового лобі;

– підключення до сервера гри;

– приєднання до доступного лобі;

– створення нової гри;

2. Функції:

– взаємодія з ігровим полем;

– вибір елементу поля;

– взаємодія з юнітами (переміщення, перегрупування);

– взаємодія з будівлями (будівництво, виробництво юнітів);

– управління економікою (золото, витрати);

– проведення битви;

– взаємодія з іншими гравцями;

3. Учасники:

– гравець;

– сервер гри;

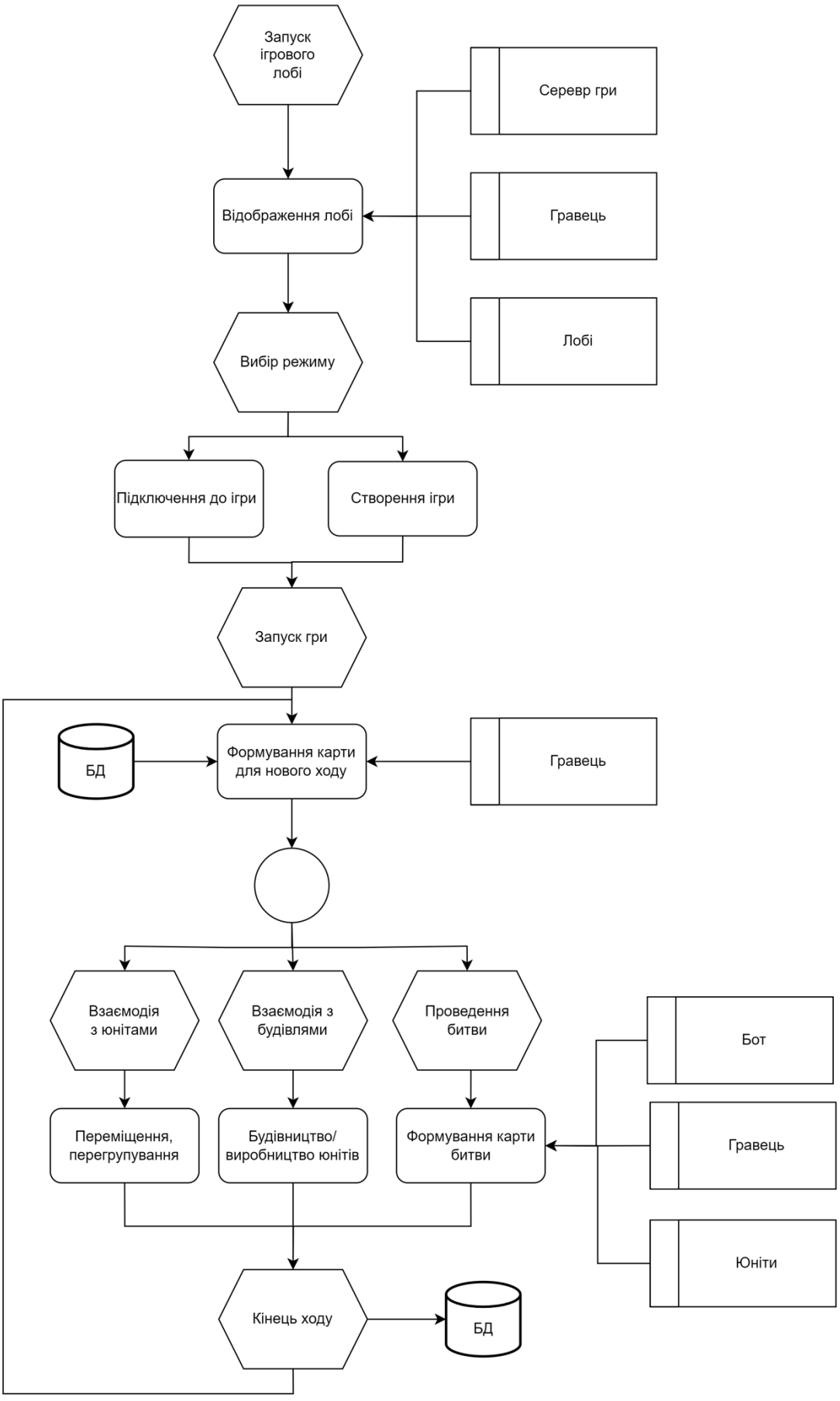

Рисунок 2.12 – Діаграма EPC гри жанру 4X

– бот;

– юніти;

– будівлі;

4. Зв'язки:

– підключення до сервера після запуску гри;

– приєднання до доступного лобі;

– створення нової гри;

– взаємодія з ігровим полем після вибору елементу;

– взаємодія з юнітами (переміщення, перегрупування);

– взаємодія з будівлями (будівництво, виробництво юнітів);

– управління економікою (золото, витрати);

– проведення битви.

2.2 Проєктування макету ігрового середовища

Проєктування макету ігрового поля включає такі етапи:

1. Визначення розміру поля: ігри жанру 4X мають спільний дизайн, який передбачає карту, інформаційні вікна, вікна характеристик обраних об’єктів та кнопку управління прогресом гри (рис. 2.13).

2. Встановлення типу поля: використовується клітинкова структура, де кожна клітинка представляє одиницю території або місця. Для гри було обрано мінімалістичний дизайн і розроблено прототип ігрового поля (рис. 2.15).

3. Розміщення об'єктів на полі: будівлі, ресурси, вороги і т.д. (рис. 2.16).

4. Встановлення обмежень та перешкод: обмеження та перешкоди на полі, які впливають на рух та взаємодію об'єктів.

5. Визначення іконок та символів: використано візуальні елементи, такі як іконки та символи, для представлення різних об'єктів та їх стану на полі.

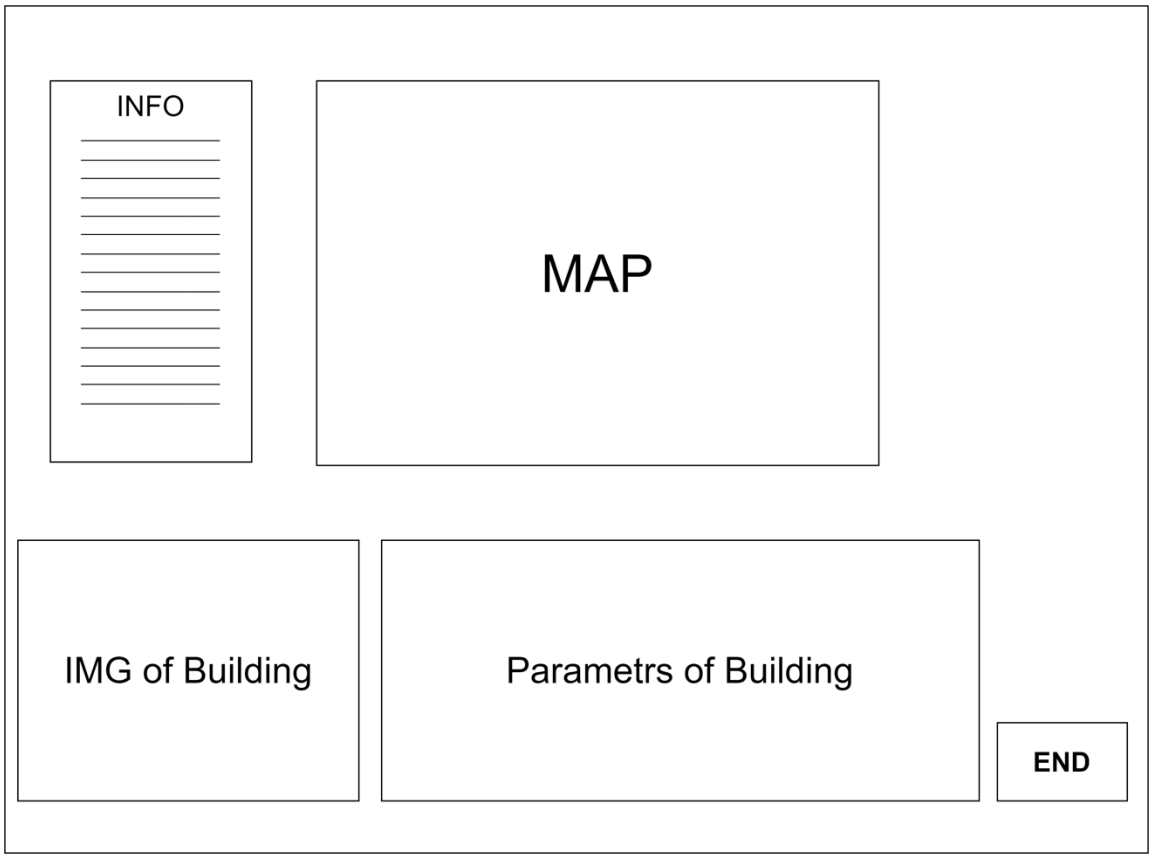

Рисунок 2.13 – Макет ігрового середовища

2.3 Інформаційне забезпечення

Проєктування інформаційного забезпечення гри включає розробку системи для збереження, обробки і передачі інформації, необхідної для функціонування гри. Проведений аналіз процесів дозволив спроєктувати концептуальну модель БД (рис. 2.14).

З урахуванням складності зв’язків і обраної мови програмування React.JS вирішено використовувати noSQL СУБД Firebase.

Особливості розгортання бази даних у форматі NoSQL на Firebase також дозволили досягти: гнучкості структури даних (Firebase надає гнучкість щодо структури даних і можна зберігати дані у форматі JSON або колекції, уникнувши складних схем, які зазвичай потрібні в реляційних базах даних); – швидкодія (Firebase пропонує швидкий доступ до даних завдяки своїй архітектурі, включаючи розподілені сервери та кешування на рівні клієнта).

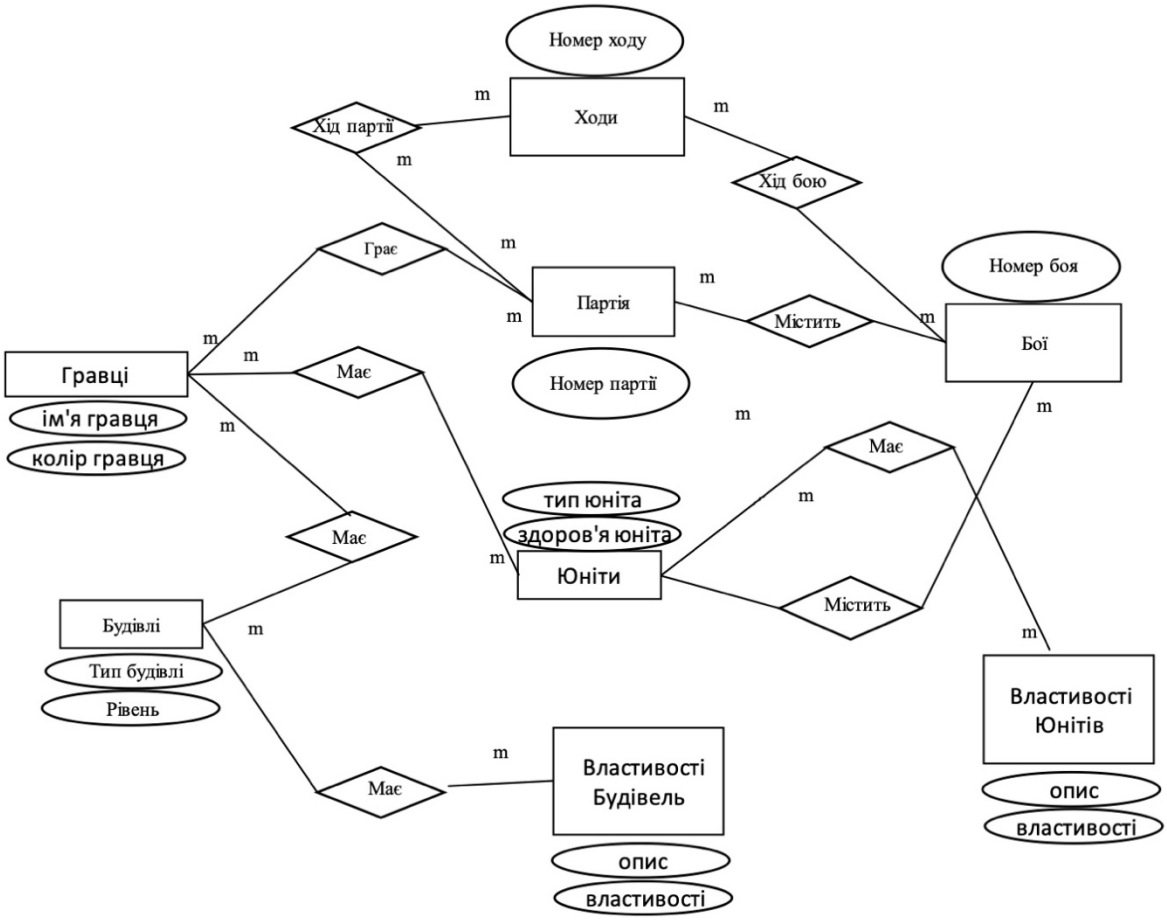

Рисунок 2.14 – Концептуальна модель БД для гри жанру 4X

Частина файлу JSON представлено в додатку Б. Розглянемо більш детально структуру noSQL БД:

– ключ "games" містить об'єкт, в якому кожен ключ є ідентифікатором гри, а значення - об'єкт зі списком дій (actions) в рамках цієї гри (рис. 2.15);

А) Б)

Рисунок 2.15 – Структура noSQL БД: А) верхній рівень; Б) рівень games «1»

– кожен об'єкт дії має ключі: "action" (тип дії), "armyId" (ідентифікатор армії), "player" (гравець, який виконує дію) та додаткові ключі, такі як "path" (шлях руху армії), "buildingId" (ідентифікатор будівлі), "buildingType" (тип будівлі), "hex" (координати) та "type" (тип юніта) (рис. 2.16);

А) Б)

Рисунок 2.16 – Структура noSQL БД: А) рівень armies; Б) рівень buildings

– деякі дії мають тип "turnEnd", що вказує на закінчення ходу гравця;

– дії з типом "constructionStartAction" та "productionStartAction" вказують на початок будівництва та виробництва відповідно, і містять додаткову інформацію про будівлю та тип виробу;

– ключ "colors" містить масив з двома кольорами - "green" та "orange";

– ключ "masterPlayerId" містить ідентифікатор головного гравця (master player);

– ключ "players" містить об'єкт, в якому кожен ключ є ідентифікатором гравця, а значення - об'єкт з властивістю "isOnline", яка вказує, чи гравець наявний в мережі (true або false). Також є окремий ключ "color", що містить колір "orange";

– ключ "startTime" містить значення часу початку гри;

– ключ "strategy" містить об'єкт з двома масивами: "armies" та "buildings";

– масив "armies" містить об'єкти з інформацією про армію. Кожен об'єкт

армії має ключі: "hex" (координати армії), "id" (ідентифікатор армії), "movingPath" (шлях руху армії), "player" (гравець, якому належить армія) та "units" (масив з об'єктами, що представляють юнітів армії). Кожен об'єкт юніта має ключі "health" (здоров'я юніта), "id" (ідентифікатор юніта), "player" (гравець, якому належить юніт) та "type" (тип юніта);

– масив "buildings" містить об'єкти з інформацією про будівлі. Кожен об'єкт будівлі має ключі "buildingType" (тип будівлі), "hex" (координати будівлі), "id" (ідентифікатор будівлі) та "player" (гравець, якому належить будівля);

– ключ "turn" містить номер поточного ходу.

2.4 Висновки до другого розділу

В результаті написання розділу було проведено аналіз сучасних тенденцій та напрямків розвитку веб-ігор жанру 4X а також було представлено діаграми IDEF0 "ЯК БУДЕ" гри жанру 4X, які показують процеси, які використовуються в іграх жанру 4X. Аналіз основних процесів предметного середовища для розробки веб-ігор жанру 4X дозволив зрозуміти, які елементи та функції повинні бути враховані при розробці ігри.

За рахунок побудови діаграм послідовності використання системи стало можливим відокремити процеси і сформувати зміст їх функціонування. Розроблено макети дизайну гри та описано БД для збереження інформації.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ СТРАТЕГІЧНОЇ ГРИ ЖАНРУ 4X

3.1 Архітектура системи та описання оточення для розгортання стратегічної гри жанру 4X

Для розробки системи було обрано мову React.JS. З огляду на це запуск системи і зміст компонентів мають чітко виражену структуру [26], яка притаманна програмам мовою React.JS.

На рис. 3.1 представлено архітектуру проєкта.

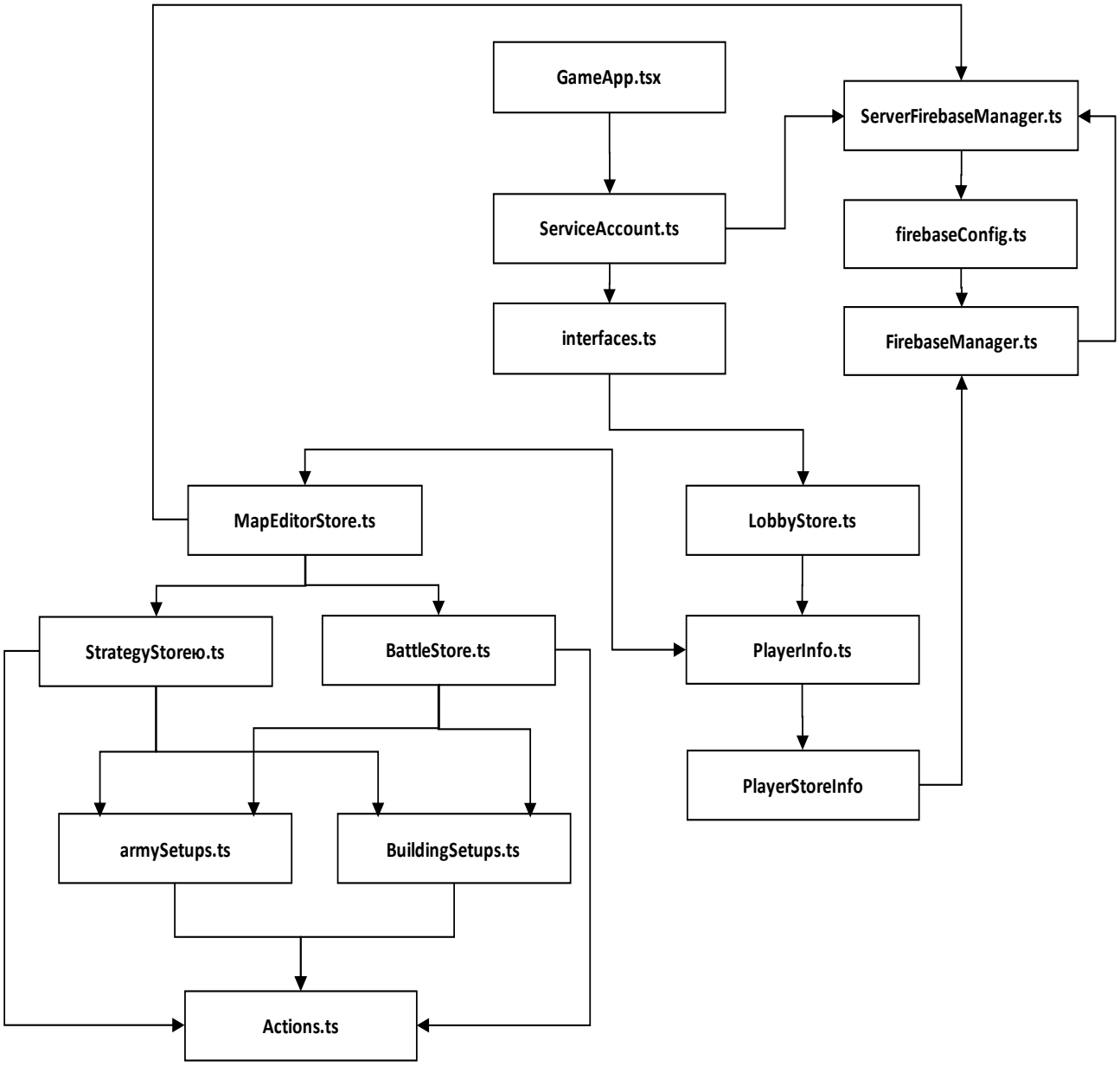

Рисунок 3.1 – Архітектура системи для реалізації гри жанру 4X

Запуск системи передбачає запуск окремих скриптів, де після старту додатку, активізується скрипт рівнів доступу клієнта (ServiceAccount.ts), через який розкривається інтерфейс системи і проводиться запуск сервера БД, який містить свій набір скриптів для роботи (firebaseConfig.ts і FirebaseManager.ts).

Візуалізація першого контакту з користувачем проходить через лоббі (LobbyStore.ts), яке активізує скрипти інформації про користувача (PlayerInfo.ts і PlayerStoreInfo).

Користувач починає взаємодію з ігровою картою (MapEditorStore.ts), яка передбачає дії на карті (StrategyStoreю.ts) і дії в бою (BattleStore.ts), які застосовують скрипти дій (Actions.ts) для армій (armySetups.ts) і будівель (BuildingSetups.ts).

Скрипти можуть підключати додаткові бібліотеки і імпортувати дані з інших скриптів. Всі дії фіксують в БД, шляхом постійного запису кожного кроку гравця.

Покрокове представлення етапів запуску гри (рис. 3.2):

1) запускається сервер;

2) через API підключаємося до Firebase;

3) передаються данні про користувача в лобі;

4) отримується відповідь на екрані, що в лобі є Гравець 0 з певним ID;

5) кнопка "Join" дозволяє приєднатися до гри.

6) якщо лобі не оновилось (пройшло 10 хвилин), то гра не запуститься, оскільки інформація про гравців не надходила.

7) якщо лобі оновлено, то завантажуються наступні файли: src/store/strategy/PlayerStore.ts (Інформація про гру) та src/store/strategy/PlayerInfoStore.ts (Інформація про гравців).

В проєктах на React.js з використанням Firebase каталоги зазвичай мають свою структуру і особливості. Основна структура каталогів зазвичай відображає структуру бази даних Firebase.

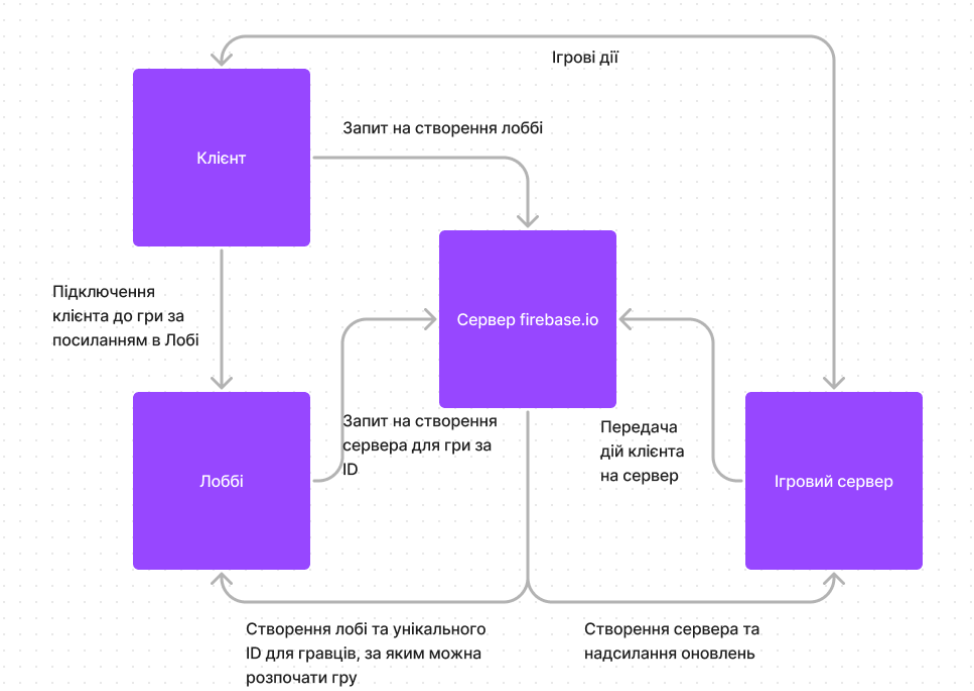

Рисунок 3.2 – Порядок завантаження ігрового додатку

3.2 Принципи функціонування основних скриптів системи

Для запуску гри виконується підключення до сервера firebase.io. За даний модуль відповідає код з файлу src/server/ServerFirebaseManager.ts.

Дані передаються на сервер firebase.io і у відповідь отримуємо id гравця, а також id інших гравців, якщо вони в лобі та не підключені до інших ігор.

Далі керування передається в розділ «Лобі» (рис. 3.6). Файл даного модуля знаходиться за адресою pages/api/joinLobby.ts та код наведено в Додатку А.

Рисунок 3.3 – Вигляд лобі системи

3.2.1 Обробка подій в режимі лобі і старт гри

Гравець підключається до гри за допомогою кнопки “Join”. При підключенні до гри, передається інформація на сервер firebase.io, далі йде перевірка [GameSetupStore], файл з кодом перевірки src/store/GameSetupStore.ts.

Якщо немає даних про клієнта (не оновлене лобі або гра не створена) - виникає помилка (рис. 3.4).

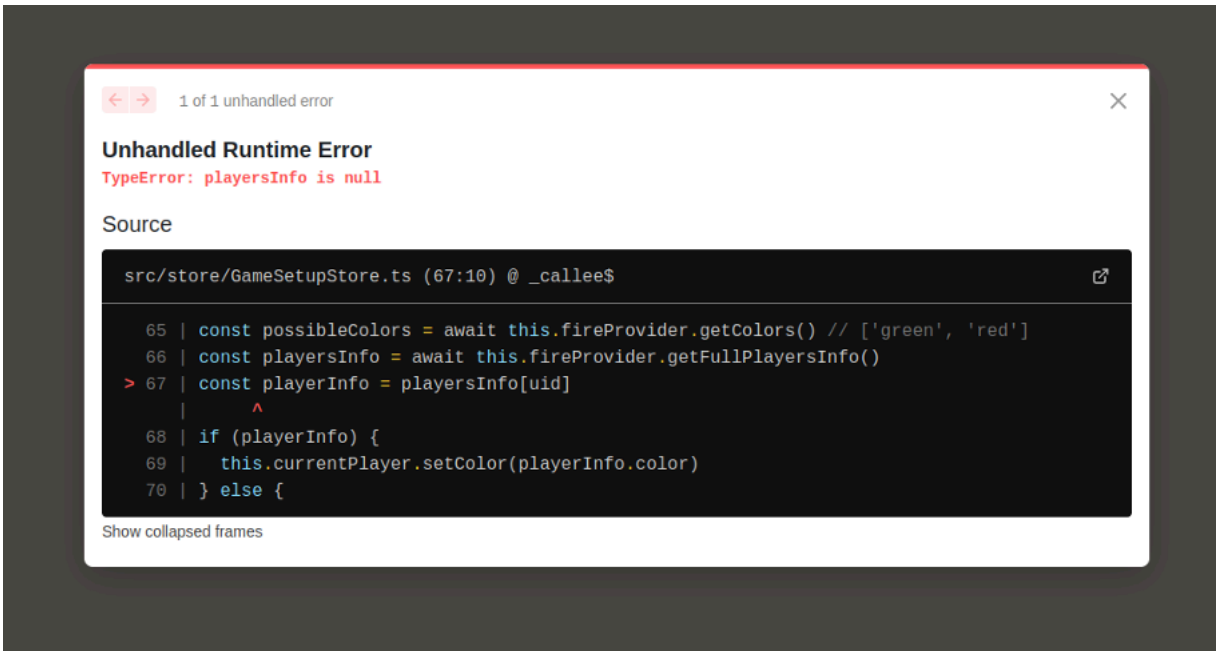

Рисунок 3.4 – Вікно помилки

Якщо гра пуста, то створюється початкова гру та присвоюється гравцю зелена команда (рис. 3.5).

Якщо у грі вже є один гравець, то присвоюється червона команда (рис. 3.6).

Приєднання нових гравців видає повідомлення (рис. 3.7), що немає доступних кольорів (гра заповнена).

Запущена гра передбачає взаємодію за наступними варіантами:

– [StrategyStore]– файл коду src/store/strategy/StrategyStore.ts;

– [CellStore] – файл коду src/store/CellStore.ts.

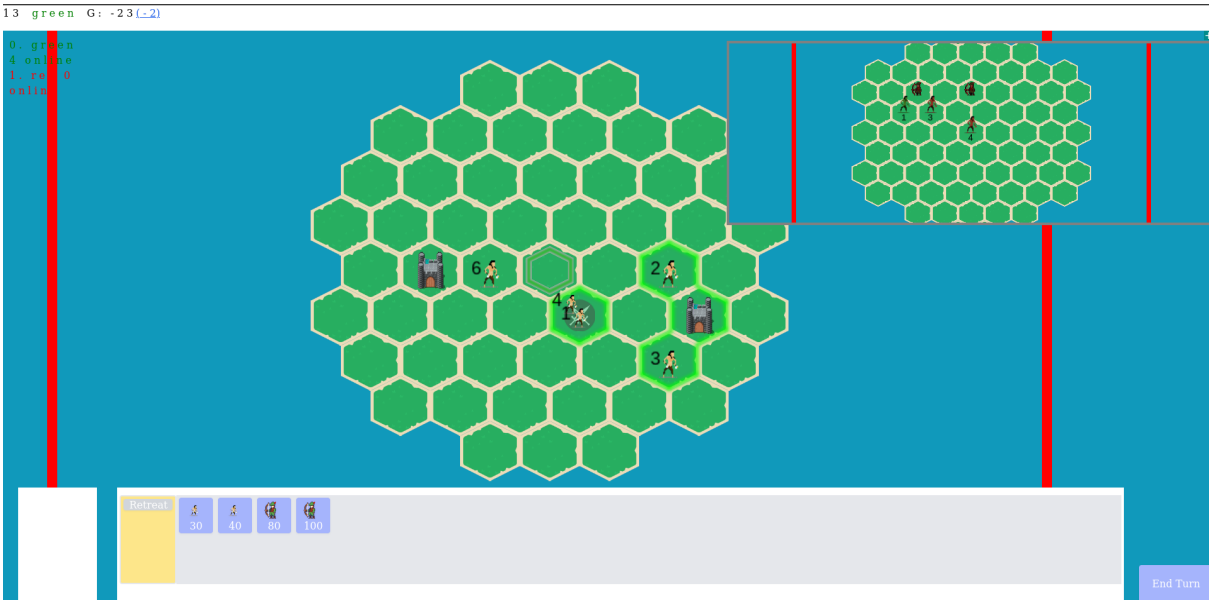

Рисунок 3.5 – Вікно початку гри для першого гравця

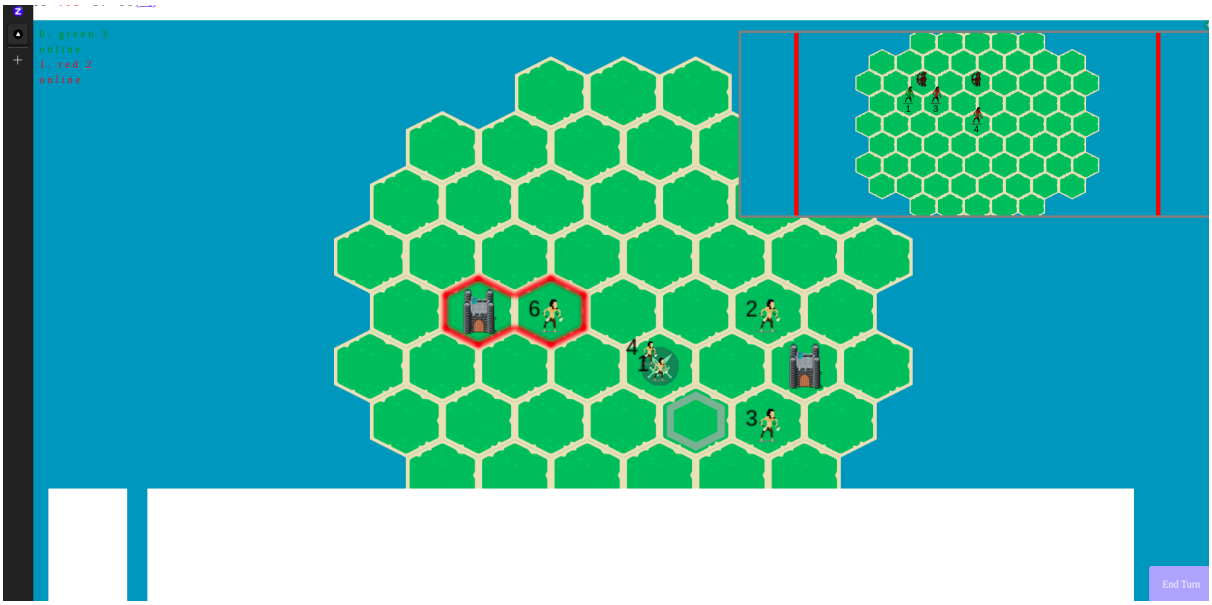

Рисунок 3.6 – Вікно початку гри для другого гравця

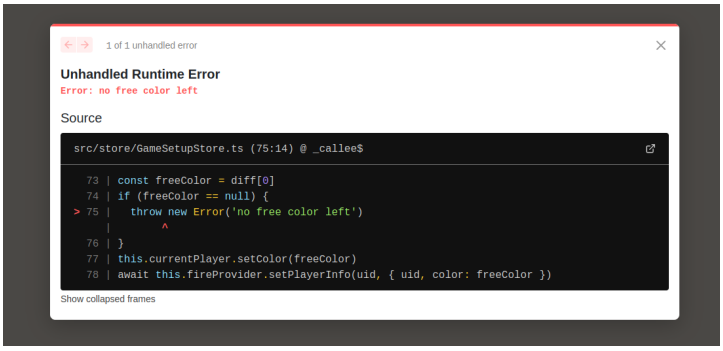

Рисунок 3.7 – Вікно помилки підключення третього гравця

3.2.2 Елементи управління на карті гри

Особливість гри формату Цивілізація передбачає роботу з гексагональною сіткою і має свої особливості порівняно з традиційними квадратними сітками:

1. Координатна система: гексагональна сітка використовувати різні системи координат. Два найпоширеніших варіанти - це "плоска верхня" (flat top) та "плоска бічна" (pointy top). В роботі обрано flat top.

2. Перетини: у гексагональній сітці, кожен шестикутник має шість сусідніх шестикутників. Це означає, що обробка перетинів між шестикутниками вимагає спеціального підходу. Наприклад, для знаходження сусідніх шестикутників треба використовувати алгоритми на основі гексагональних координат, такі як алгоритми на основі кубових координат (cube coordinates) або аксіальних координат (axial coordinates).

3. Розміщення об'єктів на гексагональній сітці вимагає додаткових обчислень. Наприклад, для знаходження координат об'єкта на сітці використовуватися формули, які враховують розміри та відстані між шестикутниками.

4. Пошук шляху: алгоритми пошуку шляху, такі як алгоритм A* або Dijkstra (в роботі використано алгоритм A*), потребують модифікацій для використання з гексагональною сіткою. Вони повинні враховувати специфіку руху по шестикутниках та знаходити оптимальні шляхи, враховуючи сусідні шестикутники.

5. Графічне відображення: рендеринг гексагональної сітки може вимагати особливих алгоритмів для коректного відображення шестикутників та їх взаємного розташування. Наприклад, для створення гексагональної сітки на графічному полотні можуть використовуватися певні математичні формули та алгоритми.

Звичайний алгоритм A* можна застосувати й на гексагональній сітці. Для цього потрібно врахувати особливості гексагональної топології. Основні кроки

реалізації алгоритму A* на гексагональній сітці у React.js наступні:

1. Створити компоненти для представлення гексагональної сітки, наприклад, HexGrid, HexTile.

2. Визначити структуру даних для збереження інформації про кожен гексагон, таку як його координати, статус (прохідність), вартість переходу тощо.

3. Реалізувати функцію для обчислення евристичної оцінки від поточного гексагону до цільового гексагону. Ця функція використовуэ евклідову відстань.

4. Реалізувати функцію findPathAStar, яка виконуватиме алгоритм A* (код представлено в Додатку А) для пошуку найкоротшого шляху між двома гексагонами. У цій функції використовуйте відкритий та закритий списки, обчислюйте вартість переходів із одного гексагону в інший, оцінюйте вартість шляху, а також оновлюйте інформацію про найкращий шлях.

5. У компоненті HexGrid реалізувати функції для взаємодії з користувачем, наприклад, вибір стартового та цільового гексагону, ініціалізація гексагональної сітки з властивостями прохідності, тощо.

6. Використовуйте отримані дані для відображення шляху на гексагональній сітці, наприклад, змінюючи колір або стиль гексагонів, через які проходить шлях.

Клік по полю - показує інформацію: кількість юнитів, будівлі, та взаємодії з цими елементами супроводжується візуально (рис. 3.8). Поля також підсвічуються, якщо ваш юніт перебував на ньому три ходи (рис. 3.9).

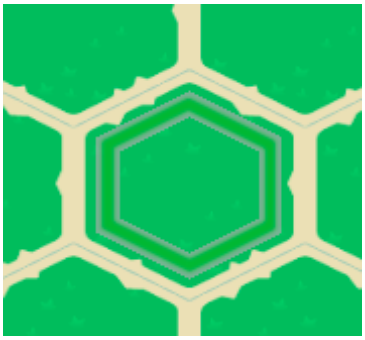

Рисунок 3.8 – Обробка кліку по полю

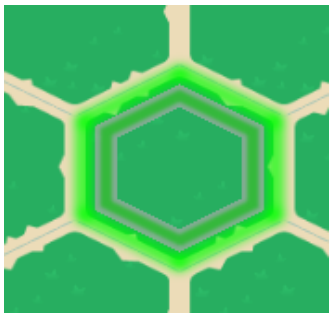

Рисунок 3.9 – Режим підсвітки поля

3.2.3 Обробка подій з юнітами

Для обробки Юнітів ([ArmyStore]) використовується файл с кодом src/store/ArmyStore.ts.

Клік по полю з юнітом має наступну варіацію дій (рис. 3.10):

– переміщення юніта;

– припинення переміщення, якщо виконувалась перша дія;

– перегрупування юнітів.

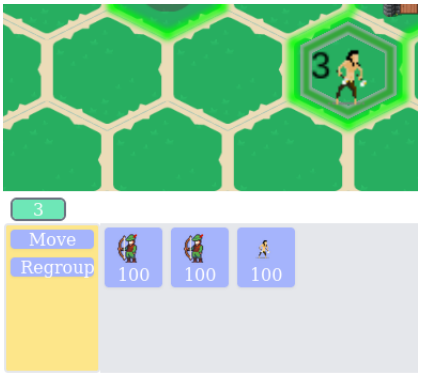

Рисунок 3.10 – Дії з юнітом

Переміщення юнітів: при кліку на цю дію, потрібно клікнути на поле куди хочемо перемістити юнітів. Буде побудовано шлях за яким юніти переміщуватимуться на вказане поле (рис. 3.11).

Для побудови шляху використовується алгоритм A* на сітці гексагональних тайлів.

Рисунок 3.11 – Переміщення юніта

Основні модифікації алгоритму від стандартного полягають в наступному:

– Клас FullHeap представляє пріоритетну чергу з використанням кучі (heap). Конструктор створює порожню чергу та об'єкт для підтримки швидкого доступу до елементів черги за допомогою словника support. Методи push та pop використовуються для додавання елементів у чергу та видалення найменшого елемента відповідно. Метод size повертає поточний розмір черги.

– Функція findPathAStar використовує алгоритм A* для пошуку найкоротшого шляху на сітці гексагональних тайлів. Вхідні параметри включають сітку grid, словник blocked для блокованих тайлів, точку from (початковий тайл) та точку end (цільовий тайл). Опціональні параметри ownColor та giveClosest використовуються для додаткових умов пошуку шляху.

Функція спочатку перевіряє, чи початковий та кінцевий тайли співпадають, і повертає порожній масив, якщо це так. Далі створюється пріоритетна черга openHeap за допомогою класу FullHeap і додається початковий вузол до черги. В основному циклі алгоритму, доки черга не порожня, вилучається вузол з найнижчим пріоритетом. Якщо цей вузол є кінцевим тайлом, то відбувається повернення шляху, що веде до цільового тайла. В іншому випадку обробляються сусідні тайли, перевіряючи їх на блокування та знаходження найкоротшого шляху до них. Знайдені тайли

додаються до черги openHeap для подальшого розгляду. Якщо giveClosest встановлено на true, то повертається найближчий тайл до цільового тайла, якщо немає прямого шляху до нього.

На виході функція повертає масив шляху, який складається з координат гексагональних тайлів.

Перегрупування юнітів (рис. 3.12) відкриває меню перегрупування юнітів. Переміщуючи їх вниз, розділяємо одну групу юнітів на дві.

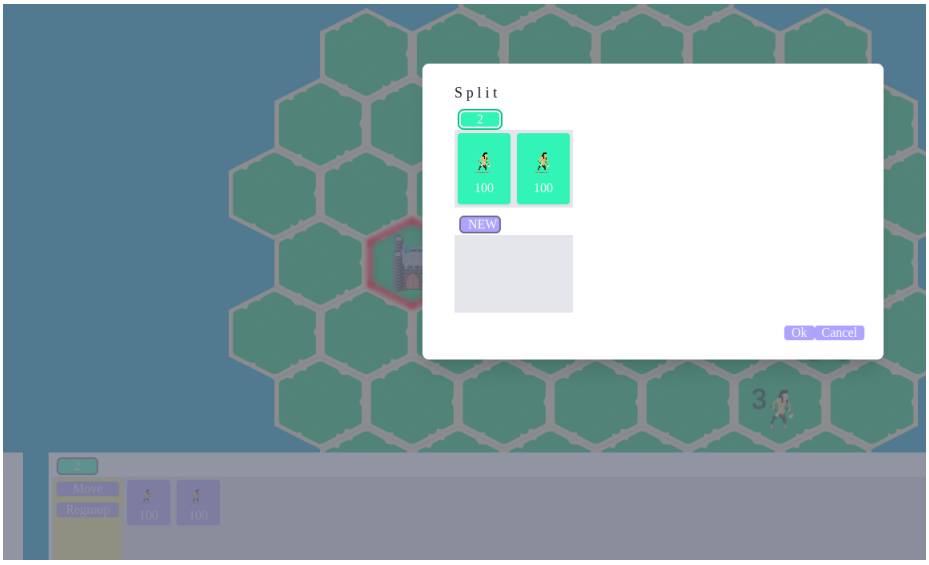

Рисунок 3.12 – Перегрупування юнітів в дві групи

Якщо на одному полі знаходиться дві групи юнітів, їх можна перегрупувати в одну групу (рис. 3.13).

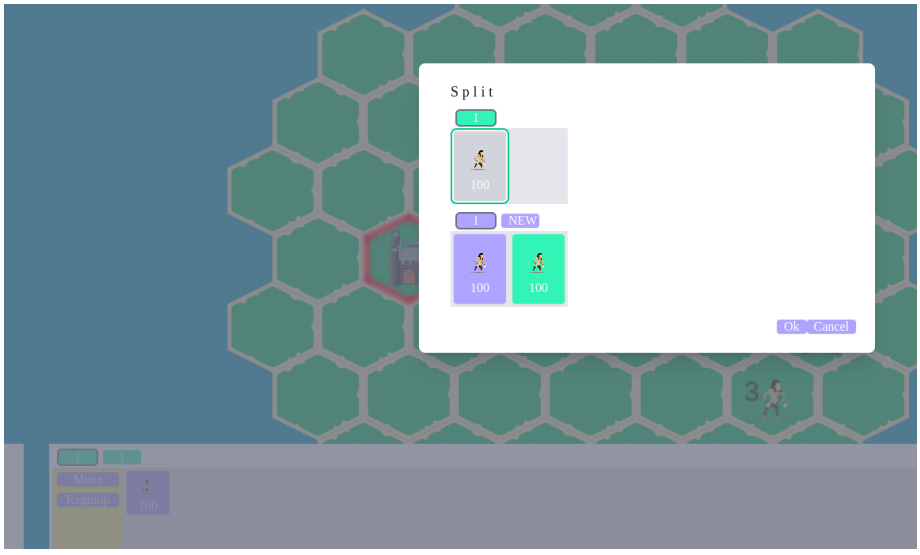

Рисунок 3.13 – Перегрупування юнітів в одну групу

3.3.4 Обробка подій з будівлями

В грі було реалізовано Будівлі ([BuildingStore]), код методу створення і обробки в файлі src/store/BuildingStore.ts.

Якщо поле належить гравцю, то є можливість створювати будівлі. За це відповідатиме клавіша “Construct” (рис. 3.14).

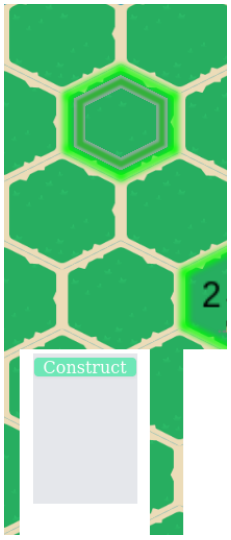

Рисунок 3.14 – Кнопка створення будівлі

При цьому з’явиться меню вибору будівлі, яку можна збудувати (рис. 3.15).

Рисунок 3.15 – Вибір будівлі

Якщо коштів недостатньо, то видаємо напис про це, та будівництво буде заборонене. В іншому випадку почнеться будівництво будівлі. Термін будівництва – 3 ходи.

Після будівництва будівлі, з нею можна взаємодіяти (рис. 3.16).

Рисунок 3.16 – Створена будівля

Основні дії з будівлями:

1) замок. Створює юніта за 3 ходи (рис. 3.17).

Рисунок 3.17 – Будівля «Замок»

2) вежа. Виконує функції захисту (рис. 3.18).

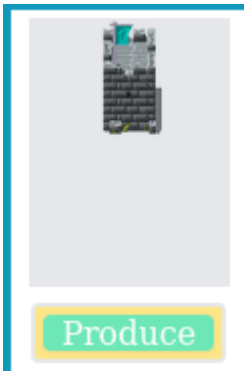

Рисунок 3.18 – Будівля «Вежа»

3) хижина. Кожного кроку приносить 1 золото (рис. 3.20).

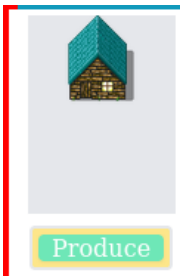

Рисунок 3.20 – Будівля «Хижина»

4) барак. Створює юніта за 5 ходів (рис. 3.21).

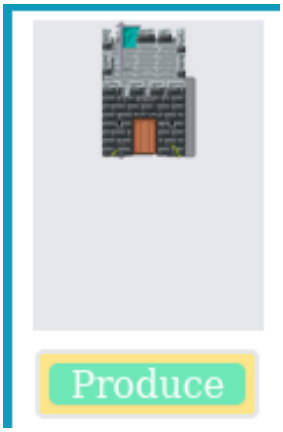

Рисунок 3.21 – Будівля «Барак»

Робота з грошима відбувається за наступними правилами:

– 1 юніт споживає 1 золото за хід;

– Головний замок приносить 4 золота за хід;

– Хижина приносить 2 золота за хід.

Кількість зроблених ходів та кількість золота знаходяться у лівому верхньому кутку. Також можна переглянути детальну інформацію про золото (рис. 3.22).

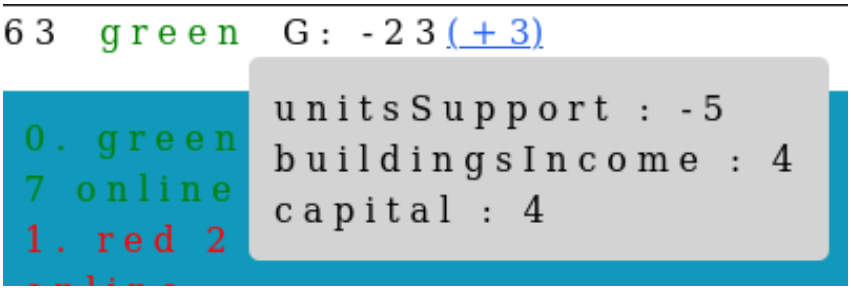

Рисунок 3.22 – Інформація про золото гравця

3.2.5 Обробка режиму битви

Якщо юніти протилежних команд зустрічаються на полі - запускається битва. Розміщення команд на полі битви пропорційне розміщенню команд на полі в основній стратегії (рис. 3.23). Режим Битви ([BattleStore]) прописано в файлі src/store/BattleStore.ts.

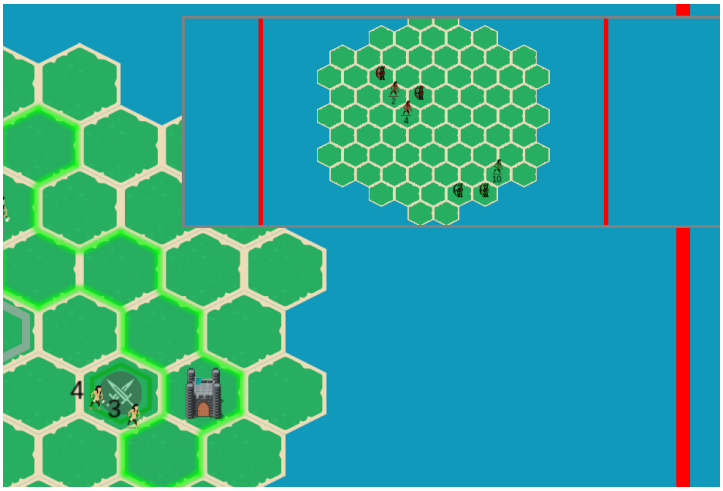

Рисунок 3.23 – Запуск режиму битви

Мета гри полягає в тому, щоб гравці змагалися між собою за контроль над територією та ресурсами на ігровому полі. Гравці створюють свої армії, збирають ресурси, займають нові території та здобувають перевагу над іншими гравцями. Битви між арміями відбуваються на ігровому полі, де кожен гравець намагається знищити армію противника та зберегти свою.

Мета гри полягає в досягненні переваги над іншими гравцями шляхом стратегічного планування, збору ресурсів та розбудови армії. Гра спрямована на розвиток навичок управління ресурсами, стратегічного мислення та планування воєнних операцій.

3.3 Висновки до третього розділу

У розділі було проведено програмну реалізацію стратегічної гри жанру 4X з використанням React.js та Firebase. Було описано компоненти гри,

включаючи класи об’єктної моделі гри. Також були розглянуті правила запуску ігрового серверу та принципи функціонування ігрових юнітів.

В розділі було детально описано обробку подій в режимі лобі і старту гри, елементи управління на карті гри, обробку подій з юнітами та будівлями, а також обробку режиму битви. Було наведено код, що ілюструє роботу кожного етапу гри.

Крім того, було проведено тестування ігрового додатку, що дозволило перевірити правильність роботи програми та виявити наявні помилки. Для тестування гри розроблено автотести.

На основі тестування гри було сформовано низки рекомендацій щодо її покращення.

ВИСНОВКИ

В даній кваліфікаційній роботі була розглянута предметна область розробки стратегічних ігор жанру 4X, проведений аналіз літературних джерел, огляд жанру та його особливостей, історія розвитку та аналіз популярних ігор жанру 4X, тенденції виходу на ринок та основні складності при розробці ігор цього жанру.

Було проаналізовано сучасні тенденції та напрямки розвитку веб-ігор жанру 4X, проведено моделювання основних процесів гри жанру 4X та розглянуто етапи розробки стратегічної гри жанру 4X.

Проведене дослідження показало стале становлення геймдизайну та розкрило принципи розробки ігр, що відносяться до 4X стратегій. Аналіз ринку 4X стратегій показав, що він дуже стабільним і розвивається вже багато десятиліть і не зараз виходить на пік. У цьому сегменті є продукти для всіх гравців: від глибокої і складної в освоєнні пісочниці Stelleris до надпотужної графіки Endless Space II. Від всесвітньо відомої Civilization до Age of Wonders, що втрачає популярність.

Усі проаналізовані ігри стали дуже популярними серед шанувальників жанру 4X, завдяки своїй глибині стратегії, різноманітності унікальних можливостей та цікавому геймплею. Кожна гра має свої особливості та унікальні риси, які відмінно відрізняють їх одна від одної, та пропонують гравцям різні способи розвитку своїх цивілізацій.

Проте, розробка ігор жанру 4X має свої складності та виклики, які пов'язані з реалізацією ігрових механік, балансуванням ресурсів, створенням штучного інтелекту, дизайном ігрового світу та багато іншого. Крім того, кожна нова гра жанру 4X повинна бути достатньо різною та оригінальною, щоб вона могла привернути увагу гравців та відмінитися серед конкурентів на ринку ігор.

Таким чином, розробка ігор жанру 4X є важливим напрямом в індустрії комп'ютерних ігор та вимагає від розробників високої кваліфікації та знань з різних областей, таких як програмування, геймдизайн, графічний дизайн та

багато іншого.

Проведено програмну реалізацію гри з використанням React.js та Firebase. Було описано компоненти гри, включаючи класи об’єктної моделі гри. Також були розглянуті правила запуску ігрового серверу та принципи функціонування ігрових юнітів.

Детально описано обробку подій в режимі лобі і старту гри, елементи управління на карті гри, обробку подій з юнітами та будівлями, а також обробку режиму битви. Було наведено код, що ілюструє роботу кожного етапу гри.

Під час розробки гри проведено тестування, що дозволило перевірити правильність роботи програми та виявити можливі помилки. Для тестування гри розроблено автотести.

В результаті роботи розроблено функціонуючий ігровий додаток, який пройшов успішне тестування та відповідає всім вимогам до стратегічних ігор жанру 4X. Було розроблено графічні компоненти гри, описано класи об’єктної моделі гри, розгорнуто ігровий сервер та проведено тестування ігрового додатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Clinton Keith. Agile Game Development: Build, Play, Repeat (2nd Edition) (AddisonWesley Signature Series (Cohn)) 2nd Edition / Clinton Keith., 2021. – 576 с.
2. Gabor Szauer. Hands-On C++ Game Animation Programming: Learn modern animation techniques from theory to implementation with C++ and OpenGL / Gabor Szauer., 2020. – 368 с.
3. Claudio Torres. Ethereum Game Development Projects: Build your own line of blockchain-based crypto-strategic games / Claudio Torres., 2020. – 447 с.
4. Michael Dawson. Beginning C++ Through Game Programming / Michael Dawson., 2014. – 352 с.
5. Tarn Adams. Procedural Storytelling in Game Design 1st Edition / Tarn Adams, Tanya X. Short., 2019. – 408 с.
6. John Epstein. Hololens and Game Programming Kindle Edition / John Epstein., 2020. – 115 с.
7. Victor Brusca. Video Game UDP Client/Server Design and Implementation: With a cross platform, networked, example game / Victor Brusca, Brian Ree., 2020. – 843 с.
8. Arajo, M., Roque, L., Leblan, S. Modeling Games with Petri Nets. Proceedings of 2009 Dig-ital Games Research Association Conference (DiGRA). Brunel University (2009) Brunel University.
9. Bj¨ork, S., Lundgren, S., Holopainen, J. Game Design Patterns. In: Level Up – Proceedings of Digital Games Research Conference (DiGRA). Utrecht University (2003).
10. Blumenthal, R. Space Invaders: A UML Case Study. Regis University, class notes (2005).
11. Bura, S. A Game Grammar. [Електронний ресурс] // Режим доступу до ресурсу: http://www.stephanebura.com/diagrams/
12. Brom, C., Abonyi, A.: Petri-Nets for Game Plot. In: Proceedings of AISB,

vol. 3 (2006).
13. Burkart, P.: Discovering a lexicon for video games: New research on structured vocabu-laries. International Digital Media and Arts Association Journal 2(1), 18– 24 (2005).
14. Valve Corporation [Електронний ресурс] // Valve. – 2023. – Режим доступу до ресурсу: https://www.valvesoftware.com/ru/.
15. Bethesda Corporation [Електронний ресурс] // Microsoft. – 2023. – Режим доступу до ресурсу: https://bethesda.net/ru/dashboard.
16. Blizzard Corporation [Електронний ресурс] // Activision. – 2023. – Режим доступу до ресурсу: https://www.blizzard.com/ru-ru/.
17. Firaxis Games Developer [Електронний ресурс] // Firaxis Games. – 2023. – Режим доступу до ресурсу: https://firaxis.com/.
18. Hearthstone — стратегічна карткова гра [Електронний ресурс] // Blizzard Entertainment. – 2023. – Режим доступу до ресурсу: https://playhearthstone.com/
19. Hearthstone Battlegrounds [Електронний ресурс] // Blizzard Entertainment. – 2023. – Режим доступу до ресурсу: https://playhearthstone.com/news/23156373.
20. Civilization VI [Електронний ресурс] // 2k Games and Firaxis Games. – 2023. – Режим доступу до ресурсу: https://www.civilization.com/.
21. War games shed light on real-world strategies [Електронний ресурс] // theconversation. – 2019. – Режим доступу до ресурсу: https://theconversation.com/wargames-shed-light-on-real-world-strategies-113631.
22. The Curious Case of 4X Games, Efficiency Engines, and Missing Strategic Gambits [Електронний ресурс] // big-game-theory.com. – 2023. – Режим доступу до ресурсу: http://www.big-game-theory.com/2020/01/the-curious-case-of-4X-games-efficiency.html.
23. Dota Underlords [Електронний ресурс] // Valve. – 2023. – Режим доступу до ресурсу: https://www.underlords.com/.

24. Андерсон К., Кейді-Лі Д., Карре С., Менґерт Г. Створення персонажів для індустрії розваг. Дизайн персонажів у анімації, ілюстрації та відеоіграх. – К.: ArtHuss. – 2023. – 306 с.
25. Костер Р. Теорія розваг для ігрового дизайну. – К.: ArtHuss. – 2023. – 228 с.

ДОДАТКИ

Додаток А

Програмні коди основних модулів

Програмний код модуля joinLobby.ts

```
// Next.js API route support: https://nextjs.org/docs/api-routes/introduction
import { map, toPairs } from 'lodash'

import { colors } from 'src/consts/colors'
import { ServerFirebaseManager } from 'src/server/ServerFirebaseManager'
import { getInitGameInfo } from 'src/store/dataProvider/StrategySetup'
import { PlayerInfo } from 'src/store/dataProvider/types'
import { Dictionary } from 'src/types/generic'

import type { NextApiRequest, NextApiResponse } from 'next'

type Data = {
  gameId: string | null
}

const handler = async (req: NextApiRequest, res: NextApiResponse<Data>): Promise<void> => {
  const fireBaseManager = new ServerFirebaseManager()
  await fireBaseManager.init()
  const players = await fireBaseManager.fsReadOnce<Dictionary<{ name?: string; ready: boolean }>>(
    'lobby/players/',
  )
  const playersArray: Array<PlayerInfo> = map(toPairs(players), ([uid, player], i) => ({
    uid,
```

```
    name: player.name ?? `GhostPlayer-${i}`,
    isMaster: i === 0,
    color: colors[i],
    isOnline: true,
    isAI: false,
  }))
  const gameData = await getInitGameInfo(playersArray)
  const gameIdRef = await fireBaseManager.fPush('games', gameData)
  const gameId = gameIdRef.key
  res.status(200).json({ gameId })
}

export default handler
```

Програмний код модуля PlayerInfoStore.ts

```
import _ from 'lodash'
import { action, makeObservable, observable } from 'mobx'
import { nanoid } from 'nanoid'

import type { PlayerInfo } from 'src/store/dataProvider/types'

export default class PlayerInfoStore {
  id: string
  color: string
  @observable isOnline: boolean

  constructor(color: string) {
    makeObservable(this)
    _.bindAll(this, [])
```

```
    this.id = nanoid()
    this.color = color
  }

  @action setID(uid: string): void {
    this.id = uid
  }

  @action setColor(color: string): void {
    this.color = color
  }

  @action setIsOnline(isOnline: boolean): void {
    this.isOnline = isOnline
  }

  data(): PlayerInfo {
    return {
      color: this.color,
      isOnline: this.isOnline,
      uid: this.id,
    }
  }
}
```

Програмний код модуля PlayerStore.ts

```
import _, { sum } from 'lodash'
import { action, computed, IObservableArray, makeObservable, observable,
ObservableSet } from 'mobx'
```

```
import { BUILDING_TYPES } from 'src/consts/building'
import ArmyStore from 'src/store/ArmyStore'
import BuildingStore from 'src/store/BuildingStore'
import { CellStore } from 'src/store/CellStore'
import type { PlayerInfo } from 'src/store/dataProvider/types'
import type StrategyStore from 'src/store/strategy/StrategyStore'
import type { Dictionary } from 'src/types/generic'

export default class PlayerStore {
  @observable id: string // uid
  @observable isOnline: boolean
  isAI: boolean
  color: string

  armies: IObservableArray<ArmyStore>
  buildings: IObservableArray<BuildingStore>
  ownedCells: ObservableSet<string>

  @observable gold: number

  // refs
  strategy: StrategyStore

  constructor(strategyStore: StrategyStore, playerId: string, playerInfo: PlayerInfo)
{
    makeObservable(this)
    _.bindAll(this, [])
    this.id = playerId
    this.strategy = strategyStore
```

```
  this.color = playerInfo.color
  this.isOnline = !!playerInfo.isOnline
  this.gold = 20
  this.armies = observable.array([], { deep: false })
  this.buildings = observable.array([], { deep: false })
  this.ownedCells = observable.set([], { deep: false })
  this.isAI = playerInfo.isAI ?? false
}

@action setID(id: string): void {
  this.id = id
}

@computed get goldIncome(): number {
  // return 3 + this.ownedCells.size
  // [...this.ownedCells.values()].filter()
  const unitsSupport = -sum(this.armies.map(army => army.units.length))
  const buildingsPlus = sum(
    this.buildings.map(building =>
      building.buildingType === BUILDING_TYPES.HOUSE_SMALL ? 2 : 0,
    ),
  )
  const capitalInc = 4
  return unitsSupport + buildingsPlus + capitalInc
}

getGoldIncomeDetailed(): Dictionary<number> {
  console.log('getGoldIncomeDetailed')
  // return 3 + this.ownedCells.size
  // [...this.ownedCells.values()].filter()
```

```
  const unitsSupport = -sum(this.armies.map(army => army.units.length))
  const buildingsIncome = sum(
    this.buildings.map(building =>
      building.buildingType === BUILDING_TYPES.HOUSE_SMALL ? 2 : 0,
    ),
  )
  const capital = 4
  return {
    unitsSupport,
    buildingsIncome,
    capital,
  }
}

@action
addArmy(army: ArmyStore) {
  this.armies.push(army)
}

@action
removeArmy(army: ArmyStore) {
  this.armies.remove(army)
}

@action setGold(gold: number): void {
  this.gold = gold
}

@action applyIncome(): void {
  this.setGold(this.gold + this.goldIncome)
```

```
  }

  @action setIsOnline(isOnline: boolean): void {
    this.isOnline = isOnline
  }

  @action addCell(cell: CellStore): void {
    this.ownedCells.add(cell.hex.id)
  }

  @action removeCell(cell: CellStore): void {
    this.ownedCells.delete(cell.hex.id)
  }

  update(info: PlayerInfo): void {
    this.setIsOnline(!!info.isOnline)
  }

  onTick() {
    this.applyIncome()
  }
}
```

Програмний код для автоматичного тестування

```
const PlayerStore = require('./PlayerStore')
describe('PlayerStore', () => {
  it('should correctly calculate gold income with default values', () => {
    const player = new PlayerStore({})
    expect(player.goldIncome).toBe(3)
  })
  it('should correctly calculate gold income with buildings and armies', () => {
    const player = new PlayerStore({})
    player.buildings.push({ buildingType: 'HOUSE_SMALL' })
    player.armies.push({ units: [{}] })
    expect(player.goldIncome).toBe(5)
  })

  it('should correctly calculate gold income with negative values', () => {
    const player = new PlayerStore({})
    player.armies.push({ units: [{}, {}] })
    expect(player.goldIncome).toBe(-2)
  })
})
```

Тестування методу getGoldIncomeDetailed():

```
const PlayerStore = require('./PlayerStore')
describe('PlayerStore', () => {
  it('should return correct gold income details with default values', () => {
    const player = new PlayerStore({})
    expect(player.getGoldIncomeDetailed()).toEqual({
      unitsSupport: 0,
      buildingsIncome: 0,
      capital: 4,
```

```
  })  })
  it('should return correct gold income details with buildings and armies', () =>
{
  const player = new PlayerStore({})
  player.buildings.push({ buildingType: 'HOUSE_SMALL' })
  player.armies.push({ units: [{}] })
  expect(player.getGoldIncomeDetailed()).toEqual({
   unitsSupport: -1,
   buildingsIncome: 2,
   capital: 4,
  })  })
  it('should return correct gold income details with negative values', () => {
  const player = new PlayerStore({})
  player.armies.push({ units: [{}, {}] })
  expect(player.getGoldIncomeDetailed()).toEqual({
   unitsSupport: -2,
   buildingsIncome: 0,
   capital: 4,
  })  })  })
```

Тестування методів addArmy() та removeArmy():

```
const PlayerStore = require('./PlayerStore')
const ArmyStore = require('./ArmyStore')
describe('PlayerStore', () => {
  it('should add and remove armies correctly', () => {
  const player = new PlayerStore({})
  const army1 = new ArmyStore({})
  const army2 = new ArmyStore({})
  expect(player.armies).toHaveLength(0)
  player.addArmy(army1)
  expect(player.armies).toHaveLength(1)
```

```
        expect(player.armies).toContain(army1)
        player.addArmy(army2)
        expect(player.armies).toHaveLength(2)
        expect(player.armies).toContain(army2)
        player.removeArmy(army1)
        expect(player.armies).toHaveLength(1)
        expect(player.armies).not.toContain(army1)
        expect(player.armies).toContain(army2)
        player.removeArmy(army2)
        expect(player.armies).toHaveLength(0)
        expect(player.armies).not.toContain(army1)
        expect(player.armies).not.toContain(army2)
      })
    })
```

Програмний код алгоритму A*

```
import Heap from 'heap'
import { minBy } from 'lodash'

import Hex from 'src/hexes/Hex'
import { GridStore } from 'src/store/GridStore'
import { PNode } from 'src/store/pathfind/PNode'
import { Dictionary } from 'src/types/generic'

class FullHeap {
  heap: Heap<string>
  support: Dictionary<PNode>
```

```
constructor() {
  this.support = {}
  this.heap = new Heap((nodeAId: string, nodeBId: string) => {
    const nodeA = this.support[nodeAId] || { bestGuessDistance: () => 999999 }
    const nodeB = this.support[nodeBId] || { bestGuessDistance: () => 999999 }
    return nodeA.bestGuessDistance() - nodeB.bestGuessDistance()
  })
}

push(node: PNode) {
  const id = node.hex.id
  const isNew = this.support[id] == null
  this.support[id] = node
  if (isNew) {
    this.heap.push(id)
  } else {
    this.heap.updateItem(id)
  }
}

pop(): PNode | undefined {
  const id = this.heap.pop()
  if (id && this.support[id]) {
    const node = this.support[id]
    delete this.support[id]
    return node
  }
  return undefined
}
```

```
  size() {
    return this.heap.size()
  }
}

export const findPathAStar = (
  grid: GridStore,
  blocked: Dictionary<string>,
  from: Hex,
  end: Hex,
  ownColor: string | null = null,
  giveClosest: boolean = false,
): Array<Hex> => {
  if (from.equals(end)) {
    return []
  }

  const openHeap = new FullHeap()

  // const closedHexes = {}
  const visitedNodes: Dictionary<PNode> = {}
  const startNode = new PNode(null, from, 1, from.getDistance(end))
  openHeap.push(startNode)
  while (openHeap.size() > 0) {
    // Get the item with the lowest score (current + heuristic).
    let current: PNode | undefined = openHeap.pop()
    if (current == null) {
      break
    }
```

```
    // SUCCESS: If this is where we're going, backtrack and return the path.
    if (
      current.hex.equals(end)
      // ||(ownColor != null && blocked[current.hex.id] != null &&
blocked[current.hex.id] !== ownColor)
    ) {
      const path = []
      while (current.parent) {
        path.push(current)
        current = current.parent
      }
      return path.map(x => x.hex).reverse()
    }

    // Close the hex as processed.
    // closedHexes[current.hex.id] = current

    const neighbors = grid.getNeighbours(current.hex)
    neighbors.forEach(n => {
      // Make sure the neighbor is not blocked and that we haven't already processed
it.
      // but if the Hex is our end - its ok
      if (
        blocked[n.id] != null &&
        blocked[n.id] === ownColor
        // !n.equals(end)
        // && (ownColor == null || blocked[n.id] === ownColor)
      ) {
        return
      }
```

```
      if (current == null) {
        return
      }
      // Get the total cost of going to this neighbor.
      const thisHexCost = 1
      const g = current.costSoFar + thisHexCost
      const visited: PNode | null = visitedNodes[n.id]

      // Is it cheaper the previously best path to get here?
      if (visited == null || g < visited.costSoFar) {
        const h = n.getDistance(end)
        const nNode = new PNode(current, n, g, h)
        visitedNodes[n.id] = new PNode(current, n, g, h)
        openHeap.push(nNode)
      }
    })
  }
  if (!giveClosest) {
    return []
  }
  const closest: PNode | undefined = minBy(
    Object.values(visitedNodes),
    (node: PNode | null) =>
      // node.bestGuessDistance()
      node?.simpleDistanceToTarget ?? 0,
  )
  if (closest == null || startNode.simpleDistanceToTarget -
closest.simpleDistanceToTarget <= 0) {
    return []
```

```
  }
  const path = []
  let current: PNode = closest
  while (current.parent) {
    path.push(current)
    current = current.parent
  }
  return path.map(x => x.hex).reverse()
}
```

Додаток Б

Приклад структури БД у json-файлі


```
{
 "games": {
  "1": {
   "actions": {
    "-NTZorCrXRB_siIJtnEo": {
     "action": "move",
     "armyId": "HmrfSR6iUyjcCmuHiOX77",
     "player": "green"
    },
    "-NTZos0K_U02-SlzG5FM": {
     "action": "move",
     "armyId": "HmrfSR6iUyjcCmuHiOX77",
     "path": [
      "-1,2,-1"
     ],
     "player": "green"
    },
    "-NTZot1NEaMbMhvwEGc1": {
     "action": "turnEnd"
    },
    "-NTZot1aIVthlRxv2Qwk": {
     "action": "turnEnd"
    },
    "-NTZot1dKrtegkViiu43": {
     "action": "turnEnd"
    },
```

```
      "player": "green"
    },
    "-NTZp5hzsCRdvw2UCGaj": {
      "action": "move",
      "armyId": "KCiD-9NsWZos1jrZ8tFjU",
      "path": [
        "-2,2,0"
      ],
      "player": "green"
    },
    "-NTZp64LvFHQFwCnlDFi": {
      "action": "turnEnd"
    },
    "-NTZp7diHev4GglEGtI9": {
      "action": "productionStartAction",
      "buildingId": "Ol6Rkf2fLxMNcbUnzf7OP",
      "hex": "-2,2,0",
      "type": "bandit"
    },
    "-NTZp8DniW646pf4Hu7o": {
      "action": "turnEnd"
    },
    "-NTZp8Doq5Klba26x2mo": {
      "action": "turnEnd"
    },
    "-NTZp8KCmY8TwkGs123o": {
      "action": "turnEnd"
    }
  }
}
```