БАКАЛАВРСЬКА РОБОТА

БР. ІП - 19.00.00.000 ПЗ

Група ІП-21-2

Баранецький Мартин

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Баранецький Мартин Миронович

(прізвище, ім'я, по батькові)

УДК 004

(індекс)

БАКАЛАВРСЬКА РОБОТА

Реалізація ігрового застосунку з регульованими рівнями складності

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Здобувач освітнього рівня Баранецький М.М.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Дмитрик Тарас Богданович, асистент

(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2025 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Баранецькому Мартину Мироновичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Реалізація ігрового застосунку з регульованими рівнями складності”

керівник проекту (роботи) Дмитрик Т.Б.

затверджені наказом закладу вищої освіти від “ 28 ” квітня 2025 р. № 264/7

2. Строк подання студентом проекту (роботи) 09 червня 2025 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз і стан застосування інформаційних технологій для побудови ігрових застосунків
2. Розробка специфікації вимог до програмної реалізації ігрового застосунку
3. Алгоритмічна реалізація ігрового застосунку з регульованими рівнями складності
4. Реалізація інтерфейсу ігрового застосунку головоломки з регульованими рівнями складності
5. Опис алгоритму пошуку рішення

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

1. Інтерфейс гри Klotski (рис. 1.1)
2. Екрани гри Unblock Me (рис. 1.2)
3. Різні представлення гри 15-puzzle (рис. 1.3)
4. Основні та додаткові функціональні вимоги (рис. 1.4)
5. Діаграма класів (рис. 2.1)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **28 квітня 2025 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз і стан застосування інформаційних технологій для побудови ігрових застосунків	01.05.2025	виконано
2	Розробка специфікації вимог до програмної реалізації ігрового застосунку	10.05.2025	виконано
3	Алгоритмічна реалізація ігрового застосунку з регульованими рівнями складності	20.05.2025	виконано
4	Реалізація інтерфейсу ігрового застосунку головоломки з регульованими рівнями складності	30.05.2025	виконано
5	Опис алгоритму пошуку рішення	03.06.2025	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	09.06.2025	виконано

Студент – дипломник _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 75 сторінок, 20 рисунків, список використаних джерел із 30 найменуваннями.

Метою даної роботи є розробка ігрового застосунку головоломки з динамічно регульованими рівнями складності, що базується на сучасних програмних технологіях і методах системного проєктування.

Об'єкт дослідження - процес розробки ігрових застосунків з головоломками, які мають адаптивну складність.

Предмет дослідження - алгоритми, методи і технічні засоби реалізації програмного забезпечення для ігрової головоломки з регульованими рівнями складності.

У першому розділі розглянуто сучасний стан застосування ІТ у створенні ігрових застосунків, проведено аналіз аналогів, сформульовано вимоги до системи.

У другому розділі подано опис алгоритмічної та архітектурної реалізації, використано засоби візуального моделювання.

Третій розділ присвячено практичній реалізації графічного інтерфейсу, створенню логіки гри та проведенню тестування.

Висновок: результати дослідження демонструють доцільність інтеграції методів програмної інженерії в розробку сучасних логічних ігор, що сприяють розвитку мислення та забезпечують гнучкий геймплей.

КЛЮЧОВІ СЛОВА: ЛОГІЧНА ГРА, ГОЛОВОЛОМКА, ПРОГРАМНА ІНЖЕНЕРІЯ, РІВНІ СКЛАДНОСТІ, ІГРОВИЙ ЗАСТОСУНОК, ІНТЕРФЕЙС КОРИСТУВАЧА, АЛГОРИТМ ПОШУКУ, UML-ДІАГРАМИ, АДАПТИВНІСТЬ, ТЕСТУВАННЯ

ANNOTATION

This Bachelor's thesis comprises 75 pages, 20 figures, and a list of 30 references.

The goal of this work is to develop a puzzle game application with dynamically adjustable difficulty levels, based on modern software technologies and system design methods.

The object of the research is the process of developing puzzle game applications with adaptive difficulty.

The subject of the research includes algorithms, methods, and technical tools for implementing software for a puzzle game with adjustable difficulty levels.

The first chapter reviews the current state of IT application in game development, analyzes existing analogs, and formulates system requirements.

The second chapter provides a description of the algorithmic and architectural implementation, utilizing visual modeling tools.

The third chapter is dedicated to the practical implementation of the graphical user interface, the creation of game logic, and testing.

Conclusion: The research results demonstrate the feasibility of integrating software engineering methods into the development of modern logic games, which promotes cognitive development and provides flexible gameplay..

KEYWORDS: LOGIC GAME, PUZZLE, SOFTWARE ENGINEERING, DIFFICULTY LEVELS, GAME APPLICATION, USER INTERFACE, SEARCH ALGORITHM, UML DIAGRAMS, ADAPTABILITY, TESTING

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

BFS – Breadth-First Search (Пошук в ширину)

A* – A* Search Algorithm (Алгоритм пошуку A*)

GUI – Graphical User Interface (Графічний інтерфейс користувача)

BGM – Background Music (Фонова музика)

JSON – JavaScript Object Notation (Нотація об'єктів JavaScript)

OOP – Object-Oriented Programming (Об'єктно-орієнтоване програмування)

DFD – Data Flow Diagram (Діаграма потоків даних)

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій спостерігається зростання інтересу до інтелектуальних ігрових застосунків, що не лише розважають, а й розвивають логічне мислення та когнітивні здібності користувачів. Головоломки з регульованими рівнями складності набувають особливого значення, оскільки дозволяють адаптувати геймплей до індивідуального рівня підготовки користувача. Незважаючи на наявність великої кількості схожих програмних продуктів, актуальним залишається питання створення ефективної, гнучкої та оптимізованої системи, яка поєднує зручний інтерфейс, чітку логіку та адаптивність складності. Саме тому розробка ігрового застосунку, що враховує ці вимоги, є актуальною задачею як з точки зору програмної інженерії, так і освітніх або тренувальних цілей.

У сучасному цифровому суспільстві інформаційні технології проникають у всі сфери людської діяльності, включаючи розважальну, освітню та когнітивну. Особливо динамічно розвивається галузь створення інтерактивних ігрових застосунків, які не лише виконують функцію розваги, але й формують інтелектуальні та логічні навички користувачів. Важливу роль у цьому відіграють комп’ютерні головоломки — різновид логічних ігор, що стимулюють критичне мислення, просторове уявлення, уважність та вміння приймати рішення в умовах обмежень.

Актуальність роботи

Зростаюча потреба у персоналізованих цифрових продуктах актуалізує питання створення гнучких і адаптивних програмних систем. Серед таких систем особливе місце посідають ігрові застосунки з регульованими рівнями складності, які дозволяють забезпечити індивідуальний підхід до кожного користувача — від новачка до досвідченого гравця. Це сприяє не лише

розширенню аудиторії продукту, а й підвищенню зацікавленості, залученості та мотивації гравців.

Попри наявність великої кількості ігрових головоломок на ринку цифрових застосунків, багато з них мають низку обмежень: відсутність адаптивності, недостатньо опрацьовану логіку взаємодії, нераціональне використання ресурсів або складність підтримки в процесі розширення функціоналу. Тому виникає потреба у створенні нового підходу до проєктування і реалізації таких систем — з урахуванням сучасних принципів програмної інженерії, засобів візуального моделювання, гнучкого управління станами інтерфейсу та ефективних алгоритмів обробки ігрових ситуацій.

Метою даної роботи є розробка ігрового застосунку головоломки з динамічно регульованими рівнями складності, що базується на сучасних програмних технологіях і методах системного проєктування.

Основними завданнями дослідження виступають:

- аналіз існуючих ігрових головоломок,
- формалізація вимог до майбутнього застосунку,
- побудова архітектури програмного забезпечення,
- розробка користувацького інтерфейсу,
- реалізація алгоритму пошуку рішень,
- тестування та оцінка ефективності створеної системи.

Об’єкт дослідження - процес розробки ігрових застосунків з головоломками, які мають адаптивну складність.

Предмет дослідження - алгоритми, методи і технічні засоби реалізації програмного забезпечення для ігрової головоломки з регульованими рівнями складності.

Методи дослідження:

- Аналіз та порівняння ігрових додатків;
- Методи об'єктно-орієнтованого проєктування;

- Проєктування діаграм (UML: діаграми класів, варіантів використання, послідовності);

- Метод розробки інтерфейсів користувача;

- Тестування програмного забезпечення.

У роботі розглядається процес створення ігрового застосунку логічної головоломки з регульованими рівнями складності. Основну увагу приділено аналізу існуючих аналогів, формуванню вимог до програмного продукту, побудові архітектури та реалізації інтерфейсу. Важливими аспектами також стали оптимізація алгоритмів пошуку рішень та забезпечення адаптивності до потреб користувача.

Робота структурована таким чином, щоб відобразити весь цикл створення програмного продукту — від теоретичного обґрунтування до практичної реалізації та перевірки працездатності.

Практичне застосування

Отримані результати засвідчують доцільність застосування системного підходу до розробки ігрових застосунків із можливістю масштабування, адаптації до потреб користувача та оптимізації обчислювальних процесів. Розроблений застосунок може бути використаний як основа для подальших досліджень, освітніх цілей або вдосконалення комерційних програмних рішень у сфері логічних ігор.

Наукова новизна

У роботі запропоновано архітектурне рішення для створення ігрової головоломки з можливістю регулювання складності, що дозволяє користувачам з різним рівнем підготовки адаптувати ігровий процес під свої потреби, а також реалізовано алгоритм оптимізації пошуку рішення.

Бакалаврська робота містить 75 сторінок, 25 рисунків, 3 розділи список використаних джерел із 35 найменуваннями.

РОЗДІЛ 1. АНАЛІЗ І СТАН ЗАСТОСУВАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ДЛЯ ПОБУДОВИ ІГРОВИХ ЗАСТОСУНКІВ

1.1. Опис ігрового застосунку з регульованими рівнями складності на прикладі головоломки

Головоломка "Година пік" ("Rush Hour") є представником класу ковзних блокових головоломок (sliding block puzzles). Вона була розроблена японським винахідником Нобом Йошіґахарою (Nob Yoshigahara) у 1970-х роках і наразі комерціалізується компанією ThinkFun. Гра розгортається на двовимірному полі розмірністю 6×6 клітинок, що концептуально імітує умови паркувального майданчика. Виїзд з майданчика розташовано у правому кінці третього ряду сітки. Початкові конфігурації розміщення транспортних засобів, що є елементами головоломки, визначаються відповідними картками завдань.

Транспортні засоби представлені моделями, що займають на полі 1×2 або 1×3 клітинки. Ці елементи можуть пересуватися виключно вздовж пазів сітки, що відповідає їхній орієнтації (горизонтальній або вертикальній), тобто обертання блоків заборонено.

Мета гри полягає у знаходженні послідовності допустимих переміщень транспортних засобів, яка дозволить цільовому автомобілю (як правило, ідентифікованому за кольором) досягти та виїхати за межі ігрового поля через визначений виїзд. З точки зору обчислювальної складності, узагальнена версія головоломки "Година пік" на полі n×n є PSPACE-повною задачею.

Для розуміння терміну "PSPACE-повна задача" потрібно визначити основи теорії обчислювальної складності. Розглянемо його покроково:

1. Клас складності PSPACE.

Цей клас включає задачі розпізнавання (decision problems), які можуть бути розв'язані детермінованою машиною Тюрінга, використовуючи обсяг

пам'яті (простору), що є поліномом від розміру вхідних даних. Кількість часу, необхідного для розв'язання, при цьому може бути довільною (хоча вона, як правило, експоненційна від простору, а отже, також експоненційна від розміру входу). Головний критерій для PSPACE – це саме поліноміальне використання пам'яті.

2. PSPACE-складна задача (PSPACE-hard).

Задача H є PSPACE-складною, якщо будь-яка задача з класу PSPACE може бути зведена до H за поліноміальний час. Зведення означає, що існує поліноміальний за часом алгоритм, який перетворює будь-який примірник задачі A з PSPACE на примірник задачі H таким чином, що відповідь для A є "так" тоді і тільки тоді, коли відповідь для H є "так". Якщо ви можете ефективно (за поліноміальний час) розв'язати PSPACE-складну задачу, то ви можете ефективно розв'язати будь-яку задачу з PSPACE. Це означає, що PSPACE-складні задачі є "принаймні такими ж складними", як і будь-які задачі в PSPACE.

3. PSPACE-повна задача (PSPACE-complete).

Задача є PSPACE-повною, якщо вона задовольняє двом умовам:

- Вона належить до класу PSPACE (тобто її можна розв'язати з використанням поліноміальної пам'яті).

- Вона є PSPACE-складною.

Таким чином, PSPACE-повні задачі є "найскладнішими" задачами в класі PSPACE. Вони слугують своєрідними "представниками" складності всього класу.

Коли задача доводиться як PSPACE-повна, це є сильним свідченням того, що для неї, ймовірно, не існує алгоритму, який працює за поліноміальний час (як для задач класу P) або навіть за поліноміальний час для перевірки запропонованого рішення (як для задач класу NP, якщо P □= NP). Це означає, що для розв'язання таких задач в загальному випадку, ймовірно, буде потрібен експоненційний час (від розміру вхідних даних),

хоча для окремих примірників або зі специфічними обмеженнями можуть існувати більш ефективні підходи.

Приклади PSPACE-повних задач:

- Узагальнена головоломка "Година пік" (на полі n×n).

- Проблема виконання кванторних булевих формул (Quantified Boolean Formulas, QBF) – часто вважається канонічною PSPACE-повною задачею.

- Узагальнені ігри на дошці (наприклад, шахи, шашки, Го) на полі n×n, де питання полягає в тому, чи має перший гравець виграшну стратегію.

Доведення PSPACE-повноти задачі зазвичай здійснюється шляхом зведення відомої PSPACE-повної задачі до нової задачі. У випадку "Пікової години", дослідники показали, як сконструювати екземпляр "Пікової години", який імітує виконання кванторної булевої формули, доводячи таким чином її PSPACE-складність. А оскільки очевидно, що "Година пік" може бути розв'язана з поліноміальним обсягом пам'яті (наприклад, за допомогою пошуку в станах гри), вона належить до PSPACE, що робить її PSPACE-повною.

Традиційна фізична реалізація головоломки вимагає ручного налаштування ігрового поля перед початком кожного нового завдання або при необхідності перезапуску. Цей процес може бути незручним та обмежувати доступність гри. З метою подолання зазначених логістичних та операційних обмежень, у рамках даного проєкту була розроблена цифрова настільна версія головоломки "Година пік". Ця імплементація дозволяє користувачам взаємодіяти з головоломкою без необхідності використання фізичних компонентів, а ініціалізація нового завдання виконується за допомогою мінімальної кількості дій інтерфейсу. Окрім базової візуалізації та симуляції ігрового процесу, розроблена програмна система включає додаткові функціональні можливості, відсутні у фізичному варіанті, зокрема: автоматизований підрахунок кількості ходів для оцінки ефективності вирішення, систему надання підказок для допомоги гравцям у складних

ситуаціях, а також потенційну можливість імплементації алгоритмів пошуку оптимального розв'язку.

1.2. Представлення мети розробки комп'ютерної реалізації головоломки

Основною метою даного дослідження є розробка віртуальної версії настільної гри-головоломки "Година пік". Актуальність створення комп'ютерної реалізації зумовлена необхідністю усунення логістичних та операційних обмежень, властивих її фізичному аналогу, зокрема потреби у транспортуванні набору фізичних компонентів та виконанні ручного налаштування ігрового поля для кожної нової початкової конфігурації чи перезапуску завдання.

Розроблювана програмна система забезпечує симуляцію ігрового процесу "Години пік" в цифровому середовищі персонального комп'ютера, відтворюючи базовий функціонал фізичної версії, проте з покращеною зручністю взаємодії. Ініціалізація нових завдань та повернення до початкового стану (скидання) здійснюється ефективно через графічний інтерфейс користувача за допомогою мінімальної кількості дій. Переміщення віртуальних моделей транспортних засобів по ігровому полю реалізовано за допомогою стандартних маніпуляцій введення, таких як кліки та перетягування (drag-and-drop), що спрощує ігровий процес.

На додаток до базової симуляції, в програмній реалізації імплементовано розширений функціонал, який відсутній у фізичній версії. Зокрема, реалізовано систему кількісної оцінки ефективності вирішення завдання (система оцінювання), що може базуватися, наприклад, на мінімізації кількості ходів. Також інтегровано функцію надання підказок (hinting system), яка може допомагати гравцеві у пошуку наступних допустимих переміщень або вказувати на кроки, що ведуть до розв'язку. Цей

додатковий функціонал спрямований на покращення досвіду користувача та надання аналітичних інструментів для оцінки прогресу гравця.

Розробка даної програмної системи здійснена з використанням мови програмування Python та графічної бібліотеки Pygame, що є типовим вибором для створення 2D-симуляцій та прототипів ігор. Більш детальний опис архітектури програми, використаних алгоритмів представлення стану гри та обробки взаємодії користувача, а також специфіка реалізації додаткових функцій буде представлено та обговорено в наступних розділах даної роботи.

1.3. Аналіз ігрових додатків блокових головоломок з різними рівнями складності

По функціоналу та механіці, "Година пік" належить до широкого класу ковзних блокових головоломок (sliding block puzzles). Існує багато ігор, які поділяють ключові принципи "Пікової години", а саме: переміщення блоків по сітці з метою досягнення певної конфігурації або звільнення шляху для цільового елемента.

Розглянемо кілька ігор, схожих за функціоналом.

1.3.1. Головоломка Klotski

Клотскі (Klotski) - це одна з найвідоміших класичних ковзних блокових головоломок. Вона, як правило, грається на полі 5×4 і включає блоки різних розмірів (1x1, 1x2, 2x1) та один великий блок 2x2 (часто називається "Червоний блок" або "Huāróng Dào" 花容道 у китайській версії).

Мета гри – перемістити великий блок 2x2 до певного виходу, зазвичай розташованого внизу поля. Механіка руху блоків (тільки горизонтально або вертикально без обертання) і принцип звільнення шляху дуже схожі на "Годину пік".

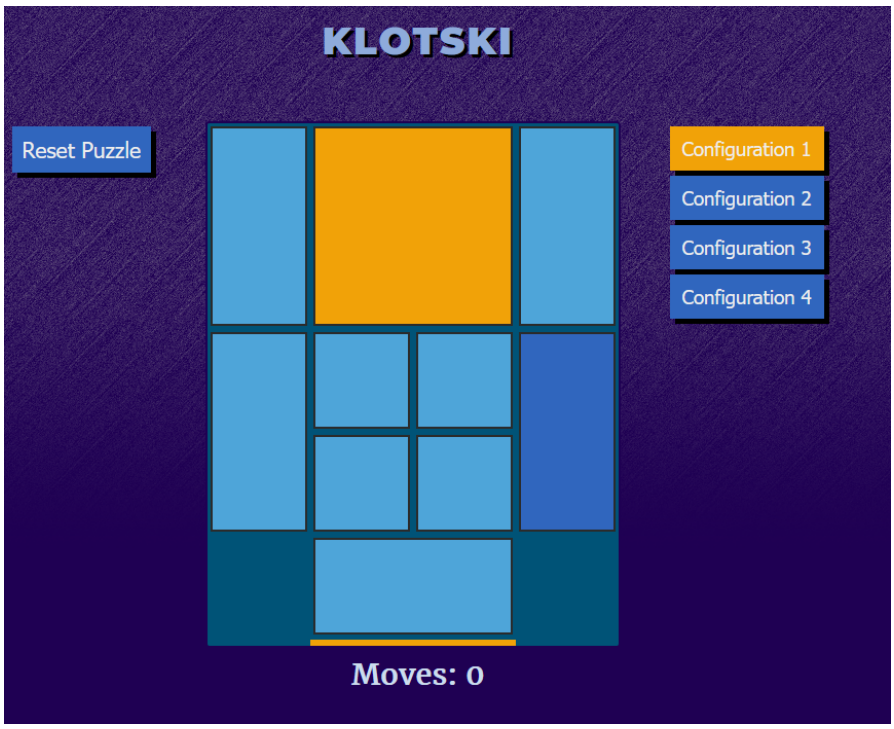

Рисунок 1.1 – Інтерфейс гри Klotski

Типова головоломка клотскі складається з:

- Дошки або ігрового поля. Зазвичай має форму прямокутника розміром 5×4 клітинки, з вирізом або виходом у нижній частині.

- Набору блоків. Блоки мають прямокутну або квадратну форму і різні розміри, які кратні розміру однієї клітинки. Стандартний набір блоків часто включає:

- Один великий квадратний блок розміром 2×2 клітинки (це цільовий блок, який потрібно вивести).

- Два вертикальні прямокутні блоки розміром 1×2 клітинки.

- Два горизонтальні прямокутні блоки розміром 2×1 клітинки.

- Чотири маленькі квадратні блоки розміром 1×1 клітинка.

Клотскі, як і "Година пік", є чудовим прикладом задачі на пошук шляху у просторі станів, де кожен стан визначається унікальним розташуванням усіх блоків на дошці. Складність головоломки може варіюватися залежно від початкової конфігурації блоків. Деякі завдання вимагають значної кількості ходів і ретельного планування

1.3.2. Гра L'âne rouge

L'âne rouge ("Червоний вислюк") — це відома ковзна блокова головоломка, яка є однією з найпопулярніших варіацій класичної головоломки Клотскі. Вона поділяє основні принципи та механіку з Клотскі та "Годиною пік".

Зазвичай, гра L'âne rouge представлена на ігровому полі розміром 5×4 клітинки з виходом (шириною у дві клітинки) у нижній частині дошки. Набір ігрових елементів (блоків) є дуже схожим на стандартний Клотскі:

- "Червоний вислюк": Це цільовий квадратний блок розміром 2×2 клітинки, який, зазвичай, має червоний колір (звідси й назва головоломки). Саме його потрібно вивести з ігрового поля.

- Інші блоки: Набір включає різноманітні прямокутні та квадратні блоки меншого розміру, які перешкоджають руху "Червоного вислюка". Типовий набір включає:

- Два вертикальні блоки розміром 1×2.

- Два горизонтальні блоки розміром 2×1.

- Чотири маленькі квадратні блоки розміром 1×1.

Правила L'âne rouge ідентичні правилам Клотскі. Блоки можна пересувати (ковзати) лише вільними клітинками на дошці. Рух дозволений лише вздовж осі блоку: горизонтальні блоки рухаються тільки по горизонталі, а вертикальні та квадратні — тільки по вертикалі та горизонталі відповідно (якщо є вільне місце).

Головне завдання гравця в L'âne rouge — знайти послідовність дозволених переміщень інших блоків на полі, яка дозволить цільовому блоку "Червоний вислюк" (2x2) досягти виходу в нижній частині дошки і вийти з неї.

L'âne rouge часто вважається специфічною версією Клотскі через стандартний набір блоків та розмір поля. Вона дуже схожа на "Годину пік" за своєю основною механікою (ковзні блоки, рух без обертання, звільнення

шляху для цільового блоку на сітці), але відрізняється розмірами ігрового поля, формами та розмірами інших блоків, а також розташуванням виходу. Наприклад, у "Годині пік" поле 6x6, вихід збоку, і блоки мають розміри 1x2 або 1x3, тоді як у L'âne rouge/Клотскі поле 5x4, вихід знизу, і використовуються блоки 1x1, 1x2, 2x1, 2x2.

1.3.3. Гра Unblock Me

Unblock Me — це надзвичайно популярна цифрова гра-головоломка, що належить до категорії ковзних блокових головоломок і є одним із найвідоміших сучасних прикладів, функціонально дуже схожих на "Годину пік" та Клотскі. Гра була розроблена компанією Kiragames.

Рисунок 1.2 – Екрани гри Unblock Me

Переважно доступна як мобільний додаток для смартфонів та планшетів (на платформах iOS, Android), хоча існують версії для веб-браузерів та ПК. Ігрове поле представлене у вигляді прямокутної сітки, яка візуально нагадує паркувальний майданчик або вузький прохід. Розмір сітки може варіюватися залежно від конкретного завдання, але концептуально схожий на поле "Години пік".

Ігрові елементи представлені як прямокутні блоки або стилізовані під автомобілі. Вони мають різну довжину (зазвичай займають 2 або 3 клітинки

сітки) і розташовані горизонтально або вертикально. Блоки відрізняються кольором. Один зі спеціальних блоків, зазвичай виділений червоним кольором. Це блок, який гравець має вивести за межі ігрового поля. Спеціальна зона на одному з країв ігрового поля, через яку має виїхати цільовий червоний блок.

Правила гри в Unblock Me дуже прості та відповідають механіці ковзних блокових головоломок:

- Блоки можна пересувати (ковзати) лише по прямій лінії вздовж своєї осі: горизонтальні блоки рухаються тільки горизонтально, а вертикальні блоки — тільки вертикально.

- Блоки не можна обертати.

- Блоки не можуть пересуватися в клітинки, зайняті іншими блоками.

Мета гри полягає у тому, щоб, пересуваючи інші блоки по ігровому полю, звільнити шлях для червоного цільового блоку та вивести його через позначений вихід.

Unblock Me відома величезною бібліотекою головоломок — від кількох тисяч до десятків тисяч завдань, часто згрупованих за рівнями складності (від початкового до експертного). Гра, як правило, оцінює виконання завдання (наприклад, за кількістю ходів, присуджуючи "зірки") та відстежує прогрес гравця.

Завдяки своїй простій, але глибокій механіці та великій кількості завдань, Unblock Me стала одним із найуспішніших прикладів ковзних блокових головоломок у цифровому світі, пропонуючи ігровий досвід, дуже близький до "Година пік".

1.3.4. Головоломка 15-puzzle

П'ятнашки, відома також як 15-puzzle, Slide Puzzle, або узагальнено N-puzzle, є однією з найстаріших і найвідоміших ковзних плиткових

головоломок. Вона відрізняється за механікою від блокових головоломок на, оскільки рух елементів залежить від наявності єдиного вільного місця.

Головоломка грається на квадратній сітці. Для класичних П'ятнашок це поле розміром 4×4 клітинки (всього 16 позицій). На полі розташовані квадратні плитки розміром 1×1 клітинка. У випадку 15-puzzle їх 15 штук. Кожна плитка, як правило, має номер (від 1 до 15). На полі завжди є одна вільна клітинка. Саме ця порожня позиція дозволяє здійснювати рухи іншими плитками.

N-puzzle - це узагальнення головоломки на квадратну сітку розміром n×n клітинок, де використовується n2−1 плитка з номерами від 1 до n2−1, і є одна порожня клітинка. Найпоширеніші варіанти – 8-puzzle (на полі 3x3) та 15-puzzle (на полі 4x4).

Рисунок 1.3 – Різні представлення гри 15-puzzle

Правила переміщення плиток дуже прості. Пересувати можна лише ту плитку, яка є суміжною (по горизонталі або вертикалі, але не по діагоналі) до порожньої клітинки. Обрана плитка пересувається (ковзає) на місце порожньої клітинки, роблячи її новою порожньою позицією. Плитки не можна обертати або піднімати з поля.

Основна механічна відмінність П'ятнашок полягає у принципі руху, який повністю залежить від наявності та розташування єдиного порожнього місця. У блокових головоломках рух блоків обмежується наявністю вільного шляху, що може складатися з кількох суміжних порожніх клітинок, і блоки

рухаються вздовж цього шляху, а не просто міняються місцями з порожнім полем. Крім того, у П'ятнашках усі рухомі елементи мають однаковий розмір (1x1), тоді як у "Година пік" та Клотскі використовуються блоки різних розмірів.

П'ятнашки є об'єктом вивчення в теорії алгоритмів та штучному інтелекті, зокрема як приклад задачі для демонстрації пошукових алгоритмів. Важливою властивістю є те, що не всі початкові конфігурації 15-puzzle є розв'язними. Точно половина всіх можливих початкових розташувань плиток є нерозв'язними, що визначається парністю кількості інверсій у послідовності плиток та парністю відстані порожньої клітинки від її кінцевої позиції.

Таким чином, П'ятнашки є класичною головоломкою, що використовує механіку переміщення плиток у єдине вільне місце, відрізняючись від блокових головоломок, де рух залежить від звільнення шляхів.

Існує безліч інших варіацій цього типу головоломок, часто з унікальними формами блоків, різними розмірами сіток або специфічними правилами руху, але базова ідея переміщення елементів для досягнення цільової конфігурації в межах обмеженого простору залишається спільною.

Таким чином, "Година пік" належить до добре дослідженої категорії ковзних блокових головоломок, типу Клотскі та сучасних цифрових імплементацій, як-от Unblock Me.

1.4. Специфікація вимог до програмної реалізації ігрового застосунку з регульованими рівнями складності

Розроблена програмна система має відповідати низці функціональних вимог (рис. 1.4), що забезпечують відтворення ігрового процесу, взаємодію з користувачем та надання додаткового аналітичного функціоналу. Специфікація вимог охоплює як основні функції, необхідні для базової

імітації гри, так і додаткові можливості, спрямовані на покращення користувацького досвіду та стимулювання ігрової активності.

Рисунок 1.4 – Основні та додаткові функціональні вимоги

1.4.1. Основні функціональні вимоги

Розглянемо основні функціональні вимоги:

1. Графічне представлення ігрового стану. Програмне забезпечення повинно забезпечувати візуалізацію ігрового середовища шляхом створення окремого віконного інтерфейсу. У цьому вікні має здійснюватися рендеринг ключових компонентів гри, включаючи:

- Ігрове поле розмірністю 6×6 клітинок, що представляє паркувальний майданчик.

- Сітку, що визначає допустимі позиції та траєкторії руху на полі.

- Моделі транспортних засобів (автомобілі та вантажівки), що відповідають їхнім розмірам (1×2 та 1×3 клітинки відповідно) та орієнтації (горизонтальна або вертикальна).

2. Інтуїтивно зрозумілий інтерфейс користувача та ергономіка взаємодії. Інтерфейс взаємодії з програмним забезпеченням повинен бути інтуїтивно зрозумілим для користувача, забезпечуючи легкість освоєння та використання. Ключові аспекти взаємодії включають:

- Реалізацію механізму переміщення моделей транспортних засобів по ігровому полю за допомогою маніпулятора типу "миша", зокрема, із застосуванням функціоналу "клацни та перетягни" (click-and-drag).

- Автоматичну валідацію запитів на переміщення: система має запобігати виконанню недійсних операцій, таких як переміщення транспортного засобу в уже зайняту клітинку, вихід за межі ігрового поля або спроба змінити орієнтацію об'єкта.

- Чіткий візуальний зворотний зв'язок при взаємодії (наприклад, підсвічування активних об'єктів).

3. Підтримка набору стандартизованих початкових конфігурацій. Програмна система повинна містити бібліотеку завдань, що представляють собою заздалегідь визначені початкові конфігурації розміщення транспортних засобів на ігровому полі. Ці завдання мають бути класифіковані за рівнями складності, охоплюючи діапазон від початкового до експертного або гросмейстерського рівнів. Набір конфігурацій може бути адаптований з офіційних комплектів фізичної версії гри.

1.4.2. Додаткові функціональні вимоги

1. Система оцінки ефективності вирішення. Після успішного завершення завдання (виведення цільового автомобіля за межі поля) програма повинна розраховувати та відображати кількісну оцінку ефективності дій гравця. Основною метрикою для оцінки має слугувати загальна кількість переміщень транспортних засобів, здійснених гравцем для досягнення цільового стану. Критерієм ефективності є мінімізація цієї кількості.

2. Система персистентного зберігання досягнень. Програмне забезпечення повинно забезпечувати механізм запису та зберігання найкращих результатів користувача для кожного завдання. Це включає фіксацію мінімальної кількості ходів та, за можливості, мінімального часу, необхідного для вирішення завдання. Збережені дані мають оновлюватися при досягненні користувачем нових рекордів. Система запису досягнень слугує інструментом для відстеження індивідуального прогресу гравця та посилення мотивації до оптимізації стратегій вирішення.

3. Система надання підказок. Для допомоги користувачеві у ситуаціях стагнації або при виникненні труднощів з пошуком подальших допустимих кроків, особливо у завданнях високого рівня складності, програмна система повинна надавати функціонал підказок. Ця система має бути здатною аналізувати поточний стан ігрового поля та генерувати інформацію щодо одного або декількох наступних допустимих переміщень, які належать до шляху розв'язку. Реалізація може базуватися на імплементації алгоритмів пошуку шляху в просторі станів гри для знаходження оптимального або субоптимального розв'язку від поточного стану.

Ці функціональні вимоги визначають необхідний набір можливостей програмної реалізації, спрямований на створення повноцінного та зручного віртуального аналога головоломки "Година пік" з розширеним аналітичним та допоміжним функціоналом.

Отже, було проведено ґрунтовний аналіз сучасного стану використання інформаційних технологій для створення ігрових застосунків, зокрема — логічних головоломок із регульованими рівнями складності. На прикладі конкретного ігрового застосунку окреслено основні засади побудови ігрової логіки, визначено принципи формування інтерфейсу користувача та обґрунтовано доцільність впровадження функціоналу, що дозволяє адаптувати складність задач до індивідуальних потреб користувача.

На основі проведеного аналізу сформульовано вимоги до функціоналу майбутнього програмного продукту, серед яких: підтримка багаторівневої складності, зручність навігації, ефективна візуалізація процесів гри, гнучкість у налаштуванні параметрів тощо. Розподіл вимог на основні та додаткові дозволяє більш структуровано підходити до планування розробки, забезпечуючи збалансованість між базовою функціональністю та потенційними можливостями розширення

РОЗДІЛ 2. АЛГОРИТМІЧНА РЕАЛІЗАЦІЯ ІГРОВОГО ЗАСТОСУНКУ ГОЛОВОЛОМКИ З РЕГУЛЬОВАНИМИ РІВНЯМИ СКЛАДНОСТІ

Цей розділ дипломної роботи присвячено опису архітектурного проєкту програмної системи, що імплементує головоломку в цифровій формі, а також деталізації методології та інструментарію, використаних у процесі розробки.

2.1. Архітектура програмного забезпечення

Архітектура програмного забезпечення побудована за принципом керування станами інтерфейсу користувача (User Interface State Management), де різні етапи взаємодії користувача з грою відповідають певним "видам" (Views). Такий підхід дозволяє чітко розділити логіку відображення та обробки подій для кожного окремого стану, спрощуючи розробку та підтримку коду.

2.1.1. Управління станами інтерфейсу (Views)

Програмна система передбачає щонайменше три основні стани інтерфейсу, між якими здійснюється перемикання в процесі використання додатка:

1. Стан головного меню (Main Menu View): Цей стан є початковим при запуску програми. Він відповідає за:

- Відображення назви гри та іншої привітальної інформації.
- Надання користувачеві можливості вибору конкретного завдання для початку гри зі списку доступних конфігурацій.
- Ініціалізацію переходу до стану ігрового процесу після вибору завдання.

2. Стан ігрового процесу (Game Play View / Task View): Цей стан активується після вибору завдання і представляє основне ігрове поле. Його функціонал включає:

- Візуалізацію ігрового поля (сітка 6×6), моделей транспортних засобів згідно з початковою конфігурацією завдання.

- Відображення ідентифікатора або псевдоніма поточного завдання.

- Відображення поточної кількості зроблених користувачем ходів.

- Відображення найкращого результату (мінімальної кількості ходів) користувача для даного завдання, отриманого раніше.

- Обробку подій взаємодії користувача, зокрема механізму "клацни та перетягни" для переміщення транспортних засобів згідно з правилами гри.

- Автоматичний облік та оновлення лічильника зроблених ходів при кожному допустимому переміщенні.

- Надання можливості ініціалізувати перезапуск поточного завдання (скидання до початкової конфігурації).

- Надання можливості ініціалізувати перехід назад до стану головного меню (вибору завдання).

- Надання функціоналу запиту та отримання підказки від системи.

3. Стан завершення завдання (Task Completion View / Result View): Цей стан активується автоматично при успішному вирішенні завдання (виведенні цільового автомобіля через вихід). Він відповідає за:

- Відображення повідомлення про успішне завершення завдання.

- Підрахунок та відображення фінального результату (наприклад, кількості зроблених ходів).

- Порівняння досягнутого результату з наявним рекордом для даного завдання.

- Збереження та оновлення запису найкращого результату, якщо поточний результат є кращим.

- Надання можливості ініціалізувати повернення до стану головного меню.

- Надання можливості ініціалізувати перезапуск тільки що завершеного завдання.

- У цьому стані функціонал надання підказок має бути деактивований.

2.1.2. Глобальні елементи управління

Окрім станів, прив'язаних до конкретного етапу гри, система може містити глобальні елементи управління, доступ до яких можливий незалежно від поточного активного виду.

Прикладом такого елемента є функціонал керування відтворенням фонової музики (увімкнення/вимкнення). Доступність цього функціоналу має зберігатися протягом всього ігрового процесу, за винятком, можливо, коротких періодів інтенсивної обробки даних (наприклад, під час пошуку рішення системою підказок), коли введення користувача може тимчасово ігноруватися.

2.2. Візуалізація структури та взаємодії

Для більш глибокого розуміння структури програмної системи та динаміки її роботи використовуються стандартні діаграми моделювання.

2.2.1. Проектування діаграми класів

Діаграма класів ілюструє статичну структуру програмного забезпечення, визначаючи основні класи, їхні атрибути, методи та зв'язки між ними (наприклад, асоціацію, агрегацію, наслідування), що відображає об'єктно-орієнтований дизайн системи.

Діаграма класів (рис. 2.1) моделює структуру програмного забезпечення, щодо гри типу головоломки з переміщенням об'єктів (Rush Hour).

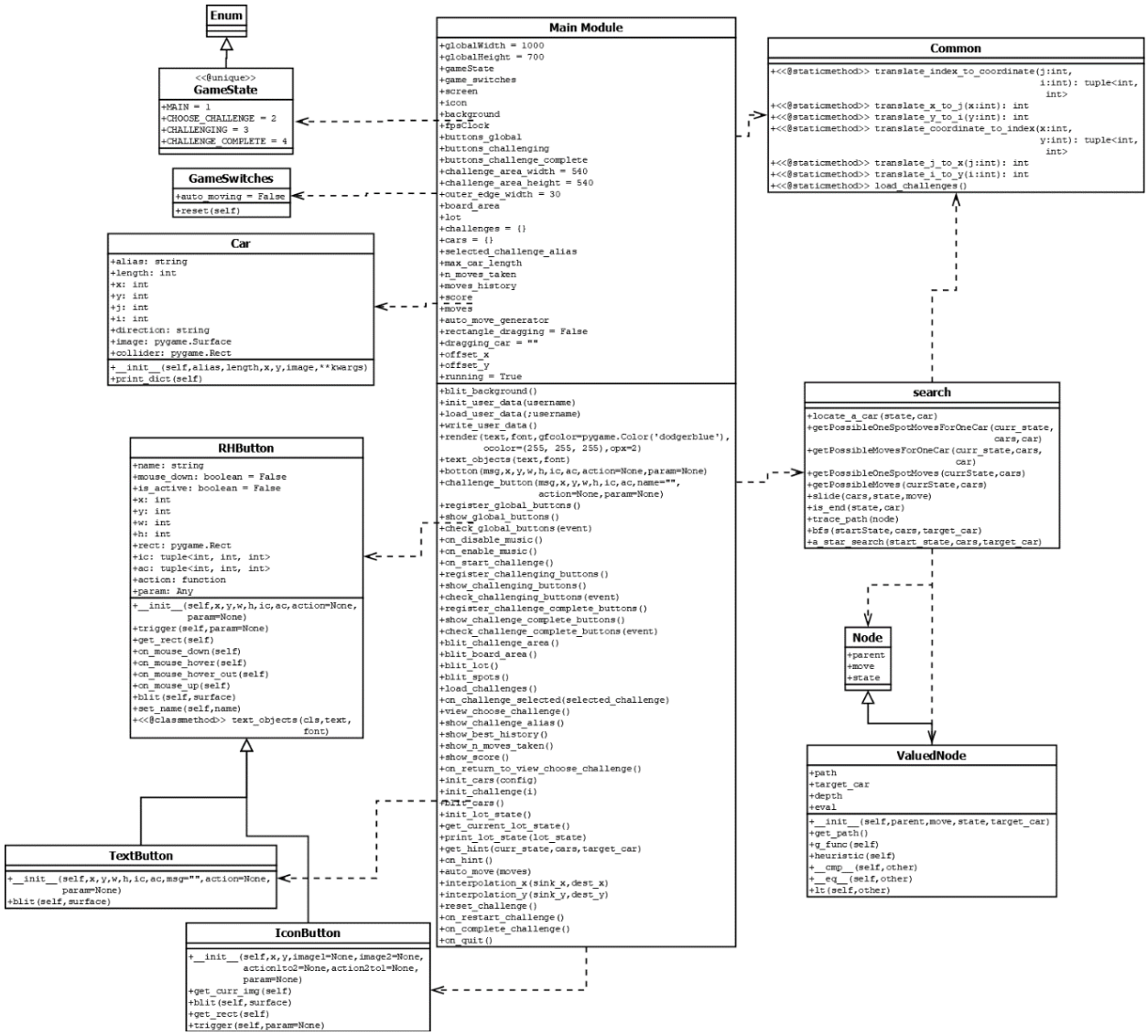

Рисунок 2.1 – Діаграма класів

Здійснимо опис класів та їхніх зв'язків, що показані на діаграмі.

1. GameState (Enum):

- Перерахування, що визначає можливі стани гри.

- Має члени: MAIN_MENU, CHOOSE_CHALLENGE, GAME_PROCESS, CHALLENGE_COMPLETE.

2. GameSwitches:

- Клас, що містить булеві атрибути (перемикачі), які, ймовірно, керують різними опціями чи станами гри.

- Має атрибути: hint_enabled, sound_enabled.

3. Car:

- Клас, що представляє автомобіль у грі.

- Атрибути: id, size, length, color, direction, start_pos, end_pos, orientation, rect, pivot_rect, image. Описують ідентифікатор, розмір, довжину, колір, початкову/кінцеву позицію, орієнтацію (горизонтальна/вертикальна), прямокутні області для позиціонування та зображення.

- Методи: init, move_by, can_move_to. Включають ініціалізацію, переміщення на певну відстань та перевірку можливості переміщення до нової позиції.

4. RIButton:

- Абстрактний (або базовий) клас для елементів користувацького інтерфейсу - кнопок.

- Атрибути: name, string, rect, color, is_active, is_hovered. Описують ім'я, текст/підпис, прямокутну область, колір, активність та стан наведення курсора.

- Методи: init, render, is_clicked, set_active, set_hovered. Включають ініціалізацію, візуалізацію, перевірку натискання, встановлення активності та стану наведення. Також, ймовірно, є метод trigger, який викликається при натисканні.

5. TextButton:

- Клас, що успадковується від RIButton. Представляє кнопку з текстом.

- Має додаткові атрибути та/або перевизначає методи для роботи з текстом.

- Методи: init, render.

6. IconButton:

- Клас, що успадковується від RIButton. Представляє кнопку з іконкою (зображенням).

- Має додаткові атрибути та/або перевизначає методи для роботи з зображеннями.

- Атрибути: image, image2 (для стану наведення).

- Методи: init, render.

7. Common:

- Клас, що містить статичні методи (позначені <<staticmethod>>) для загальних утилітних функцій.

- Методи: translate_coords_to_xy, translate_xy_to_coords, translate_coords_to_int, translate_int_to_coords, is_valid_challenge. Використовуються для перетворення координат та перевірки коректності даних.

8. Node:

- Клас, що представляє вузол у структурі даних, ймовірно, для пошуку (наприклад, у дереві пошуку стану).

- Атрибути: parent, state. Вказує на батьківський вузол та стан гри, пов'язаний з цим вузлом.

9. ValueNode:

- Клас, що успадковується від Node. Додає додаткову інформацію, пов'язану з "цінністю" вузла у пошуку.

- Атрибути: path, target_car, level. Шлях до вузла, цільовий автомобіль, рівень у дереві пошуку.

- Методи: init, eq (для порівняння), hash (для використання у хеш-таблицях).

10. Search:

- Клас, що містить статичні методи для реалізації алгоритмів пошуку.

- Методи: find_solution, is_end, get_possible_moves. Використовуються для пошуку розв'язку гри, перевірки досягнення кінцевого стану та отримання всіх можливих наступних ходів. Має залежності від класів Car, Node, ValueNode.

11. Main Module:

- Центральний клас, який керує грою.

- Атрибути: Містить численні атрибути, що представляють стан гри, ігрові об'єкти, елементи інтерфейсу, налаштування тощо (global_width, global_height, game_state, game_switches, cars, global_buttons, challenge_buttons, board_area, challenges, selected_challenge_alias, solver, running, user_data, render_tray, font, text_objects). Він агрегує або використовує об'єкти багатьох інших класів.

- Методи: Містить безліч методів, які реалізують основну логіку гри: ініціалізація (init), обробка подій (handle_input), оновлення стану гри (update), візуалізація (render), запуск/зупинка гри (run, stop), керування станами гри (change_game_state), роботу з кнопками (register_global_buttons, register_challenge_buttons, check_global_buttons_event, check_challenge_ buttons_event), логіку гри (is_win, get_solution), роботу з викликами/рівнями (load_challenge, show_challenge_view, show_solve_button), автовирішення (auto_move).

Тепер розглянемо зв'язки між класами.

Асоціація/Залежність: Main Module має асоціації або залежності від більшості інших класів (GameState, GameSwitches, Car, RIButton, Common, Search, Node, ValueNode), оскільки він створює, використовує або керує їхніми екземплярами чи викликає їхні методи. Пунктирні стрілки вказують на залежності (використання).

Успадкування: TextButton та IconButton успадковуються від RIButton. ValueNode успадковується від Node. Це показано суцільними стрілками з трикутним наконечником.

Залежність: Search залежить від Car, Node та ValueNode. Common використовується Main Module.

Загалом, діаграма описує структуровану систему для реалізації гри-головоломки з графічним інтерфейсом користувача, логікою гри, управлінням станами та, можливо, функціональністю для автоматичного пошуку рішень.

2.2.2. Розробка діаграми варіантів використання

Діаграма варіантів використання (Use Case Diagram) - діаграма описує функціональні вимоги до системи з точки зору різних ролей користувачів (акторів) та сценаріїв їх взаємодії з програмним забезпеченням.

На рисунку 2.2 зображено діаграму варіантів використання (Use Case Diagram) для ігрового застосунку "Година пік (Rush Hour)".

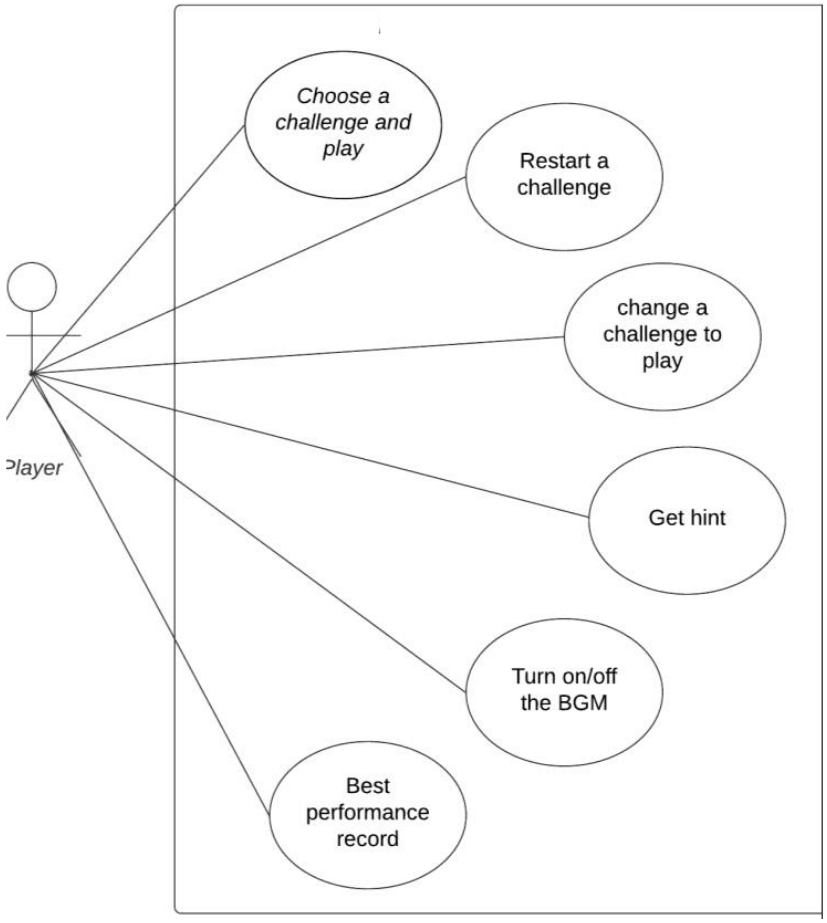

Рисунок 2.2 - Use Case Diagram

Ця діаграма показує взаємодію між користувачем (актором) та системою.

Актор. Player (Гравець) - представляє користувача, який взаємодіє із симулятором.

Варіанти використання (Use Cases):

- Choose a challenge and play (Вибрати завдання та грати): Гравець може вибрати певне завдання або рівень складності та розпочати гру.

- Restart a challenge (Перезапустити завдання): Гравець може скинути поточний стан завдання та почати його знову.

- change a challenge to play (Змінити завдання для гри): Гравець може перейти до іншого завдання під час гри або з меню вибору.

- Get hint (Отримати підказку): Гравець може запросити підказку від системи для допомоги у вирішенні завдання.

- Turn on/off the BGM (Увімкнути/вимкнути фонову музику): Гравець може керувати відтворенням фонової музики в симуляторі.

- Best performance record (Найкращий рекорд виконання): Гравець може переглядати або, можливо, зберігати свої найкращі результати проходження завдань.

Лінії, що з'єднують актора "Player" з кожним варіантом використання, позначають асоціацію. Це означає, що Гравець ініціює або бере участь у виконанні відповідного варіанта використання.

Діаграма варіантів використання ілюструє, які основні функції доступні користувачеві в цій системі та як користувач взаємодіє з нею. Вона дає загальне уявлення про функціональні вимоги до ігрового застосунку з точки зору користувача.

2.2.3. Проектування діаграми потоку даних

Діаграма потоку даних (Data Flow Diagram) - діаграма візуалізує рух інформації між основними компонентами системи та процеси, що

трансформують ці дані, надаючи уявлення про інформаційні потоки всередині додатка.

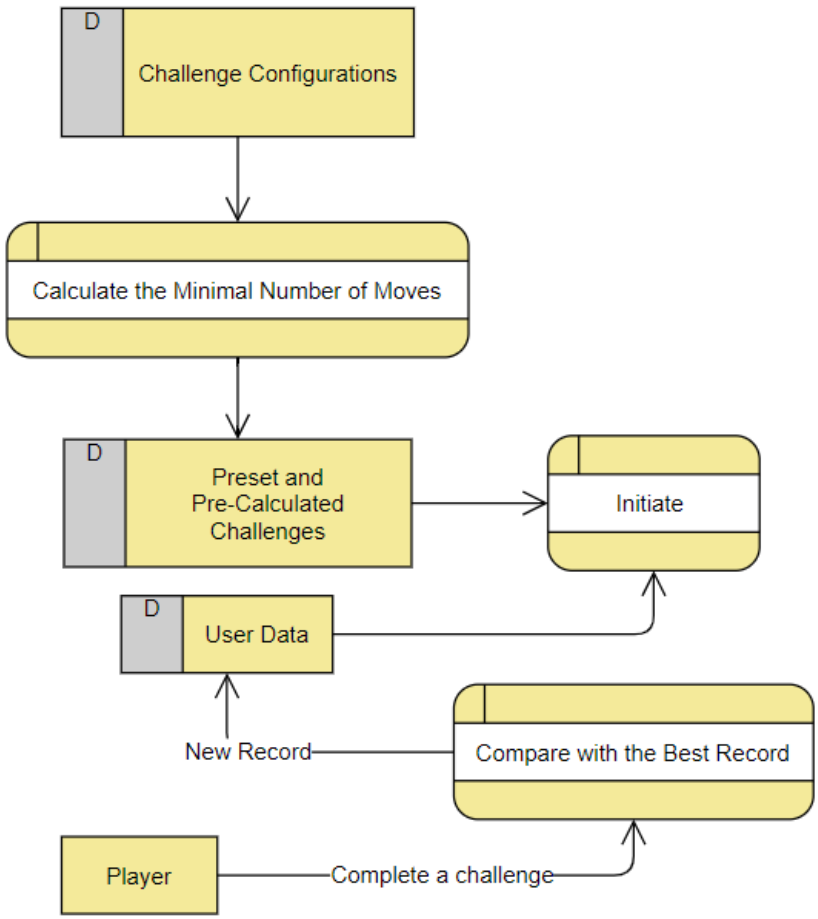

Рисунок 2.3 - Діаграма потоку даних (Data Flow Diagram)

На рисунку 2.3 зображено діаграму потоків даних (Data Flow Diagram - DFD), яка ілюструє, як дані переміщуються через систему, щостосується управління завданнями та записами гравців у ігровому застосунку.

Діаграма включає такі компоненти:

Процеси (Processes):

1. Calculate the Minimal Number of Moves (Обчислити мінімальну кількість ходів): Цей процес приймає конфігурації завдань і обчислює оптимальну (мінімальну) кількість ходів, необхідних для їх вирішення.

2. Initiate (Ініціювати): Цей процес запускає певне завдання гри. Він використовує дані із заздалегідь підготовлених завдань та, можливо, дані користувача.

3. Compare with the Best Record (Порівняти з найкращим рекордом): Цей процес приймає результат проходження завдання гравцем і порівнює його з поточним найкращим рекордом гравця.

Сховища даних (Data Stores) - позначені відкритими прямокутниками з літерою "D":

1. Challenge Configurations (Конфігурації завдань): Зберігає вихідні дані або налаштування для різних завдань.

2. Preset and Pre-Calculated Challenges (Заздалегідь підготовлені та попередньо обчислені завдання): Зберігає завдання, які вже були оброблені (можливо, з обчисленою мінімальною кількістю ходів).

3. User Data (Дані користувача): Зберігає інформацію про користувача, включаючи його найкращі результати (рекорди).

Зовнішні сутності (External Entities):

1. Player (Гравець): Зовнішній користувач, який взаємодіє із системою.

Тепер розглянемо потоки даних (Data Flows):

- Стрілка від Challenge Configurations до Calculate the Minimal Number of Moves: Потік даних конфігурацій завдань до процесу обчислення.

- Стрілка від Calculate the Minimal Number of Moves до Preset and Pre-Calculated Challenges: Потік обчислених даних (мінімальна кількість ходів) до сховища попередньо підготовлених завдань.

- Стрілка від Preset and Pre-Calculated Challenges до Initiate: Потік даних завдань для їх ініціації.

- Стрілка від User Data до Initiate: Потік даних користувача (можливо, поточний прогрес або налаштування) для ініціації завдання.

- Стрілка від Player до Compare with the Best Record з позначкою "Complete a challenge": Потік даних (сигнал або результат) про завершення завдання гравцем, що ініціює процес порівняння.

- Стрілка від Compare with the Best Record до User Data з позначкою "New Record": Потік даних про новий найкращий рекорд, який записується у сховище даних користувача.

- Стрілка від User Data до Compare with the Best Record: Потік даних про поточний найкращий рекорд користувача, який використовується для порівняння.

Діаграма ілюструє життєвий цикл даних, пов'язаних із завданнями та рекордами гравців: від початкових конфігурацій завдань, їхньої обробки та зберігання, до ініціації завдань для гравця, порівняння його результатів із записаними рекордами та оновлення даних користувача при досягненні нового рекорду.

2.3. Реалізація діаграми послідовності

Діаграма послідовності (Sequence Diagram) це діаграма, яка демонструє динамічну взаємодію між об'єктами або компонентами системи у часовій послідовності для виконання конкретного сценарію використання, такого як типовий ігровий сеанс від вибору завдання до його завершення.

На рисунку 2.4 зображено діаграму послідовності (Sequence Diagram), яка ілюструє взаємодію між користувачем (гравцем) та різними компонентами системи (Main Module, Stored Data, Search) у часі.

Об'єкти/Лінії життя (Lifelines):

1. Player (Гравець): Представляє зовнішнього користувача.

2. Main Module (Головний Модуль): Представляє центральну логіку програми, яка координує дії.

3. Stored Data (Збережені Дані): Представляє компонент для зберігання та отримання даних (наприклад, файли, база даних).

4. Search (Пошук): Представляє компонент, відповідальний за пошук, ймовірно, рішень або підказок для головоломки.

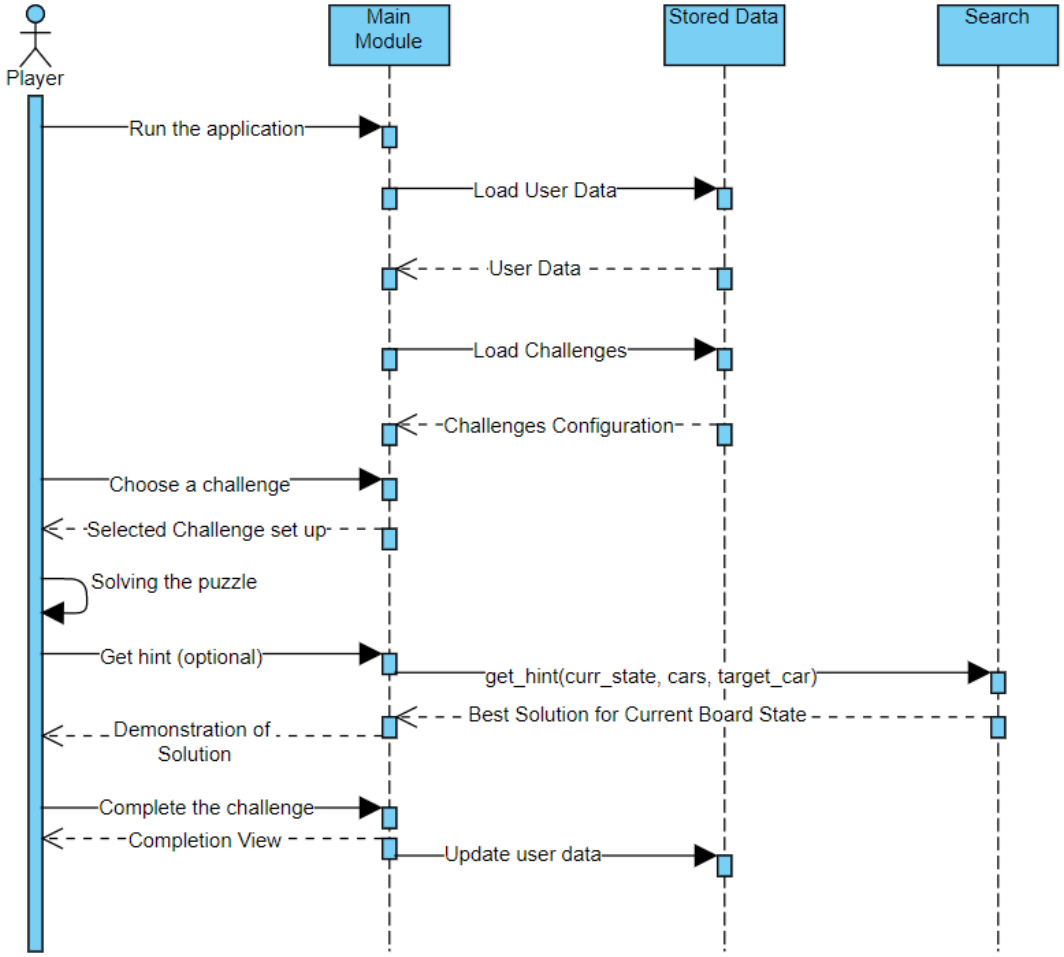

Рисунок 2.4 - Діаграма послідовності (Sequence Diagram)

Діаграма показує такі взаємодії в хронологічному порядку (зверху вниз):

1. Run the application (Запустити додаток): Гравець ініціює запуск програми, надсилаючи повідомлення до Main Module.

2. Load User Data (Завантажити дані користувача): Main Module надсилає запит до Stored Data для завантаження даних користувача.

3. <-- User Data (Дані користувача): Stored Data надсилає у відповідь дані користувача до Main Module.

4. Load Challenges (Завантажити завдання): Main Module надсилає запит до Stored Data для завантаження конфігурацій завдань.

5. <-- Challenges Configuration (Конфігурація завдань): Stored Data надсилає у відповідь конфігурації завдань до Main Module.

6. Choose a challenge (Вибрати завдання): Гравець вибирає певне завдання через Main Module.

7. <-- Selected Challenge set up (Вибране завдання налаштовано): Main Module відповідає гравцеві, сигналізуючи про завершення налаштування вибраного завдання.

8. Solving the puzzle (Вирішення головоломки): На цій фазі Гравець взаємодіє з Main Module, переміщуючи елементи головоломки. Це показано як активність на лініях життя обох учасників.

9. Get hint (optional) (Отримати підказку (опціонально)): Гравець (за бажанням) надсилає запит на отримання підказки до Main Module.

10. get_hint(curr_state, cars, target_car) (отримати_підказку (поточний_стан, автомобілі, цільовий_автомобіль)): Main Module надсилає запит до компонента Search, передаючи поточний стан гри та інформацію про цільовий елемент, щоб отримати підказку.

11. <-- Best Solution for Current Board State (Найкраще рішення для поточного стану дошки): Компонент Search надсилає у відповідь найкраще знайдене рішення (або його частину) до Main Module.

12. <-- Demonstration of Solution (Демонстрація рішення): Main Module надає гравцеві демонстрацію або інформацію про отриману підказку/рішення.

13. Complete the challenge (Завершити завдання): Гравець завершує завдання (успішно чи іншим чином), надсилаючи повідомлення до Main Module.

14. <-- Completion View (Вигляд завершення): Main Module відображає екран завершення або результати гравцеві.

15. Update user data (Оновити дані користувача): Main Module надсилає дані для оновлення (наприклад, прогрес, рахунок, найкращий рекорд) до Stored Data.

Ця діаграма послідовності описує сценарій використання програми: від її запуску та завантаження початкових даних, через процес вибору та вирішення завдання гравцем (з можливістю отримати підказку), до завершення завдання та збереження оновлених даних користувача. Вона чітко показує порядок викликів та обміну інформацією між різними частинами системи та користувачем.

Ці діаграми є інструментами для документації та аналізу архітектури, надаючи візуальне представлення складних взаємозв'язків та потоків виконання.

2.4. Вибір та обґрунтування методологій реалізації

Розробка програмного забезпечення здійснювалася з використанням наступних технологій та інструментів.

1. Мова програмування: Основною мовою програмування для реалізації проєкту є Python. Використовувалася версія 3.9.5.

2. Середовище розробки (IDE): Як інтегроване середовище розробки (Integrated Development Environment) застосовувався PyCharm.

PyCharm — це інтегроване середовище розробки (IDE) для мови програмування Python, створене компанією JetBrains. Воно забезпечує повний спектр інструментів для ефективної розробки, налагодження, тестування й обслуговування програмного коду на Python. PyCharm є одним із найпопулярніших середовищ серед Python-розробників завдяки своїй гнучкості, підтримці сучасних технологій та широкому функціоналу.

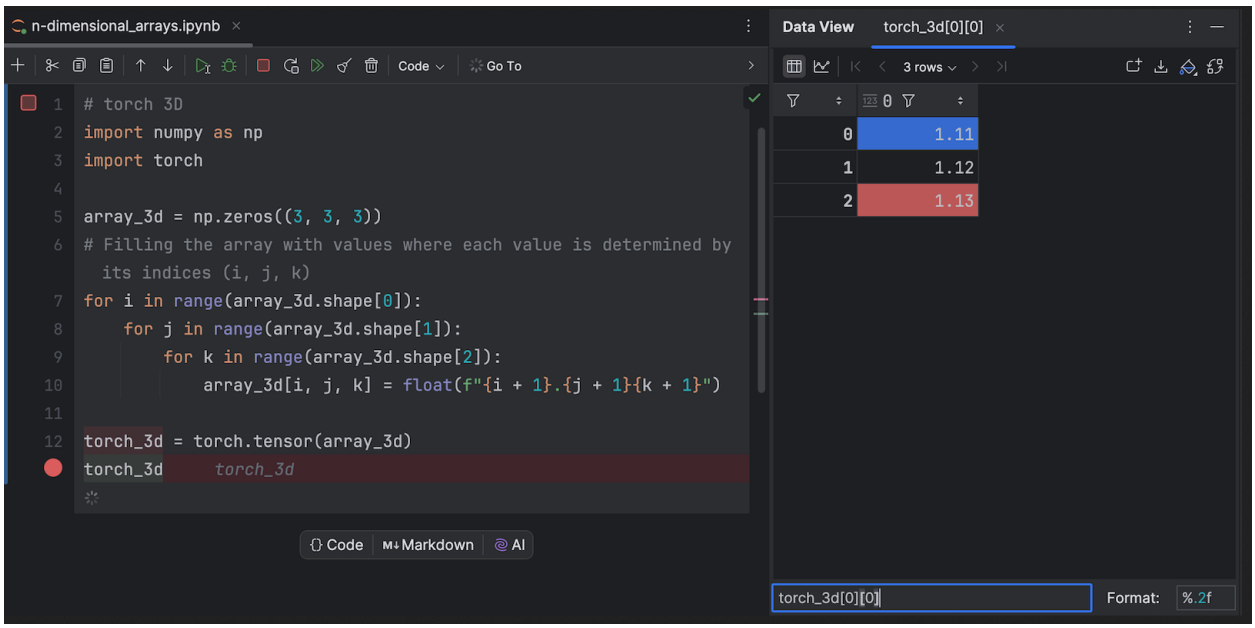

Рисунок 2.5 – Вигляд IDE PyCharm

Наведемо ключові характеристики PyCharm:

- PyCharm забезпечує інтелектуальне автодоповнення, рефакторинг, перевірку помилок у реальному часі та навігацію по проєкту.

- IDE має потужний вбудований відлагоджувач, підтримує юніт-тестування, а також інтегрується з такими фреймворками, як pytest, unittest, doctest.

- PyCharm дозволяє підключатися до різних СУБД, переглядати структуру таблиць, виконувати SQL-запити прямо з IDE.

- Вбудована підтримка Git, GitHub, Mercurial, Subversion тощо.

- Професійна версія підтримує HTML, CSS, JavaScript, фреймворки Django, Flask, Jinja2 та інші технології.

- Зручне створення та управління Python-інтерпретаторами (віртуальні середовища, Conda, Docker).

- PyCharm доступний для Windows, macOS та Linux.

3. Зовнішні залежності: Проєкт має залежності від наступних зовнішніх програмних пакетів, які необхідно встановити окрім стандартної бібліотеки Python:

4. pip: Система керування пакетами Python (стандартно постачається з сучасними версіями Python, але може потребувати оновлення).

pip (Python Installer Package) — це офіційна система керування пакетами для мови програмування Python, яка використовується для встановлення, оновлення, видалення та управління сторонніми бібліотеками з репозиторію Python Package Index (PyPI).

Особливості pip:

- Повністю інтегрований із системою віртуальних середовищ (venv, virtualenv, conda).

- Працює кросплатформено — на Windows, macOS, Linux.

- Підтримує локальні архіви та нестандартні джерела пакетів (через --index-url, --find-links).

- Має потужний механізм вирішення залежностей, що автоматично встановлює всі необхідні додаткові пакети.

pip — це незамінний інструмент для будь-якого Python-розробника, що забезпечує просте та ефективне управління зовнішніми бібліотеками. У процесі створення ігрових застосунків, зокрема з використанням pygame, pygame-functions, numpy тощо, pip відіграє ключову роль у забезпеченні гнучкого, масштабованого й стабільного середовища розробки.

5. setuptools: Набір інструментів для пакування Python проєктів.

setuptools — це стандартна та широко використовувана бібліотека Python, призначена для пакування, встановлення та поширення програмного забезпечення, створеного на мові Python. Вона дозволяє розробникам організовувати проєкти у вигляді модулів і пакетів, створювати архіви для публікації, а також забезпечує автоматичну установку залежностей.

Призначення та можливості setuptools:

- Дозволяє організувати програму або бібліотеку у вигляді пакета, який можна легко встановити через pip.

- setuptools може автоматично завантажувати та встановлювати інші потрібні бібліотеки під час інсталяції пакета.

- Створення виконуваних скриптів (entry points), наприклад, консольні команди, які викликаються після встановлення (через console_scripts).

- Інтеграція з pip, PyPI і віртуальними середовищами, тобто пакети, зібрані за допомогою setuptools, легко публікувати на Python Package Index (PyPI) та встановлювати через pip install.

6. pygame: Крос-платформна бібліотека для розробки комп'ютерних ігор та мультимедійних застосунків. Ця бібліотека надає функціонал для створення графічного вікна, рендерингу зображень, обробки подій введення та роботи зі звуком.

7. pygame-functions: Додаткова бібліотека, що може спрощувати розробку з використанням Pygame, надаючи високоуровневі функції для типових операцій. Використовувалася версія 0.0.1.

pygame-functions — це додаткова бібліотека, побудована на основі pygame, яка надає спрощений інтерфейс для створення ігрових застосунків. Вона орієнтована насамперед на новачків у програмуванні, викладачів, школярів і студентів, які хочуть швидко створити гру без занурення в усі технічні особливості класичного pygame.

Наведемо переваги і особливості pygame-functions:

- Замість десятків рядків коду pygame, бібліотека дозволяє використовувати прості функції на кшталт makeSprite(), moveSprite(), showSprite(), що робить розробку інтуїтивно зрозумілою.

- Багато функціональностей pygame обгорнуто у зручні функції, які приховують складність обробки подій, оновлення екрана, анімацій тощо.

- Дозволяє швидко створити основне вікно гри, розмістити спрайти, обробити клавіатуру або мишу без необхідності налаштовувати низькорівневі структури pygame.

- Бібліотека широко використовується в навчальних курсах з програмування, у шкільних гуртках і під час перших занять з розробки ігор.

Отже, pygame-functions — це зручна допоміжна бібліотека для розробки простих 2D-ігор на Python, що надає легкий старт для новачків. Вона дозволяє зосередитися на ідеї гри, а не на технічних деталях реалізації, і може використовуватися як ефективний навчальний інструмент.

Ці інструменти та бібліотеки забезпечили необхідну технічну базу для реалізації графічного інтерфейсу, ігрової логіки та інтеграції додаткових функціональних можливостей, визначених у специфікації вимог.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ ІГРОВОГО ЗАСТОСУНКУ ГОЛОВОЛОМКИ З РЕГУЛЬОВАНИМИ РІВНЯМИ СКЛАДНОСТІ

3.1. Реалізація підсистеми зберігання даних

Підсистема зберігання даних охоплює інформацію щодо попередньо визначених та обчислених завдань, а також дані користувачів.

Завдання сконфігуровані відповідно до фізичних карт завдань, що входять до комплекту настільної гри. Конфігураційні дані зберігаються у текстовому файлі стандартизованого формату, який використовується основним модулем для завантаження параметрів завдань до оперативної пам'яті під час ініціалізації програми. Файл містить структуровану інформацію для кожного завдання, включаючи його унікальний ідентифікатор (псевдонім), кількість транспортних засобів (автомобілів), ідентифікатор кожного автомобіля, його початкову позицію та орієнтацію на ігровій дошці.

Оптимальна (мінімальна) кількість рухів, необхідна для вирішення кожного завдання, обчислюється із застосуванням алгоритму пошуку в ширину (BFS). Отримані значення зберігаються в окремому текстовому файлі. Ця інформація використовується для фінальної оцінки продуктивності гравця після успішного завершення завдання. Дані користувачів, що включають показники найкращої продуктивності (рекорди) для кожного завдання для кожного гравця, зберігаються у форматі JSON файлу.

3.2. Програмна реалізація графічного інтерфейсу користувача

Архітектура та функціональність графічних інтерфейсів користувача представлені у цьому розділі з відповідними описами візуальних компонентів та інтерактивних елементів.

Рисунок 3.1 – Головна сторінка застосунку

Після ініціалізації програми користувачу відображається "Основний вигляд" (Main View). Цей інтерфейс призначений для вибору завдання. Представлений перелік попередньо сконфігурованих завдань супроводжується індикаторами рівня складності, реалізованими у вигляді іконок зірочок. Область "Вибране завдання" (Selected Challenge) у нижній центральній частині інтерфейсу динамічно оновлюється відповідно до вибору користувача, відображаючи поточне обране завдання. Після здійснення вибору завдання, кнопка "Го!" (Go!) стає видимою та активується, дозволяючи перейти до ігрового процесу.

Після активації кнопки "Го!" завантажується початковий стан обраного завдання та відображається "Вигляд завдання" (Challenge View). Взаємодія гравця полягає у виборі (клацанням лівою кнопкою миші) та переміщенні (перетягуванням) графічного представлення автомобілів на ігровій дошці. Механізм переміщення обмежує рух транспортних засобів, запобігаючи їх перетинанню або накладанню. Після відпускання лівої кнопки миші, вибраний автомобіль автоматично позиціонується у найближчому

допустимому місці (як ілюстровано на Рисунках 7 та 8). Якщо в результаті цієї дії положення автомобіля змінилося, лічильник виконаних ходів збільшується на одиницю.

Рисунок 3.2 - Вигляд завдання (Challenge View)

Рисунок 3.3 – Приклад переміщення рожевого автомобіля

Рисунок 3.4 – Приклад як при відпусканні лівої кнопки миші рожева машина автоматично переміщується до найближчого місця

Рисунок 3.5 – Елементи керування

Під час перебування у "Вигляді завдання" доступні такі елементи керування (рис. 3.5):

- Кнопка "Повернутися" (Back) здійснює перехід до "Вигляду вибору завдання".

- Кнопка "Перезапустити" (Restart) скидає поточний стан завдання до початкового, включаючи положення автомобілів та лічильник ходів.

- Кнопка "Вийти" (Exit) завершує виконання програми.

- Кнопка "Підказка" (Hint) ініціює процес пошуку оптимального рішення на основі поточного стану ігрової дошки із застосуванням відповідного алгоритму пошуку. Під час виконання пошуку ігровий процес тимчасово призупиняється.

Під час виконання алгоритму пошуку рішення користувацькі взаємодії блокуються. Після успішного знаходження рішення, відповідне індикаційне повідомлення червоним шрифтом (показане на рисунку 3.6) зникає, і автоматично запускається демонстрація послідовності ходів, що ведуть до вирішення, починаючи з поточного стану. Демонстрація рішення передбачає автоматичне переміщення автомобілів відповідно до знайденої послідовності дій. Під час автоматичного відтворення натискання на кнопку "Підказка" дозволяє зупинити демонстрацію після завершення поточного одиничного кроку переміщення. Якщо демонстрація не переривається гравцем, вона відтворює всі кроки рішення, за винятком фінального переміщення цільового автомобіля за межі ігрового поля. Виконання цього останнього кроку залишається за гравцем.

Рисунок 3.6 - Вигляд завдання при очікуванні пошуку рішення

Рисунок 3.7 - Демонстрація рішення залишить останній хід гравцеві

Рисунок 3.8 - Вигляд виконаного завдання

Завдання вважається успішно завершеним після переміщення цільового автомобіля за межі ігрової дошки. Після завершення завдання здійснюється

розрахунок оцінки продуктивності гравця. Отримана оцінка оновлюється у даних користувача та зберігається.

Діапазон можливих оцінок продуктивності знаходиться в інтервалі [0,100]. Чим меншу кількість рухів виконує гравець для завершення завдання, тим вищою буде отримана оцінка. Методологія розрахунку оцінки базується на співвідношенні кількості надлишкових ходів до мінімально необхідної кількості ходів для даного завдання. Формула для розрахунку оцінки має наступний вигляд:

$$\text{Оцінка} = \max\left(\left\lfloor 100 \times \left(1 - \frac{n-m}{m}\right)\right\rfloor, 0\right)$$

де:

n — кількість рухів, виконаних гравцем для завершення завдання;

m — мінімальна кількість рухів, необхідна для завершення завдання.

Наприклад, якщо для успішного вирішення Завдання 333 гравець здійснив n=34 рухи, при мінімально необхідній кількості m=34, розрахунок оцінки виконується наступним чином:

$$\text{Оцінка} = \max\left(\left\lfloor 100 \times \left(1 - \frac{34-34}{34}\right)\right\rfloor, 0\right) = \max(\lfloor 100 \times (1-0)\rfloor, 0) = 100$$

Якщо кількість виконаних гравцем рухів становить n=40, при мінімально необхідній кількості m=34, то оцінка буде:

$$\text{Оцінка} = \max\left(\left\lfloor 100 \times \left(1 - \frac{40-34}{34}\right)\right\rfloor, 0\right) = 82$$

як показано на рисунку 3.9.

Рисунок 3.9 - Якщо буде зроблено 40 ходів, то рахунок становитиме 82

Якщо кількість виконаних рухів n більша або дорівнює подвоєній мінімальній кількості рухів m для даного завдання (n≥2m), то оцінка приймає значення 0.

Рисунок 3.10 – Випадок, коли надто багато ходів призводить до нульового рахунку

Також є кнопка ввімкнення музики в нижньому лівому куті вікна. Вона завжди відображається, незалежно від стану гри. Користувач може увімкнути або вимкнути фонову музику гри, клацнувши по цій кнопці.

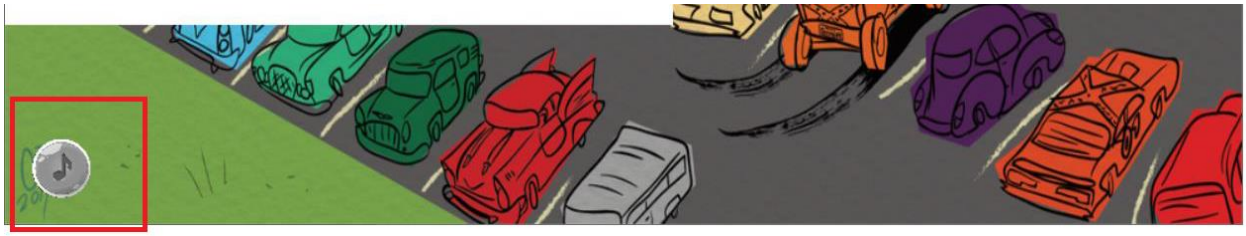

Рисунок 3.11 – Відображення кнопки ввімкнення фонової музики

В лістингу 3.1 представлено код Python, що визначає клас Car (Автомобіль), призначений для використання в грі, яка розробляється за допомогою бібліотеки Pygame.

Лістинг 3.1. Клас car.py

```
import pygame
import os

class Car:
    x = 2
    y = 4
    direction = "H"  # horizontal
    image = pygame.image.load((os.path.split(os.path.realpath(__file__))[0] + "/img/car1.png"))
    collider = image.get_rect()

    def __init__(self, alias, length, x, y, direction, image, **kwargs):
        for k, v in kwargs.items():
            self.__dict__[k] = v
        self.alias = alias
        self.length = length
        self.x = x
        self.y = y
        self.j = int(x / 80 + 1) - 1
        self.i = int(y / 80 + 1) - 1
        self.direction = direction
        self.image = image
        if self.length == 2:
            self.image = pygame.transform.scale(self.image, (80, 160))
        elif self.length == 3:
            self.image = pygame.transform.scale(self.image, (80, 240))
        if direction == "H":
            self.image = pygame.transform.rotate(self.image, -90)
        self.collider = self.image.get_rect()
        self.collider.topleft = (x, y)
```

Загалом, цей клас є основним блоком для гри, бо потрібно керувати автомобілями різної довжини, що рухаються по сітці в горизонтальному і вертикальному напрямку.

Наступний код (лістинг 3.2) надає структуру для керування станом гри (за допомогою GameState) та контролю певних динамічних аспектів гри (за допомогою GameSwitches), зокрема пов'язаних з автоматичними рухами.

Лістинг 3.2. gameState.py

```
from enum import Enum, unique

@unique
class GameState(Enum):
    MAIN = 1
    CHOOSE_CHALLENGE = 2
    CHALLENGING = 3
    CHALLENGE_COMPLETE = 4

class GameSwitches:
    auto_moving = False
    auto_move_stopping = False

    def reset(self):
        self.auto_moving = False
        self.auto_move_stopping = False
```

Цей код складається з двох основних частин: визначення переліку (enum) для станів гри та визначення простого класу для зберігання булевих прапорців (перемикачів) гри.

Наступний код (лістинг 3.3) є основою для реалізації алгоритму пошуку рішення для гри. Класи Node та ValuedNode використовуються для побудови дерева пошуку станів гри. Функції init_... відповідають за налаштування гри та завантаження завдань. Функції get_..._lot_state та locate_a_car працюють з представленням дошки. Функції getPossible...MovesForOneCar генерують можливі наступні стани гри з поточного. Ймовірно, ці функції будуть використовуватися в якомусь алгоритмі пошуку (A* або BFS), який буде перебирати можливі стани,

використовуючи heapq для ефективного управління чергою станів для перевірки, доки не знайде стан, де цільовий автомобіль може виїхати.

Лістинг 3.3. search.py

```
import operator
import random
import pygame
import pprint
import copy
import time
import sys
import heapq
sys.path.append(".")
from car import Car
from common import Common

class Node:
    def __init__(self, parent, move, state):
        self.parent = parent
        self.move = move
        self.state = state

class ValuedNode(Node):
    path = list()
    def __init__(self, parent, move, state, target_car):
        super().__init__(parent, move, state)
        self.target_car = target_car
        self.get_path()
        self.depth = self.parent.depth + 1 if self.parent is not None else 0
        g_score = self.g_func()
        h_score = self.heuristic()
        self.eval = g_score + h_score
    def get_path(self):
        if self.parent is not None:
            self.path = self.parent.path + [self.state]
        else:
            self.path = [self.state]
    def g_func(self):
        return self.depth
    def heuristic(self):
        eval = 0
        target_found = False
        for i in range(0, 6):
            curr_spot = self.state[2][i]
            if curr_spot == self.target_car:
                target_found = True
            if not target_found:
                continue
            else:
                if curr_spot != self.target_car and curr_spot != ".":
                    eval += 1
        return eval
    def __lt__(self, other):
        return self.eval < other.eval
    def __le__(self, other):
        return self.eval <= other.eval
```

```
challenges = {}
cars = {}
outer_edge_width = 30
max_car_length = 3
target_car = None

def load_challenges():
    global challenges
    challenges = Common.load_challenges()

def init_cars(config):
    global cars, target_car
    cars = {}
    for k, v in config.items():
        carImg = pygame.image.load(f"img/car_{k}.png")
        car = Car(k, int(v["length"]), Common.translate_index_to_coordinate(v["j"], v["i"])[0],
                  Common.translate_index_to_coordinate(v["j"], v["i"])[1], v["direction"],
                  carImg, board_offset=outer_edge_width)
        cars[k] = car
    target_car = cars[next(iter(cars))]

def init_challenge(i):
    challenge = challenges[str(i)]
    init_cars(challenge["config"])

def init_lot_state() -> list:
    lot_state = [["." for i in range(6)] for j in range(6)]
    for _ in range(max_car_length):
        lot_state[2].append(".")
    return lot_state

def get_current_lot_state() -> list:
    current_lot_state = init_lot_state()
    for car in cars.values():
        j = car.j
        i = car.i
        if car.direction == "H":
            for k in range(car.length):
                current_lot_state[i][j + k] = car.alias
        elif car.direction == "V":
            for k in range(car.length):
                current_lot_state[i + k][j] = car.alias
    return current_lot_state

def locate_a_car(state, car):
    i, j = -1, -1
    for ii in range(6):
        if ii != 2:
            for jj in range(6):
                if state[ii][jj] == car:
                    i, j = ii, jj
                    break
        else:
            for jj in range(9):
                if state[ii][jj] == car:
                    i, j = ii, jj
                    break
    if i == -1 or j == -1:
        raise ValueError(f"car {car} not found on the board")
    return i, j
```

```
def getPossibleOneSpotMovesForOneCar(curr_state, cars, car):
    possibleMoves = list()
    i, j = locate_a_car(curr_state, car)
    if cars[car].direction == "H":
        if (i != 2 and j + (cars[car].length - 1) + 1 <= 5) or (i == 2 and j + (cars[car].lengt
            if curr_state[i][j + (cars[car].length - 1) + 1] == ".":
                possibleMoves.append((car, "H", 1))
        if j - 1 >= 0:
            if curr_state[i][j - 1] == ".":
                possibleMoves.append((car, "H", -1))
    else:
        if i + (cars[car].length - 1) + 1 <= 5:
            if curr_state[i + (cars[car].length - 1) + 1][j] == ".":
                possibleMoves.append((car, "V", 1))
        if i - 1 >= 0:
            if curr_state[i - 1][j] == ".":
                possibleMoves.append((car, "V", -1))
    return possibleMoves

def getPossibleMovesForOneCar(curr_state, cars, car):
    possibleMoves = list()
    i, j = locate_a_car(curr_state, car)
    if cars[car].direction == "H":
        leftSpace = j
        rightSpace = 6 - j - cars[car].length if i != 2 else 7 - j - cars[car].length
        if leftSpace > 0:
            for s in range(1, leftSpace + 1):
                if curr_state[i][j - s] == ".":
                    possibleMoves.append((car, "H", -s))
        if rightSpace > 0:
            for s in range(1, rightSpace + 1):
                if curr_state[i][j + s] == ".":
                    possibleMoves.append((car, "H", s))
    else:
        upSpace = i
        downSpace = 6 - i - cars[car].length
        if upSpace > 0:
            for s in range(1, upSpace + 1):
                if curr_state[i - s][j] == ".":
                    possibleMoves.append((car, "V", -s))
        if downSpace > 0:
            for s in range(1, downSpace + 1):
                if curr_state[i + s][j] == ".":
                    possibleMoves.append((car, "V", s))
    return possibleMoves
```

Цей код на Python, є частиною програми для розв'язання головоломки “Година пік”, де потрібно переміщувати автомобілі по сітці, щоб вивести один з них.

Ось детальний опис кожної частини (лістинг 3.3):

1. **Імпорти:**

- operator: Модуль для ефективних функцій операторів. Використовується для порівнянь.
- random: Модуль для генерації випадкових чисел. Є корисним в інших частинах гри (наприклад, для вибору випадкового завдання).
- pygame: Бібліотека для розробки ігор. Використовується для роботи із зображеннями (pygame.image).
- pprint: Модуль для "красивого" друку структур даних (наприклад, списків списків, що представляють стан дошки).
- copy: Модуль для створення копій об'єктів. Важливо при роботі зі станами гри, щоб не змінювати оригінальний стан при генерації нових.
- time: Модуль для роботи з часом. Може використовуватися для вимірювання часу виконання алгоритмів.
- sys: Модуль для доступу до системних параметрів та функцій. Використовується для додавання шляху до пошуку модулів (sys.path.append(".")).
- heapq: Модуль, що реалізує алгоритм черги з пріоритетом (min-heap). Використовується для реалізації алгоритму пошуку шляху, наприклад, A*.
- from car import Car: Імпортує клас Car з іншого файлу car.py. Цей клас, як ми бачили раніше, представляє окремий автомобіль.
- from common import Common: Імпортує клас Common з файлу common.py. Ймовірно, містить допоміжні функції, такі як завантаження завдань або перетворення координат.

2. **Клас Node:**
   - Базовий клас для представлення вузла в дереві пошуку станів.
   - parent: Посилання на батьківський вузол. Дозволяє відновити шлях від початкового стану до поточного.

- move: Дія (хід), яка призвела до цього стану з батьківського вузла. Ймовірно, кортеж типу (alias_автомобіля, напрямок, кількість_кроків).
- state: Представлення стану ігрової дошки після виконання move. Ймовірно, список списків, що представляє сітку.

3. **Клас ValuedNode (успадковує від Node):**
   - Розширює клас Node, додаючи функціональність для оцінки вузлів, що необхідно для алгоритмів пошуку з пріоритетом (наприклад, A*).
   - path: Список станів, що ведуть від початкового вузла до поточного.
   - target_car: Об'єкт або ідентифікатор цільового автомобіля (того, який потрібно вивести з дошки).
   - __init__(...): Конструктор. Викликає конструктор батьківського класу (super().__init__(...)). Розраховує глибину вузла (depth), оцінює його за допомогою функцій g_func та heuristic і зберігає загальну оцінку в self.eval.
   - get_path(): Рекурсивно будує шлях від кореневого вузла до поточного, зберігаючи послідовність станів.
   - g_func(): Функція вартості шляху від початкового вузла до поточного. У даному випадку просто повертає глибину вузла (self.depth), що відповідає кількості ходів.
   - heuristic(): Евристична функція, яка оцінює "близькість" поточного стану до цільового стану (коли цільовий автомобіль може виїхати). Вона рахує кількість автомобілів, які блокують шлях цільовому автомобілю в його ряду (ряд 2).
   - Методи порівняння (__lt__, __le__, __gt__, __ge__): Дозволяють порівнювати об'єкти ValuedNode на основі їхньої оцінки (self.eval). Це необхідно для роботи черги з пріоритетом у heapq.

4. **Глобальні змінні:**

- challenges: Словник для зберігання конфігурацій різних завдань/рівнів.
- cars: Словник для зберігання об'єктів Car, проіндексованих за їхніми псевдонімами.
- outer_edge_width: Ширина зовнішнього краю дошки в пікселях.
- max_car_length: Максимальна довжина автомобіля.
- target_car: Посилання на об'єкт цільового автомобіля.

5. **Функції ініціалізації:**
   - load_challenges(): Завантажує конфігурації завдань з файлу за допомогою Common.load_challenges().
   - init_cars(config): Ініціалізує словник cars на основі переданої конфігурації завдання. Створює об'єкти Car, завантажує відповідні зображення та встановлює їхні початкові позиції та напрямки.
   - init_challenge(i): Завантажує конфігурацію конкретного завдання за його індексом i і викликає init_cars для ініціалізації автомобілів для цього завдання.
6. **Функції стану дошки:**
   - init_lot_state() -> list: Створює початкове представлення порожньої ігрової дошки (парковки) як списку списків. Розмір дошки 6x6, але ряд 2 (індекс 2) має додаткові клітинки для виїзду.
   - get_current_lot_state() -> list: Перетворює поточні позиції всіх об'єктів Car на дошці в представлення стану дошки у вигляді списку списків, де кожна клітинка містить псевдонім автомобіля або "." (порожньо).
7. **Функції для визначення можливих ходів:**
   - locate_a_car(state, car): Знаходить координати (рядок i, стовпець j) верхнього лівого кута заданого автомобіля (car) на дошці (state).
   - getPossibleOneSpotMovesForOneCar(curr_state, cars, car): Визначає всі можливі ходи на *одну* клітинку для заданого автомобіля (car) у

поточному стані дошки (curr_state). Перевіряє, чи є сусідня клітинка порожньою.

- getPossibleMovesForOneCar(curr_state, cars, car): Визначає *всі* можливі ходи (на будь-яку кількість вільних клітинок) для заданого автомобіля (car) у поточному стані дошки (curr_state). Перевіряє послідовність клітинок у напрямку руху.

3.3. Опис алгоритму пошуку рішення

В рамках реалізованого програмного забезпечення алгоритми пошуку застосовуються у двох ключових сценаріях:

- для попереднього розрахунку мінімальної кількості рухів для кожного завдання;

- для пошуку рішення у режимі реального часу.

Для розрахунку мінімальної кількості рухів для кожного завдання було обрано алгоритм пошуку в ширину (Breadth-First Search, BFS).

Пошук в ширину (Breadth-First Search, BFS) — це алгоритм обходу або пошуку в графі або дереві. Його основний принцип полягає в тому, що він досліджує всі вершини на поточному рівні глибини, перш ніж перейти до вершин наступного рівня.

Алгоритм починає з початкової вершини (кореня) і спочатку відвідує всіх її безпосередніх сусідів. Потім він відвідує всіх сусідів цих сусідів (тобто всі вершини на відстані 2 від початкової), потім усіх сусідів наступного рівня (на відстані 3) і так далі. Цей процес триває доти, доки не буде знайдено цільову вершину або не будуть відвідані всі досяжні вершини.

Вибір BFS зумовлений його властивістю гарантувати знаходження найкоротшого шляху у просторі станів, що є необхідним для визначення теоретичного мінімуму ходів.

Для реалізації пошуку рішення в режимі реального часу були розглянуті два алгоритми: пошук в ширину (BFS) та алгоритм A*. Алгоритм A* є інформованим пошуковим алгоритмом, який використовує евристичну функцію для оцінки потенційної вартості шляху до цільового стану.

Розроблена евристична функція h(s) для алгоритму A* оцінює вартість досягнення цільового стану s′ з поточного стану s шляхом підрахунку кількості вертикальних транспортних засобів, розташованих праворуч від цільового автомобіля на шляху до виїзду. Ключовою властивістю цієї евристики є її допустимість (admissibility). Це означає, що вона ніколи не переоцінює фактичну кількість ходів, необхідних для завершення гри, оскільки кожен з блокуючих автомобілів має бути переміщений для звільнення виїзду, що є необхідною умовою для виходу цільового автомобіля. Таким чином, застосування допустимої евристики в алгоритмі A* також гарантує знаходження оптимального рішення (найкоротшого шляху).

Однак, ефективність цієї евристики може бути обмеженою у значній частині простору станів. Часто переміщення блокуючих автомобілів вимагає виконання нетривіальних послідовностей ходів, включаючи переміщення інших транспортних засобів, що не є безпосередньо блокуючими. У деяких випадках оптимальна послідовність ходів може тимчасово збільшувати кількість блокуючих автомобілів. У таких ситуаціях дана евристика не забезпечує суттєвого скорочення кількості вузлів, що розглядаються під час пошуку. Незважаючи на зазначені обмеження, дана евристика демонструє ефективність на пізніх стадіях пошуку, зокрема на рівнях пошукового дерева, що відповідають станам, близьким до цільового. Це зумовлено тим, що фінальні кроки вирішення завдання передбачають послідовне переміщення блокуючих вертикальних автомобілів для звільнення виїзду. Чим довшою є ця послідовність фінальних переміщень блокуючих автомобілів, тим

значнішим є скорочення кількості вузлів, що відвідуються алгоритмом A* порівняно з неевристичним пошуком.

Порівняльний аналіз продуктивності алгоритмів BFS та A* за результатами їхнього виконання на різних завданнях представлено у таблиці 3.1.

Таблиця 3.1 - Порівняння продуктивності алгоритмів BFS та A* на тестових завданнях

Завдання	**Автомобілів**			**Мінімально рухів**	**Вузлів відвідано**		**Час виконання (с)**	
	Загалом	Автомобілі	Вантажівки		BFS	A*	BFS	A*
3	6	4	2	14	776	593	0.82	0.86
7	9	9	0	13	4945	2222	16.31	6.56
8	14	11	3	12	952	913	0.91	1.24
19	8	7	1	22	485	481	0.39	0.51
29	12	9	3	31	4328	4289	9.41	13.59
302	13	10	3	23	4737	3984	12.19	13.58
303	10	5	5	23	5795	4447	20.77	21.43
304	13	8	5	23	116	116	0.08	0.11
321	14	10	4	28	853	827	0.72	0.89
325	13	10	3	27	8482	6780	51.93	43.23
332	10	5	5	34	2299	1857	3.32	3.48
333	13	9	4	34	648	543	0.55	0.62
340	13	12	1	43	3043	2994	4.22	6.0

Як свідчать дані, наведені у таблиці 3.1, алгоритм A* демонструє тенденцію до розгортання (відвідування) меншої кількості вузлів у просторі станів порівняно з BFS у більшості тестових завдань. Однак, з точки зору часу виконання, алгоритм A* не завжди демонструє перевагу над BFS. За

результатами тестування на 13 завданнях, A* виявився повільнішим за BFS у 11 випадках, причому різниця у часі виконання варіювалася від 0.04 до 4.18 секунд. У решти 2 випадках A* був швидшим за BFS, з різницею від 8.70 до 9.75 секунд.

Це явище пояснюється додатковими обчислювальними витратами, притаманними алгоритму A*. Зокрема, необхідність підтримки пріоритетної черги (відкритого списку), реалізованої як структура даних "купа", вимагає операцій пошуку дублікатів та реструктуризації купи. Крім того, для кожного відвідуваного вузла алгоритм A* має обчислювати значення оціночної функції f(s)=g(s)+h(s), де g(s) - вартість досягнення стану s, а h(s) - евристична оцінка вартості від s до цільового стану. Ці операції збільшують загальний час виконання алгоритму порівняно з BFS.

Час виконання пошукового алгоритму є критичним параметром в даній реалізації, оскільки він безпосередньо впливає на тривалість блокування користувацького інтерфейсу (GUI) під час виконання пошуку у фоновому режимі, що може негативно позначитися на досвіді гравця. На основі емпіричних даних, представлених у таблиці 3.1, було прийнято рішення використовувати алгоритм BFS для пошуку рішення у режимі реального часу, оскільки він демонструє менший час виконання у більшості тестових сценаріїв порівняно з A*.

Перспективи подальшого розвитку програми включають дослідження можливостей підвищення ефективності пошуку рішення. Одним із напрямків є розробка більш інформативної, але при цьому допустимої евристичної функції для алгоритму A*. З теоретичної точки зору, більш інформативна евристика ближче до фактичної вартості шляху до цілі, що потенційно може скоротити простір пошуку. Наприклад, окрім кількості безпосередньо блокуючих автомобілів, можна розглянути врахування транспортних засобів, які блокують рух самих блокуючих автомобілів. Однак слід враховувати, що збільшення інформативності евристичної функції часто призводить до

зростання її обчислювальної складності, що, у свою чергу, збільшує час на обчислення оцінки для кожного вузла. Таким чином, існує необхідність пошуку оптимального компромісу між інформативністю евристичної функції та її обчислювальною вартістю. Іншим напрямком є аналіз доцільності застосування різних пошукових алгоритмів або їхніх гібридних варіантів для завдань різної структури та складності.

3.3.1. Дослідження щодо оптимізації часу виконання обчислень

За результатами реалізації даного проєкту встановлено, що виконання обчислень у режимі реального часу, зокрема пошуку рішення для надання підказки, потребує значних часових витрат. Це явище потенційно може негативно впливати на досвід взаємодії користувача з інтерфейсом програми, спричиняючи затримки. Існують щонайменше два принципові підходи до вирішення зазначеної проблеми оптимізації часу відгуку системи.

Перший підхід передбачає дослідження та застосування більш ефективних пошукових алгоритмів або розробку вдосконалених евристичних функцій з метою прискорення процесу пошуку рішення у режимі реального часу. Оптимізація самого алгоритму пошуку може скоротити кількість станів, що розглядаються, і, відповідно, час обчислень.

Другий підхід полягає у попередньому обчисленні та зберіганні всіх можливих станів для кожного завдання, а також відповідних оптимальних рішень для кожного стану. Цей набір даних може бути представлений у вигляді орієнтованого графа станів, де вершини відповідають станам, а ребра – переходам між ними. У такому випадку, при запиті підказки, системі буде достатньо знайти відповідний поточний стан гри у збереженій структурі даних та отримати вже обчислене оптимальне рішення.

Критичним аспектом реалізації другого підходу є ефективність пошуку відповідного поточного стану у збереженій структурі даних. В умовах, коли кількість можливих станів для певного завдання є значною, необхідно

забезпечити, щоб обчислювальні витрати на пошук відповідного стану не перевищували вигоди від попереднього обчислення рішень. Зберігання простору станів втрачає свою доцільність, якщо вартість пошуку у цьому просторі стає надмірною.

Лінійний пошук відповідного стану у простій структурі даних (наприклад, вектор або список) демонструватиме низьку продуктивність зі складністю O(N), де N - кількість збережених станів. Натомість, якщо вдасться редукувати складність пошуку до O(logN) (шляхом використання оптимізованого бінарного дерева пошуку) або навіть наблизити її до O(1) (за допомогою ефективно спроектованої хеш-таблиці), концепція використання попередньо обчислених та збережених станів може виявитися перспективною.

Подальші дослідження повинні включати емпіричний аналіз обох запропонованих підходів: розробки вдосконалених евристик для пошуку в реальному часі та реалізації ефективних структур даних для зберігання та швидкого пошуку попередньо обчислених станів. Результати цих досліджень дозволять визначити оптимальне рішення, що забезпечить мінімізацію часу очікування для користувача під час взаємодії з функціоналом підказок у грі "Година пік".

3.4. Опис процесу тестування ігрового застосунку

Валідація коректності функціонування програмного забезпечення є критично важливою, що досягається шляхом тестування. У рамках даного проєкту тестування проводилося на різних рівнях для забезпечення високої якості кінцевого продукту.

Юніт-тестування (модульне тестування) здійснювалося на кожному етапі життєвого циклу розробки. Імплементація нових функціональних можливостей або модифікація існуючих компонентів супроводжувалася

проведенням відповідних юніт-тестів для верифікації коректності роботи окремих програмних одиниць. Методологія проведення юніт-тестів включала аналіз вихідного коду та виконання тестових сценаріїв з моніторингом значень ключових змінних у межах функцій (або модулів) за допомогою налагоджувальних інструментів або інструментації коду.

Після внесення змін або додавання нових функцій також проводилося функціональне тестування. Це тестування здійснювалося з метою верифікації коректності роботи програми в цілому та відсутності несподіваної поведінки або збоїв після інтеграції змін до кодової бази.

Після завершення розробки було проведено комплексне системне тестування. Воно охоплювало всі компоненти користувацького інтерфейсу (GUI) та сценарії взаємодії з ними в межах усіх станів (виглядів) програми (наприклад, головне меню, вибір завдання, ігровий процес). Додатково перевірялася коректність взаємодії та обміну даними між різними модулями та станами системи, що забезпечило відсутність непередбачуваної поведінки.

Дефекти, виявлені в ході кожного тестового прогону (юніт-, функціонального чи системного), були оперативно усунені для забезпечення стабільності та надійності програмного забезпечення.

За результатами проведеного тестування підтверджено відповідність реалізованого програмного забезпечення всім визначеним функціональним та нефункціональним вимогам, що свідчить про його готовність до експлуатації.

Отже, в рамках даного проєкту здійснено розробку програмного забезпечення, що імітує механіку настільної головоломки з ковзаючими блоками під назвою "Година пік".

Функціональність реалізованої системи охоплює базові компоненти, необхідні для відтворення ігрового процесу, включаючи графічне представлення ігрового поля та об'єктів (автомобілів), а також інтуїтивно зрозумілий користувацький інтерфейс для взаємодії. Система інтегрує набір

попередньо сконфігурованих завдань, що відповідають оригінальним варіантам головоломки. Додатково імплементовані функції, такі як система оцінювання продуктивності гравця на основі кількості виконаних ходів, підсистема управління рекордами для відстеження найкращих результатів та механізм надання алгоритмічно згенерованих підказок.

Розроблена програма надає можливість прихильникам даної головоломки відтворювати ігровий процес на комп'ютерній платформі, усуваючи необхідність використання фізичних елементів та ручного налаштування початкових конфігурацій завдань. Система оцінювання та підсистема управління рекордами слугують мотиваційними елементами для гравців, стимулюючи їх до покращення власних результатів та підвищуючи зацікавленість в ігровому процесі.

Для реалізації функціоналу пошуку рішення в режимі реального часу було імплементовано та обрано алгоритм пошуку в ширину (Breadth-First Search, BFS). Також було проведено дослідження ефективності алгоритму A* із застосуванням допустимої евристичної функції. Ця евристика оцінює кількість транспортних засобів, що безпосередньо блокують виїзд цільового автомобіля з ігрового поля. Незважаючи на те, що застосування алгоритму A* з даною евристикою дозволило скоротити загальну кількість розглянутих вузлів у просторі станів порівняно з BFS, емпіричний аналіз показав збільшення часу виконання пошуку у більшості тестових сценаріїв. Це свідчить про значні обчислювальні витрати, пов'язані з управлінням пріоритетною чергою та обчисленням евристичної функції на кожному кроці алгоритму A*.

Перспективи подальших досліджень включають розробку більш інформативних евристичних функцій для алгоритму A*. Врахування додаткових факторів, що впливають на складність вирішення (наприклад, транспортні засоби, що блокують рух самих блокуючих автомобілів), може підвищити точність евристичної оцінки та потенційно призвести до

скорочення часу виконання пошуку в режимі реального часу порівняно з BFS. Іншим напрямком оптимізації є дослідження доцільності попереднього обчислення та зберігання всіх можливих станів для кожного завдання разом з відповідними оптимальними рішеннями. Ефективна організація зберігання (наприклад, за допомогою хеш-таблиць або оптимізованих дерев пошуку) та швидкий пошук у отриманій структурі даних можуть значно прискорити процес надання підказок. Вибір оптимального підходу або їхньої комбінації потребує подальшого емпіричного аналізу для мінімізації часу очікування користувача під час взаємодії з функціоналом підказок.

ВИСНОВКИ

В дипломній роботі було всебічно досліджено теоретичні, методологічні та прикладні аспекти розробки ігрового застосунку головоломки з регульованими рівнями складності з використанням сучасних інформаційних технологій. Робота має комплексний характер і охоплює етапи аналізу, проєктування, реалізації та тестування програмного продукту.

У першому розділі здійснено детальний аналіз поточного стану розробки ігрових застосунків, зокрема логічних головоломок. Визначено мету створення застосунку, сформульовано вимоги до його функціоналу та структури, а також досліджено приклади успішно реалізованих аналогів (Klotski, L’âne rouge, Unblock Me, 15-puzzle). Це дозволило виокремити ключові принципи побудови ефективного інтерфейсу, побудови рівнів складності та забезпечення інтерактивності користувацької взаємодії.

У другому розділі представлено алгоритмічні основи реалізації ігрового застосунку. Розроблено архітектуру програмного забезпечення, здійснено моделювання за допомогою UML-діаграм (діаграма класів, діаграма варіантів використання, діаграма потоку даних, діаграма послідовності), що забезпечило структурованість та логічну узгодженість компонентів системи. Особливу увагу приділено управлінню інтерфейсними станами, глобальним елементам керування та методології програмної реалізації, що визначає чітку послідовність дій під час розробки.

Третій розділ присвячено практичним аспектам реалізації інтерфейсу та внутрішньої логіки ігрового застосунку. Розроблено підсистему зберігання даних, графічний інтерфейс користувача та реалізовано алгоритм пошуку рішення головоломки, враховуючи аспекти оптимізації часу обчислень. У завершальному етапі було проведено тестування застосунку, що дозволило виявити та усунути потенційні помилки, перевірити відповідність функціоналу технічним вимогам і підтвердити стабільність роботи системи.

Таким чином, результати дослідження підтверджують ефективність застосування сучасних методів проєктування та реалізації програмного забезпечення у створенні ігрових застосунків з динамічно регульованою складністю. Реалізований програмний продукт демонструє високий рівень інтерактивності, адаптивності та зручності для користувача, а також може бути використаний як основа для подальших досліджень і розширень функціональності в межах освітніх, розважальних або тренувальних систем.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Rabinovich, Z., & Sherman, R. (2006). Heuristics for the Rush Hour puzzle. Journal of Artificial Intelligence Research, 26, 185-205.
2. Unblock Me FREE – безкоштовне завантаження та відтворення у Windows | Microsoft Store - https://apps.microsoft.com/detail/9nblggh189f2?hl=uk-ua&gl=UA
3. 15 Puzzle -Sliding Puzzle Game – Додатки в Google Play - https://play.google.com/store/apps/details?id=com.zeeron.puzzle15&hl=uk
4. 15 Puzzle - Fifteen Game Chall – Додатки в Google Play - https://play.google.com/store/apps/details?id=com.spicags.fifteen&hl=uk
5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
6. Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Prentice Hall.
7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 4. Pressman, R. S., & Maxim, B. R. (2020). Software Engineering: A Practitioner's Approach (9th ed.). McGraw-Hill.
8. Beck, K. (2002). Test-Driven Development: By Example. Addison-Wesley.
9. Myers, G. J., Sandler, C., & Badgett, T. (2011). The Art of Software Testing (3rd ed.). John Wiley & Sons.
10. Millington, I., & Funge, J. (2009). Artificial Intelligence for Games (2nd ed.). CRC Press.
11. Rabinovich, Z., & Sherman, R. (2006). Heuristics for the Rush Hour puzzle. Journal of Artificial Intelligence Research, 26, 185-205.
12. Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97–109. 11. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of

Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100–107.

13. Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.
14. Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice-Hall.
15. Sedgewick, R. (2002). Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching (3rd ed.). Addison-Wesley Professional.
16. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in Java (3rd ed.). Pearson.
17. Goodrich, M. T., & Tamassia, R. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.
18. McConnell, S. (2004). Code Complete: A Practical Handbook of Software Construction (2nd ed.). Microsoft Press.
19. Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley.
20. Kerievsky, J. (2004). Refactoring to Patterns. Addison-Wesley.
21. Alur, D., Crupi, J., & Malks, D. (2001). Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall..
22. Tanenbaum, A. S., & Van Steen, M. (2006). Distributed Systems: Principles and Paradigms (2nd ed.). Prentice Hall.
23. Maguire, S. (1993). Writing Solid Code. Microsoft Press.
24. Binder, R. V. (1999). Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional
25. Beizer, B. (1990). Software Testing Techniques (2nd ed.). Van Nostrand Reinhold.
26. Korf R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial Intelligence, 27(1), 97–109.
27. Copeland, L. (2003). A Practitioner's Guide to Software Test Design. Artech House.

28. Jackson, P. C. (1985). Introduction to Artificial Intelligence (2nd ed.). Dover Publications.
29. Pearl, J. (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley.
30. JSON. (n.d.). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/JSON

БІБЛІОГРАФІЧНА ДОВІДКА

Тема дипломної роботи: “Реалізація ігрового застосунку з регульованими рівнями складності ”

Обсяг пояснювальної записки: 75 аркушів.

Дата закінчення роботи: 9 червня 2025 р.

Підпис студента _____