Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Розробка гри-лабіринту з елементами самонавчання і ускладнення проходження, враховуючи дії гравця
(тема)

Виконав:
студент 4 курсу, групи ІТІНФ-20-2
Прокопенко Р.П.
(прізвище, ініціали)

Спеціальності 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник ст. викл. Кіношенко Д.К.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ Кобилін О.А.
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Прокопенко Ростиславу Петровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка гри-лабіринту з елементами самонавчання і ускладнення проходження, враховуючи дії гравця

затверджена наказом університету від 20 травня 2024 року № 464 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 25 травня 2024 р.

3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі, офіційна документація мови Kotlin, офіційна документація платформи Android.

4. Перелік питань, що потрібно опрацювати в роботі

1. Аналіз алгоритмів створення та відображення лабіринту.
2. Аналіз алгоритмів для створення лабіринту та опис основних інструментів.
3. Реалізація постановки задачі та тестування.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) Актуальність проблеми утримання гравців у грі і створення нового досвіду для них, постановка задачі.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	08.04.2024	
2	Аналіз завдання, підбір літератури	08.04.24-13.04.24	
3	Аналіз літератури з досліджуваної проблеми	14.04.24-23.04.24	
4	Аналіз технічних засобів	24.04.24-27.04.24	
5	Розробка алгоритмів	28.04.24-04.05.24	
6	Програмна реалізація	05.05.24-15.05.24	
7	Оформлення пояснювальної записки	16.05.24-17.05.24	
8	Перевірка на плагіат	27.05.24	
9	Рецензування	28.05.24	
10	Підготовка презентації та доповіді	29.05.24-02.06.24	
11	Занесення роботи в електронний архів	03.06.24	
12	Попередній захист кваліфікаційної роботи	03.06.24	

Дата видачі завдання 8 квітня 2024 р.

Студент _____
(підпис)

Керівник роботи _____ ____ ст. викл. Кіношенко Д.К.___
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 64 с., 29 рис., 30 джерел.

АЛГОРИТМ КРАСКАЛА, ЛАБІРИНТ, АЛГОРИТМ ПРИМА, РЕКУРСИВНЕ ВІДСТЕЖЕННЯ, ANDROID ЗАСТОСУНОК.

Об’єктом роботи є вивчення алгоритмів побудови лабіринтів та його реалізація на їх основі лабіринту з елементами ускладнення та навчання.

Метою роботи є розробка алгоритмів для побудови лабіринту та його створення, що базуються на алгоритмах Краскала та Прима, інтеграція Python коду, створення алгоритму ускладнення в залежності від дій гравця, розробка Android застосунку. Використання методів графічного відображення лабіринту та руху користувача, розробка алгоритму ускладнення лабіринту в залежності від дій гравця та від поточного його стану.

У результаті роботи здійснена програмна реалізація Android застосунку для гри у лабіринт.

KRUSKAL ALGORITHM, MAZE, PRIM ALGORITHM, RECURSIVE TRACKING, ANDROID APPLICATION.

The object of the work is to study the algorithms for constructing mazes and its implementation on their basis of a maze with elements of complication and learning.

The aim of the work is to develop algorithms for building a maze and creating it based on the Kruskal and Prima algorithms, integrating Python code, creating a complication algorithm depending on the player’s actions, developing an Android application. Use of methods of graphical display of the maze and user movement, development of an algorithm for complicating the maze depending on the player’s actions and current state.

The methods of graphical display of the maze and user movement are used, and an algorithm for complicating the maze depending on the player’s actions and current state is developed.

As a result of the work, the software implementation of an Android application for playing the maze was carried out.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

2D – стиль графіки у комп’ютерних іграх, де вся дія відбувається в двовимірному просторі, використовуючи лише висоту та ширину

3D – стиль графіки у комп’ютерних іграх, де вся дія відбувається в тривимірному просторі, включаючи висоту, ширину та глибину

Android – операційна система, розроблена компанією «Google», яка в основному використовується для мобільних пристроїв, таких як смартфони та планшети

Auth – скорочена форма автентифікація (англ. authentication)

UI – все, з чим користувач взаємодіє в програмі або на вебсайті, тобто, інтерфейс

API – це набір правил і визначень, що дозволяють одній програмі взаємодіяти з іншою

ВСТУП

У сучасному світі відеоігри є не лише засобом розваги, а й потужним інструментом для розвитку креативності, логічного мислення та навичок вирішення проблем. Розвиток технологій дозволяє створювати ігри, які адаптуються до дій гравця, забезпечуючи більш особистісне та захоплюючий ігровий досвід. Робота має на меті розробити гру-лабіринт для платформи Android, яка використовує елементи самонавчання та ускладнення проходження залежно від дій гравця. Основна унікальність проєкту полягає у застосуванні двох алгоритмів генерації лабіринтів – Пріма і Краскала – що дозволяють створювати складні та непередбачувані ігрові сценарії.

Лабіринти, як інтерактивні ігрові середовища, вже довгий час є популярними у графічних пригодах, де кожне рішення гравця впливає на подальший розвиток подій. Впровадження алгоритмів машинного навчання дозволяє не просто створити мапу з перешкодами, але й забезпечити ігрову систему, яка аналізує стратегії гравця та адаптує виклики для підвищення складності та варіативності ігрового процесу. Це створює глибоку іммерсивну взаємодію та підвищує повторне споживання гри.

Актуальність даної роботи полягає у впровадженні технологічних рішень у розробку ігор, які здатні адаптуватися та реагувати на індивідуальні дії гравця, що є ключовим аспектом у створенні ефективних та захоплюючих ігрових досвідів. Реалізація даного проєкту передбачає не тільки технічний розвиток ігрових механік, але й поглиблене дослідження в області алгоритмічних знань, що може знайти застосування в інших сферах розробки програмного забезпечення.

Робота сприятиме подальшому розвитку комп’ютерних наук та інтерактивних технологій, обіцяючи не тільки покращити ігровий досвід, але й відкрити нові можливості для дослідження в цій захоплюючій і динамічній галузі.

1 АНАЛІЗ АЛГОРИТМІВ СТВОРЕННЯ ТА ВІДОБРАЖЕННЯ ЛАБІРИНТУ

1.1 Визначення що таке гра-лабринт

Гра-лабіринт – це вид комп’ютерної або відеоігри, в якій основним завданням для гравця є навігація через складну і часто заплутану мережу шляхів або проходів для досягнення певної мети [1]. Ці ігри зосереджуються на тестуванні спостережливості, просторового сприйняття, пам’яті, та логічного мислення гравців. Лабіринти можуть бути фізичними структурами, такими як ті, що зустрічаються в тематичних парках, але в контексті відеоігор вони зазвичай виконані віртуально і можуть містити різні перешкоди, загадки та противників [2].

Основні характеристики гри-лабіринту включають:

– структура лабіринту: комплекс маршрутів або стін, які утворюють мережу шляхів, де деякі можуть вести до цілі, а інші – в тупики;

– ціль гри: часто це досягнення кінця лабіринту або знаходження конкретних об’єктів чи розв’язання загадок всередині лабіринту;

– перешкоди та виклики: пастки, противники, закриті двері або загадки, які гравець мусить подолати або вирішити для просування;

– динаміка гри: лабіринти можуть бути статичними, де структура не змінюється, або динамічними, де лабіринт адаптується чи змінюється у відповідь на дії гравця.

Гра-лабіринт може бути реалізована в різних жанрах, від простих головоломок до складних пригодницьких або рольових ігор. Цей вид ігор особливо ефективний для розвитку навичок критичного мислення та стратегічного планування, оскільки вимагає від гравців аналізувати своє оточення, планувати маршрути та швидко приймати рішення під тиском.

Використання лабіринтів у відеоіграх також допомагає створити захоплюючу іммерсивну атмосферу, яка затягує гравців у віртуальний світ гри.

1.2 Види ігор-лабіринтів

Ігри-лабіринти можуть варіюватися за стилем, складністю та цілями гри, пропонуючи різноманітні ігрові досвіди для різних типів гравців. Нижче представлено кілька основних видів ігор-лабіринтів, кожен з яких має унікальні характеристики та ігрові механіки [3].

Серед основних видів ігор-лабіринтів виділяють:

– класичні лабіринти: ці ігри фокусуються на основній меті – знайти вихід із заплутаної мережі шляхів. Вони можуть бути простими з точки зору графіки та дизайну, але вимагають від гравців хороших навичок орієнтування у просторі. Прикладами можуть бути традиційні 2D або 3D лабіринти, де гравець керує персонажем через серію коридорів та проходів;

– головоломки з лабіринтами: у цих іграх лабіринти використовуються як частина більших головоломок. Гравцям потрібно не тільки знайти шлях, але й вирішувати різні задачі для руху вперед. Це може включати знаходження ключів для відмикання дверей, активацію механізмів, або маніпулювання елементами оточення;

– пригодницькі ігри з елементами лабіринту: у таких іграх лабіринти інтегровані у більший світ пригоди. Гравці досліджують складні локації, взаємодіють з персонажами, виконують завдання і розкривають сюжетні лінії. Лабіринти тут служать не просто як перешкоди, а як частина історії та світу гри;

– акціонні ігри з лабіринтами: ці ігри включають швидкісні елементи та бої, де лабіринти стають ареною для зіткнень з ворогами. Гравці повинні не тільки знаходити шляхи та вирішувати головоломки, але й управляти боями, ухилятися від атак і підбирати стратегії в залежності від ситуації;

– ескейп-руми: ці ігри зазвичай відбуваються у закритому просторі, де гравці мають обмежений час для виходу. Лабіринт складається з серії кімнат або зон, кожна з яких містить головоломки та загадки, що необхідно розгадати для прогресу в грі;

– ігри на виживання з елементами лабіринту: у цих іграх лабіринти використовуються для створення напруженого і стиснутого атмосферу, де ресурси обмежені, а навколишнє середовище часто вороже. Гравці повинні ефективно управляти своїми ресурсами та шукати шляхи для виживання у складних умовах.

Всі ці види ігор-лабіринтів демонструють величезну різноманітність у підходах до дизайну ігрового процесу, кожен з яких пропонує унікальні виклики та вимагає від гравців різних навичок для успішного проходження.

1.3 Порівняння подібних ігор-лабіринтів

В цьому розділі буде розглянуто головні проєкти які побудовані навколо логіки лабіринтів, побачимо їх головні переваги, досягнення і виокремимо проблеми, які варто уникати при створенні власної гри-лабіринту.

1.3.1 Гра «Pac-Man»: огляд та її значення у жанрі ігор-лабіринтів

«Pac-Man», створений Тору Іватані та випущений компанією «Namco» у 1980 році, швидко став однією з найвідоміших відеоігор у світі. Спочатку задумана як гра, яка б мала привабити як чоловіків, так і жінок, «Pac-Man» відрізняється своїм простим, але захоплюючим геймплеєм. Гравці керують персонажем «пакманом», який повинен з’їсти всі крапки у лабіринті, уникаючи при цьому зіткнення з привидами, загальний вигляд гри показано

на рисунку 1.1. Гра включає кілька рівнів складності, які збільшуються з кожним новим лабіринтом.

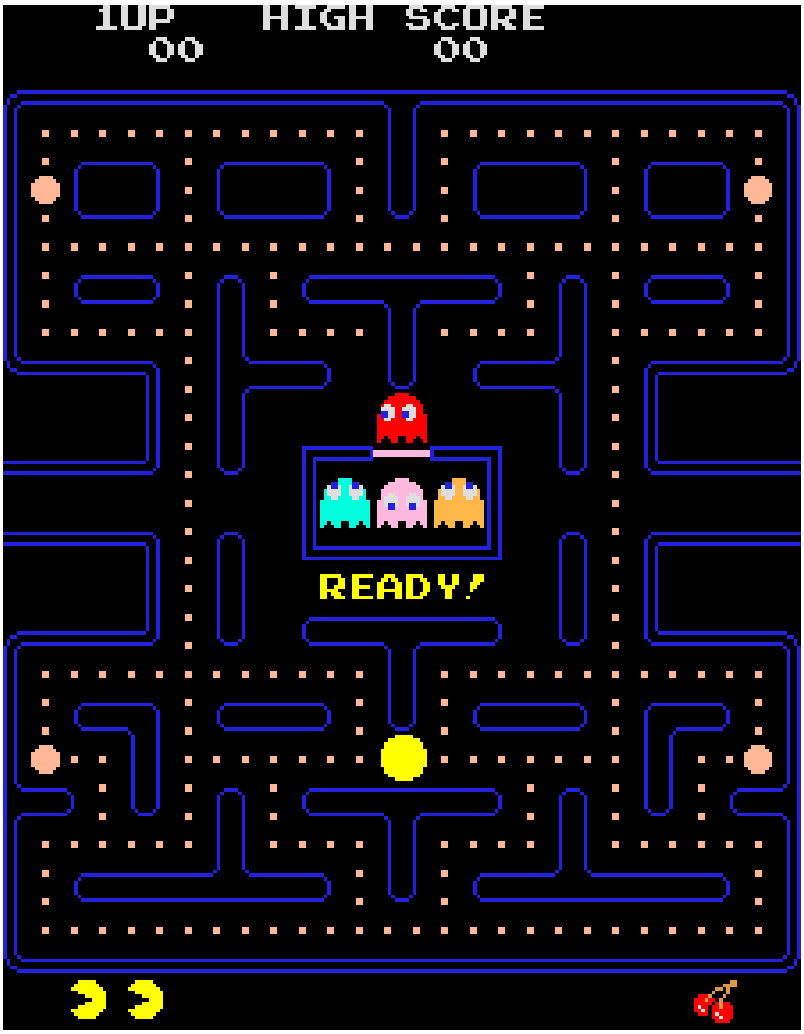

Рисунок 1.1 – Скріншот базової ігрової сесії

Однією з ключових особливостей «Pac-Man» є його здатність інтегрувати простий дизайн і глибокий ігровий процес. Гра відзначається як першопрохідцем у створенні аркадних ігор-лабіринтів і має значний вплив на ігрову культуру. Її успіх сприяв виникненню численних продовжень, а також товарів, телевізійних шоу та навіть музики.

«Pac-Man» також виділяється своїм культурним впливом, ставши іконічним символом 80-х років та відеоігрової індустрії в цілому. Гра була однією з перших, що показала потенціал аркадних ігор як форми сучасного мистецтва і розваги.

Попри свій величезний успіх, «Pac-Man» також виявив деякі проблеми, які можуть бути важливими для розробників, що бажають створювати подібні ігри. Наприклад, оригінальна гра мала обмеження у варіативності лабіринтів, що могло призвести до певної монотонності після довгих годин гри. Також, оскільки гра заснована на швидкісній реакції та запам’ятовуванні патернів, для деяких гравців вона може здатися надмірно складною або стресовою.

Створення ігор-лабіринтів на зразок «Pac-Man» вимагає зосередження на балансі між доступністю гри та її викликом. Розробники повинні враховувати як створення цікавих та варіативних лабіринтів, так і потреби різних категорій гравців. Це включає розгляд механізмів, які можуть знизити фрустрацію без зниження загальної складності гри, такі як різні рівні складності або системи підказок.

1.3.2 «The Witness» гра, як приклад гри-ідеалу у жанрі ігор-лабіринтів

«The Witness», розроблена Джонатаном Блоу, творцем культової гри «Braid», є інтелектуальною головоломкою в відкритому світі, яка була випущена у 2016 році [4]. Гра занурює гравців у розгалужений лабіринт з великої кількості загадок, розкиданих по красивому і взаємопов’язаному острову. «The Witness» ставить перед гравцями завдання знайти шлях через фізичний та метафоричний лабіринт, досліджуючи теми пізнання, сприйняття та епіфанії.

«The Witness» вирізняється завдяки своєму унікальному підходу до побудови лабіринтів та головоломок. Кожна загадка є логічним продовженням попередньої, поступово вводячи гравців у складніші механіки без явного текстового пояснення. Ця нелінійна структура дозволяє гравцям самостійно вибирати свій шлях і темп проходження, що підкреслює відчуття

дослідження та відкриття. Прикладом є рисунок 1.2, де видно одразу декілька кімнат з загадками.

Рисунок 1.2 – Скріншот одного з лабіринтів у грі

Візуальний та аудіальний дизайн «The Witness» сприяє глибокому зануренню в ігровий світ. Кожен куточок острова дизайнований так, щоб викликати роздуми і дослідження, а кольорова палітра та звукове оформлення посилюють атмосферу місця.

Однак «The Witness» також має кілька викликів, особливо стосовно доступності гри. Загадки можуть бути надзвичайно складними, що іноді призводить до фрустрації серед гравців, особливо тих, хто не має великого досвіду з головоломками. Відсутність явних підказок або напрямків може зробити деякі елементи ігрового процесу занадто непрозорими. Як приклад такої загадки виступає рисунок 1.3, де чітко видно відсутність підказок і складність загадки.

«The Witness» навчає розробників важливості творчого підходу до дизайну головоломок та лабіринтів, зосереджуючи увагу на необхідності створення цілісного і зв’язного ігрового світу. Розробники можуть взяти на замітку важливість балансу між складністю та доступністю, намагаючись

забезпечити гравцям відчуття задоволення від розгадування загадок, але при цьому не відштовхуючи їх надмірною складністю.

Рисунок 1.3 – Приклад однієї головоломки з гри

1.4 Головні висновки з аналізу ігор «Pac-Man» та «The Witness» щодо розробки ігор-лабіринтів

Аналіз ігор «Pac-Man» та «The Witness» дозволяє виділити ключові аспекти, які можуть сприяти успішному дизайну ігор-лабіринтів. Обидві гри мають вплив на жанр, але кожна з них представляє унікальні підходи, які можна адаптувати та використовувати в майбутніх проєктах. І хоч між ними і прірва у десятки років, деякі з механік уставлених «Pac-Man» живуть до сих пір, і є основою для ігор подібних «The Witness». Багато розробників саме звертаються до них регулярно, а багато університетів, у межах курсів по розробці ігор, вносять обов’язкові практики на яких треба грати у ці ігри і також намагатися виділити для себе щось корисне.

1.4.1 Інтуїтивно зрозумілі механіки

«Pac-Man» демонструє важливість простоти в дизайні гри, де базові правила легко зрозумілі, але глибокі за своєю сутністю. Гра вимагає від гравців швидкої реакції та стратегічного планування, що є доступним для широкого кола гравців.

Важливо створити механіки, які є легкими для освоєння, але пропонують достатньо глибини для залучення гравців. При цьому варто і не забувати про постійну ротацію цих механік, аби гравцю не набридало робити одне, й те саме, бо вийде ситуація «сміттєвих баків» з гри «The Last of Us», коли майже на кожному рівні була механіка пересування сміттєвого баку аби залізти на нього і пройти далі (рис 1.4). Ця механіка дуже швидко набридла гравцям і стала об’єктом жартів.

Рисунок 1.4 – Скріншот з «The Last of Us» з механікою пересування сміттєвого баку

1.4.2 Складність та доступність

«The Witness» навчає важливості балансу між складністю та доступністю. Ігри мають бути викликаючими, але не відштовхувати гравців через надмірну складність. Розробники мають звернути увагу на введення поступових навчальних елементів, які допоможуть гравцям освоїти складніші механіки без зовнішніх підказок.

Не треба вважати гравців за дурних і вести їх за руку через усі рівні явно підказуючи що робити, ідеальним рішенням може бути побудова перших рівнів таким чином, аби гравець через дуже прості задачі сам міг зрозуміти механіки і отримати задоволення від процесу отримання цих знань. Завжди можна навчити гравця новим механікам за допомогою оточуючого світу, вдало покладена записка щодо роботи нової зброї, чи вдало так вчасно прибулий ворог, який на своєму прикладі покаже як користуватися новою зброєю чи здатністю.

1.4.3 Іммерсивний світ

Обидві гри підкреслюють важливість створення захоплюючого іммерсивного світу. Незалежно від того, чи є це геометрично простий світ, як у «Pac-Man», чи деталізований 3D острів у «The Witness», гравці повинні відчувати себе частиною цілісного світу, що спонукає їх досліджувати та взаємодіяти з ним. Вони повинні вірити що знаходяться там, це досягається шляхом спрощення історії в певних деталях. Гравцю не обов’язково завжди мати сюжет про героя, який рятує принцесу, він може побачити просту історію про лицаря-невдаху, який дав клятву виграти сто двобоїв, але зовсім не вміє битися. Або звичайну історію про кохання, яка зацікавить гравця якраз своєю простотою і звичністю для кожного з нас.

1.4.4 Відповідність ігрового дизайну та цілей гри

Розробка ігор-лабіринтів повинна враховувати цілі та очікування гравців. Як «Pac-Man», так і «The Witness» мають чітко визначені цілі, але подають їх через різні ігрові механіки та стилі. Розробники повинні визначити, який досвід є найбільш важливим для їхньої гри та відповідно адаптувати дизайн. Гравець не дуже любить коли він не розуміє що робити чи як його дії допоможуть йому дійти до цілі. Але іноді подібна загадковість може існувати, до певного часу, гарним прикладом є гра «Death Stranding». В ній гравець перший час не розуміє своєї цілі і навіщо він доставляє пакунки. Але з часом гравець починає розуміти навіщо це все і починає з новою силою виконувати завдання.

1.4.5 Адаптивність до відгуків гравців

Важливо створювати ігри, які реагують на відгуки гравців та адаптуються до них. Це може включати адаптивність складності, зміну механік гри для підвищення задоволеності гравців, або навіть дизайн ігрових елементів, які можуть бути модифіковані на основі гравецької поведінки.

Вивчення та впровадження цих принципів може допомогти розробникам створити ігри-лабіринти, які будуть не тільки викликаючими та інтерактивними, але й даватимуть гравцям глибокі та задовільні досвіди.

1.5 Постановка задачі

Таким чином, побудова гри-лабіринту є актуальним завданням для спроби дослідити і покращити головоломки по проходженню лабіринту. Тому ставиться завдання розробки алгоритмів побудови лабіринту та

алгоритмів ускладнення цих лабіринтів у комбінації з алгоритмами руху гравця, з використанням технологій сучасної Android розробки та інтеграцією Python коду у Android застосунок.

Об’єктом роботи є вивчення алгоритмів побудови лабіринтів та його реалізація на їх основі лабіринту з елементами ускладнення та навчання.

Метою роботи є розробка алгоритмів для побудови лабіринту та його створення, що базуються на алгоритмах Краскала та Прима, інтеграція Python коду, створення алгоритму ускладнення в залежності від дій гравця, розробка Android застосунку. Використання методів графічного відображення лабіринту та руху користувача, розробка алгоритму ускладнення лабіринту в залежності від дій гравця та від поточного його стану. Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз існуючих алгоритмів побудови лабіринту;

– провести аналіз існуючих ігор-лабіринтів задля визначення елементів, які можуть бути використані;

– розробити алгоритми побудови лабіринту у матричному вигляді;

– реалізувати алгоритм переведення матричного лабіринту у графічний, з використанням базових інструментів та бібліотек Android розробки;

– реалізувати алгоритм ускладнення лабіринту та тасування алгоритмів;

– реалізувати алгоритм таймінгу та випадкових ускладнень при проходженні;

– реалізувати логіку підрахунку винагороди за рівні та загальну їх статистику.

2 АНАЛІЗ АЛГОРИТМІВ ДЛЯ СТВОРЕННЯ ЛАБІРИНТУ ТА ОПИС ОСНОВНИХ ІНСТРУМЕНТІВ

2.1 Процес вибору інструментів для розробки

У процесі розробки програмного забезпечення, особливо при створенні складного і функціонально насиченого програмного забезпечення, як наприклад мобільні ігри, вибір правильних інструментів є критично важливим. Ці інструменти не тільки спрощують процес розробки, але й впливають на якість, продуктивність і сумісність кінцевого продукту. При розробці Android-застосунку гри-лабіринту було обрано ряд передових технологій та інструментів, які є популярними в індустрії і використовуються великими технологічними компаніями, включаючи учасників групи FAANG.

Кожен інструмент у цьому наборі був вибраний за його здатність забезпечити необхідну функціональність, ефективність розробки, а також підтримку інноваційних функцій, що необхідні для створення сучасного мобільного застосунку. Використання таких інструментів дозволяє розробникам зосередитися на створенні якісного ігрового контенту, забезпечуючи при цьому високу продуктивність і гладкість роботи програми на різноманітних пристроях. Це також економить час на розробку і гроші, що в наш час дуже важливо, бо ігри вже давно вийшли за межі дешевих проєктів, створених у гаражі.

Далі буде розглянуто кожен з вибраних інструментів, їхні ключові особливості, переваги та потенційні недоліки, що дозволить краще зрозуміти, як вони спільно сприяють створенню ефективного та захоплюючого ігрового застосунку.

2.2 Вибір мови Kotlin, в якості основної для розробки Android-застосунків

Kotlin є сучасною мовою програмування, яка була створена компанією JetBrains і вперше представлена у 2011 році. З того часу вона швидко набула популярності серед розробників, особливо в сфері Android-розробки, після того як Google оголосив її офіційною мовою для розробки Android-застосунків у 2017 році. Ключовою перевагою Kotlin є її сумісність з Java, що дозволяє розробникам легко інтегрувати Kotlin у свої існуючі проєкти, а також користуватися всією екосистемою Java [5].

Серед переваг Kotlin виділяють:

– сучасний синтаксис: Kotlin має більш сучасний і виразний синтаксис порівняно з Java, що знижує кількість шаблонного коду і робить код більш читабельним та легшим для підтримки [6];

– безпека щодо null: мова забезпечує вбудовану підтримку обробки null, що допомагає уникнути NullPointerException. Кожен тип у Kotlin має строге розрізнення між nullable та non-nullable версіями, змушуючи розробників явно вказувати, може змінна бути null чи ні;

– підтримка функціонального програмування: Kotlin включає елементи функціонального програмування, такі як вирази лямбда, оператори вищого порядку, і незмінність даних, що дозволяє писати більш гнучкий та безпечний код;

– висока продуктивність: Kotlin компілюється у байт-код Java, що забезпечує високу продуктивність на платформі Android, порівнянну з продуктивністю Java, хоча й трохи нижчу, через додатковий функціонал, який було реалізовано в Kotlin;

– широка екосистема: завдяки сумісності з Java, Kotlin може використовувати всі бібліотеки Java, включаючи розширені фреймворки та інструменти для розробки.

Недоліки Kotlin:

– крута крива навчання: для розробників, які не знайомі з сучасними концепціями програмування, Kotlin може виявитися складнішим для освоєння порівняно з Java через його функціональні можливості та синтаксичні особливості;

– час компіляції: хоча продуктивність виконання Kotlin і Java схожа, компіляція коду на Kotlin може займати трохи більше часу, що може вплинути на швидкість розробки, особливо в великих проєктах;

– обмежені ресурси для навчання: незважаючи на широку підтримку спільноти та значний обсяг ресурсів, Kotlin все ще має менше навчальних матеріалів та готових прикладів порівняно з більш старими мовами, такими як Java;

– російське коріння: хоч і компанія JetBrains зареєстрована у Чехії, її основний офіс довгий час був на території Росії. Все це веде за собою доволі очевидні тривоги щодо безпечності використання мови.

З урахуванням цих аспектів, Kotlin є відмінним вибором для сучасної розробки Android-застосунків, пропонуючи ідеальне співвідношення між продуктивністю, безпекою і швидкістю розробки.

2.3 Jetpack Compose як основна бібліотека для розробки UI застосунку

Jetpack Compose – це сучасний інструментарій для побудови інтерфейсу користувача на платформі Android, розроблений командою Google [7]. Він використовує декларативний підхід до опису інтерфейсу користувача, що дозволяє розробникам більш інтуїтивно та ефективно створювати гнучкі UI компоненти та легше їх підтримувати [8]. Jetpack Compose був офіційно представлений у 2021 році і з того часу активно залучає спільноту Android-розробників завдяки своїм перевагам у побудові сучасних інтерфейсів.

Головними перевагами Jetpack Compose є:

– декларативний UI: використання декларативного стилю програмування дозволяє розробникам описувати, що вони хочуть відобразити, а не як це робити. Це робить код інтерфейсу більш зрозумілим і легшим для читання та підтримки;

– компонентний підхід: Compose сприяє створенню перевикористовуваних UI компонентів, які можуть бути скомбіновані для створення складних інтерфейсів без зайвого дублювання коду. На рисунку нижче (рис. 2.1) можна побачити такий приклад, де завчасно було створено MessageCard і надалі він використовується будь де, без необхідності кожний раз верстати той самий компонент, а лише передати необхідні параметри;

– інтеграція з Kotlin: розроблено з урахуванням можливостей Kotlin, Compose максимально використовує особливості мови, такі як лямбда-функції та корутини, для створення більш виразного та функціонального коду;

– швидка ітерація: завдяки інструментам, таким як Preview та Hot Reload, розробники можуть швидко бачити результати своїх змін у коді інтерфейсу, не перезапускаючи весь застосунок;

– продуктивність: Compose оптимізований для швидкої рендерингу компонентів UI, що дозволяє досягати гладкості анімацій та відгуків інтерфейсу.

```
if (showAlertDialog) {
    AlertDialog(
        onDismissRequest = {   You, 16/05/2024 1:06 am • Uncommitted changes
            showAlertDialog = false
        },
        title = { Text("Login Failed") },
        text = { Text("Something went wrong, please try again later") },
        confirmButton = {
            Button(onClick = { showAlertDialog = false }) {
                Text("OK")
            }
        }
    )
}
```

Рисунок 2.1 – Приклад компонентного підходу

У контексті розробки програми гри-лабіринту, Jetpack Compose був використаний для побудови всього інтерфейсу користувача. Це включає ігрові екрани, результати, а також діалоги та інші взаємодії. Compose дозволив створити багатий і гнучкий дизайн, що адаптується до різних розмірів екранів та орієнтацій пристрою, підвищуючи зручність користувачів.

Серед потенційних недоліків Jetpack Compose можна виділити:

– крива навчання: для розробників, які звикли до імперативних патернів створення UI, перехід на декларативний підхід може бути викликом;

– молодість технології: як відносно нова технологія, Compose все ще проходить через процес зрілості, що може включати зміни в API та невелику кількість ресурсів для деяких рідкісних задач.

Jetpack Compose відіграє ключову роль у створенні сучасних та ефективних програм для Android, дозволяючи розробникам зосередитися на якості їхнього дизайну інтерфейсу, не жертвуючи при цьому продуктивністю.

До впровадження Jetpack Compose, XML був основним інструментом для декларування інтерфейсів користувача в Android-програмах. XML використовувався для визначення структури UI, таких як кнопки, тексти, списки та інші віджети. Розробники вносили параметри компонентів UI в XML-файли, а потім управляли їх поведінкою за допомогою коду Java або Kotlin.

Характеристики XML:

– розділення дизайну та логіки: XML дозволяє відокремити дизайн інтерфейсу від бізнес-логіки застосунку, що спрощує співпрацю дизайнерів та розробників;

– візуальні редактори: XML підтримується багатьма візуальними редакторами інтерфейсів, які дозволяють драг-енд-дроп розробку, забезпечуючи швидке прототипування;

– стандартизація: до введення Compose, XML був стандартом для розробки Android UI, що забезпечило широку підтримку та численні ресурси для навчання.

Вибір між Jetpack Compose і XML постає перед початком кожного з проектів, багато факторів вказують що саме Jetpack Compose може бути кращим вибором для розробки інтерфейсів у порівнянні з традиційним підходом з використанням XML, а саме, через:

– декларативний підхід: Compose використовує декларативний стиль програмування, який зосереджується на тому, що має бути відображено, замість того, як це має бути зроблено. Це спрощує код і робить його більш інтуїтивним;

– єдиність коду: у Compose усі елементи інтерфейсу створюються і управляються за допомогою Kotlin, що зменшує потребу перемикатися між мовами та файлами та покращує узгодженість і масштабованість проєктів;

– вбудована реактивність: Compose підтримує реактивні патерни більш природньо, автоматично оновлюючи UI при зміні даних, що вимагає менше коду та забезпечує більш плавну роботу користувацького інтерфейсу;

– інтеграція з сучасними інструментами Kotlin: використання Kotlin з його корутинами, флоу та іншими функціональностями надає Compose додаткову міць та ефективність при розробці складних інтерфейсів.

Хоча XML має свої переваги, особливо в контексті підтримки та ресурсів, Jetpack Compose пропонує сучасний, ефективний та елегантний спосіб розробки інтерфейсів користувача на Android. Він спрямований на майбутнє розробки програмного забезпечення з більшою взаємодією, інтуїтивністю та реактивністю, що робить його ідеальним вибором для нових проєктів, що прагнуть скористатися перевагами останніх технологічних інновацій.

2.4 Python у розробці алгоритмів побудови лабіринту

Python є однією з найпопулярніших мов програмування в світі, відомою своєю універсальністю, легкістю вивчення та великою екосистемою. Ця мова є особливо популярною серед вчених, інженерів, а також розробників програмного забезпечення, завдяки своїй гнучкості та ефективності у вирішенні широкого спектру програмних задач.

Переваги Python для розробки алгоритмів:

– чистота та простота синтаксису: Python має виразний та читабельний синтаксис, що робить мову ідеальною для алгоритмічного кодування, де чистота та простота важливі для розуміння та налагодження алгоритмів;

– велика стандартна бібліотека та сторонні бібліотеки: Python надає широкий спектр вбудованих та сторонніх модулів, які можуть бути використані для математичних обчислень, наукових досліджень, роботи з графами, а також для реалізації алгоритмів оптимізації та планування;

– підтримка візуалізації: Python має потужні інструменти візуалізації, такі як Matplotlib, Seaborn, та Bokeh, що дозволяють легко візуалізувати складні дані та алгоритмічні процеси, включаючи побудову лабіринтів;

– інтеграція та розширюваність: Python дозволяє легко інтегрувати системи, написані на інших мовах програмування, та підтримує розширення, що може бути написане на C або C++, для покращення продуктивності критичних за часом частин.

Хоча Kotlin є основною мовою для розробки на Android і пропонує відмінні можливості для розробки застосунків, вибір Python для реалізації алгоритмів побудови лабіринту має кілька переваг [9]:

– спеціалізовані бібліотеки: Python має доступ до бібліотек, спеціалізованих на алгоритмічних та наукових обчисленнях, таких як NumPy, SciPy, і NetworkX, які ідеально підходять для обробки графів та інших складних структур даних, що є ключовими для побудови лабіринтів;

– продуктивність у розробці: Python дозволяє швидше реалізувати складні алгоритми з меншим обсягом коду порівняно з Kotlin, що може прискорити експерименти та ітерації під час розробки алгоритмів;

– візуалізація для налагодження: легкість візуалізації в Python допомагає у налагодженні та тестуванні алгоритмів побудови лабіринту, надаючи інтуїтивно зрозумілі зображення структур даних та процесів їх обробки.

У підсумку, хоча Kotlin може бути використаний для написання алгоритмів, Python пропонує більш ефективні та спеціалізовані інструменти для обробки та візуалізації складних алгоритмів, що робить його більш підходящим для цього аспекту розробки гри-лабіринту.

2.5 Використання бібліотеки Chaquopy для інтеграції Python у Android-застосунки

Chaquopy є плагіном для Android Studio, який дозволяє інтегрувати Python код у проєкти Android, написані на Kotlin або Java. Це ефективний інструмент для розробників, які хочуть використовувати потужні можливості Python разом із перевагами Kotlin або Java у створенні Android-застосунків [10].

Основні функції Chaquopy

– виклик Python з Kotlin/Java: Chaquopy дозволяє викликати функції Python безпосередньо з Kotlin або Java. Це робить можливим використання складних алгоритмів, написаних на Python, у мобільних застосунках, не переписуючи їх на Kotlin чи Java;

– виклик Kotlin/Java з Python: на додаток до виклику Python з Kotlin або Java, Chaquopy також надає можливість викликати код Kotlin або Java безпосередньо з Python. Це розширює можливості взаємодії між мовами,

дозволяючи розробникам максимально використовувати переваги кожної мови;

– доступ до API Android: Через Chaquopy Python-код може взаємодіяти з API Android, такими як доступ до файлової системи, баз даних SQLite, і навіть створення рідних інтерфейсів користувача;

– включення Python-бібліотек: Chaquopy дозволяє інтегрувати більшість Python-бібліотек, навіть якщо вони містять рідний код C або C++. Бібліотеки можуть бути завантажені через pip, що спрощує управління залежностями.

Переваги використання Chaquopy:

– швидка інтеграція: Розробники можуть швидко інтегрувати наукові та аналітичні можливості Python у свої Android-застосунки;

– гнучкість: Використання Chaquopy дозволяє розробникам вибирати найкращі інструменти для кожного конкретного завдання, не обмежуючись однією технологією;

– ефективність розвитку: Зменшується потреба в переписуванні існуючих Python-скриптів на Kotlin або Java, що зберігає час та ресурси.

У розробці гри-лабіринту на Android, Chaquopy може бути використаний для інтеграції алгоритмів побудови лабіринту, написаних на Python, без необхідності переписувати їх на Kotlin. Це дозволяє використовувати готові рішення та складні алгоритми, які вже реалізовані у Python, тим самим зберігаючи час і зусилля, які були б потрібні для створення їх з нуля на Kotlin.

Завдяки Chaquopy, розробники можуть забезпечити гладку та ефективну взаємодію між Python-алгоритмами та Android-застосунком, підвищуючи загальну продуктивність і якість гри.

2.6 Алгоритм Прима для побудови лабіринту

Алгоритм Прима – це класичний метод з теорії графів, який первісно був розроблений для знаходження мінімального основного дерева в зваженому зв’язному графі. У контексті побудови лабіринтів, цей алгоритм адаптовано для створення складних і випадкових лабіринтів, що ідеально підходять для ігор та інших застосунків, де потрібні унікальні, неповторні структури [11].

Алгоритм Прима для побудови лабіринту починається з сітки, де кожна клітина вважається окремою кімнатою або областю. Всі можливі стіни між кімнатами спочатку присутні. Алгоритм вибирає випадкову клітину як початкову точку, а потім розпочинає процес «вирощування» лабіринту з цієї клітини.

Процес працює наступним чином: з активної клітини, яка на даний момент є частиною лабіринту, алгоритм розглядає всі сусідні клітини, які ще не були додані до лабіринту. Він вибирає одну з цих сусідніх клітин випадково і прибирає стіну між обраною клітиною та активною клітиною. Обрана клітина тепер стає частиною лабіринту, і процес повторюється з нової клітини.

Особливість алгоритму Прима полягає в тому, що він вибирає наступну клітину для приєднання на основі випадкового вибору серед усіх сусідніх клітин, що ще не входять до лабіринту. Це забезпечує, що кожен лабіринт, створений за допомогою алгоритму Прима, є унікальним і має велику різноманітність віток і поворотів, що ідеально підходить для ігрових програм.

Однією з ключових переваг алгоритму Прима є його здатність створювати лабіринти з одним єдиним виходом, що є важливим для ігрових лабіринтів, де має бути лише один шлях до фінішу. Крім того, алгоритм дозволяє легко контролювати складність лабіринту шляхом налаштування кількості випадково вибраних стін, які потрібно видалити.

Використання алгоритму Прима в програмному забезпеченні для побудови лабіринту дозволяє розробникам використовувати потужний та гнучкий інструмент, який може бути адаптований для різноманітних вимог і сценаріїв, забезпечуючи як високу якість генерації лабіринтів, так і задоволення користувацьких потреб у ігровому процесі.

Прикладом як виглядає генерація лабіринту є рисунок 2.2, де видно вже згенеровані шляхи і ще певну частину непройденого поля, червоним визначаються потенційні клітки, які ще не пройдені і доступні для проходження.

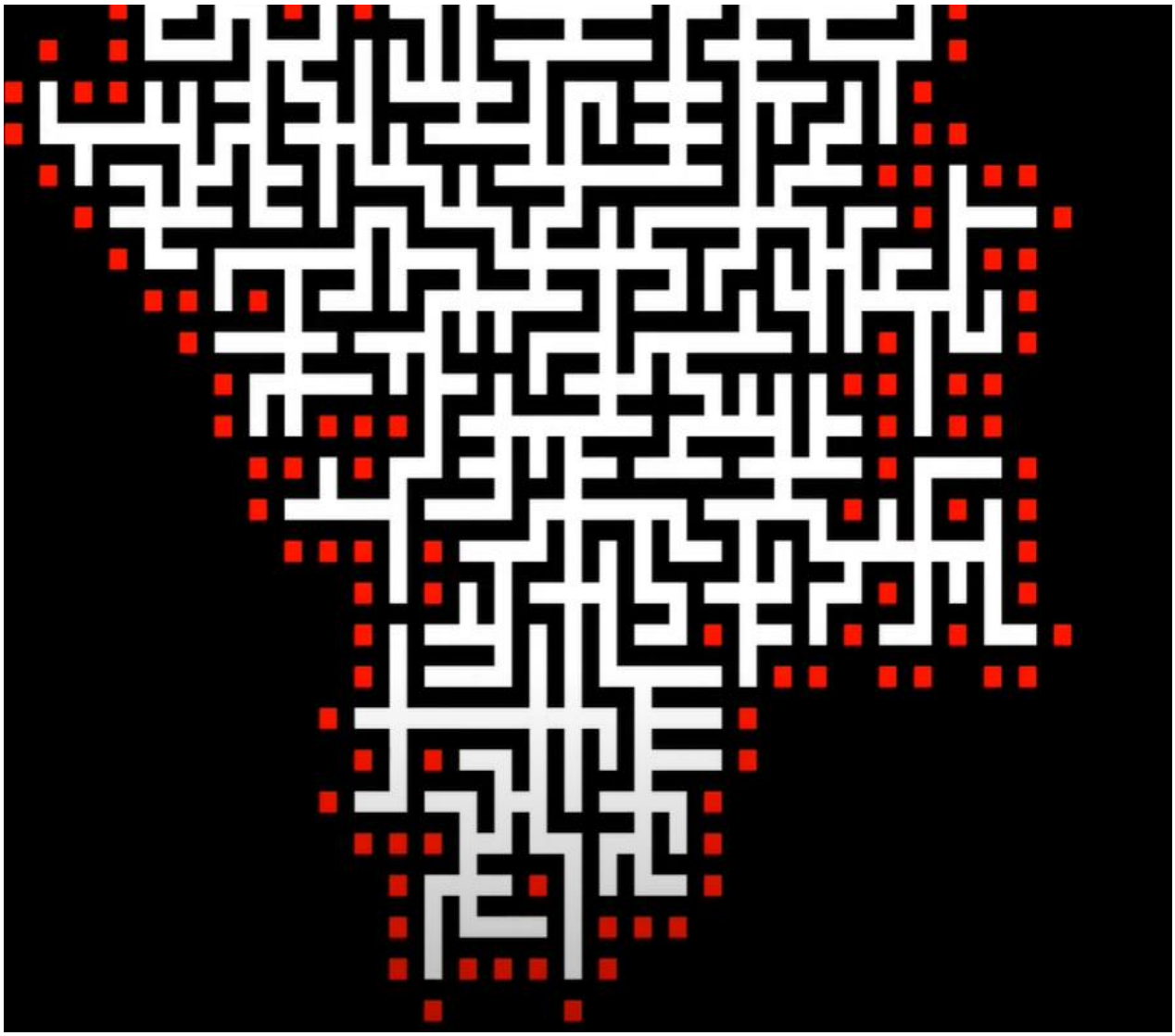

Рисунок 2.2 – Приклад побудови лабіринту за допомогою алгоритму Прима

2.7 Алгоритм Краскала для побудови лабіринту

Алгоритм Краскала є ще одним популярним методом з теорії графів, який використовується для знаходження мінімального основного дерева у зваженому графі. В контексті побудови лабіринтів, алгоритм Краскала адаптовано для створення випадкових і розгалужених лабіринтів, які мають складні структури та мінімум повторень [12].

Для створення лабіринту алгоритмом Краскала спочатку утворюється сітка, де кожна клітина розглядається як окрема вершина графа, а стіни між клітинами – як ребра, які можуть бути «знищені» або залишені. Кожному ребру присвоюється випадкова вага, що впливає на порядок, у якому ребра будуть оброблятися [13]. Прикладом є рисунок 2.3, де видно вже сформований лабіринт з проставленими вагами

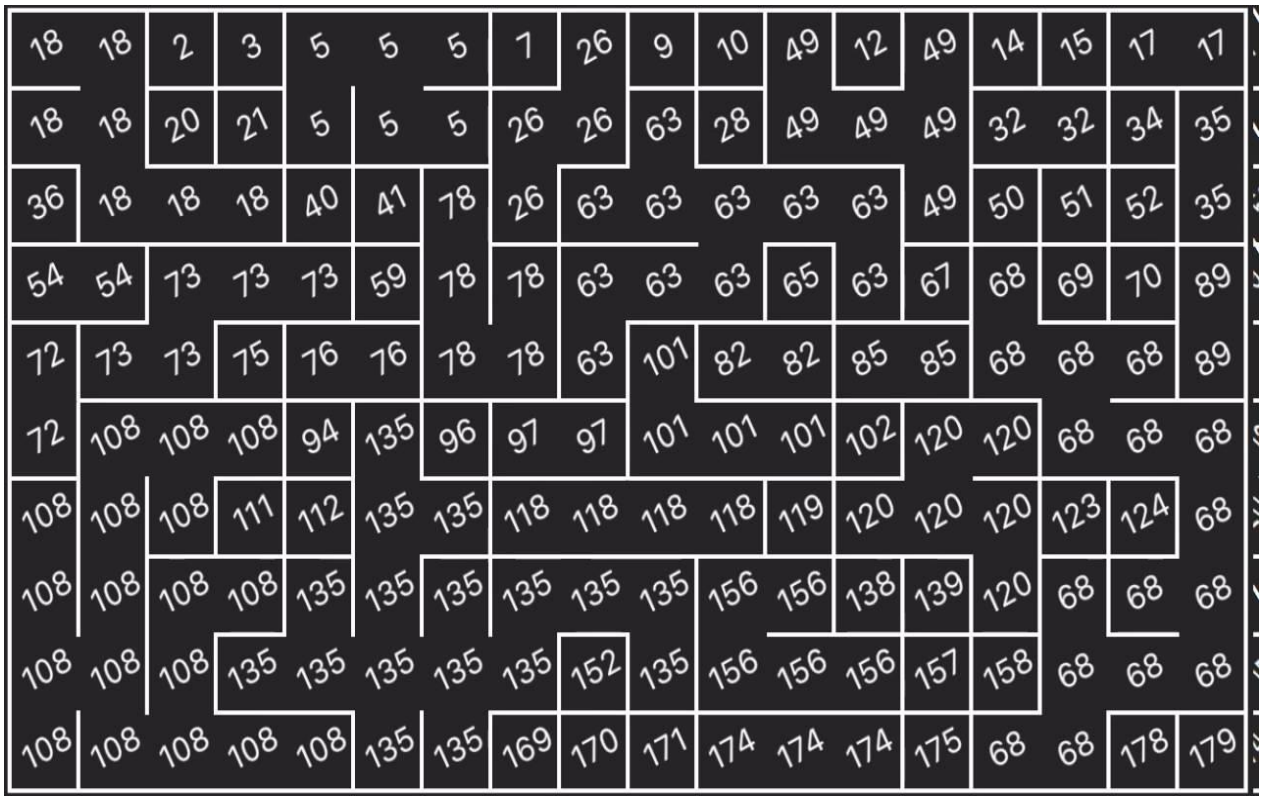

Рисунок 2.3 – Приклад згенерованого лабіринту за допомогою алгоритму Краскала, з оціненими клітинами

Процес створення виглядає ось так:

– ініціалізація: всі клітини ініціалізуються як окремі множини або «ліси», де кожна клітина представляє собою окреме дерево;

– сортування ребер: всі ребра (стіни) сортуються за збільшенням ваги;

– об’єднання множин: починаючи з ребра з найменшою вагою, перевіряється, чи з’єднують ребра вершини з різних множин. Якщо так, ребро «знищується» (стіна видаляється), а дві множини (клітини) об’єднуються в одну. Цей процес продовжується до тих пір, поки всі клітини не опиняться в одній множині;

– формування лабіринту: коли всі клітини об’єднані без ізольованих частин, процес завершується, і результатом є складний лабіринт без замкнутих циклів і з одним єдиним входом та виходом.

Переваги алгоритму Краскала для побудови лабіринтів:

– контрольована складність: завдяки випадковій вазі ребер, кожен лабіринт, створений алгоритмом Краскала, є унікальним і має різну складність, що робить кожну гру цікавою та непередбачуваною;

– велика різноманітність структур: алгоритм дозволяє створювати лабіринти з широким спектром структур, від простих і прямих до вкрай складних і заплутаних;

– мінімальна кількість зайвих коридорів: оскільки алгоритм видаляє стіни тільки для з’єднання різних множин, він мінімізує кількість непотрібних проходів, забезпечуючи чистоту дизайну.

Алгоритм Краскала є відмінним вибором для розробників ігор, які потребують генерації лабіринтів з контрольованою випадковістю та високим рівнем налаштування складності. Його можливість адаптуватися до різних потреб і сценаріїв робить його незамінним інструментом у арсеналі сучасного розробника ігор.

2.8 Порівняння алгоритмів Краскала та Прима для побудови лабіринтів

Алгоритми Краскала та Прима обидва використовуються для генерації лабіринтів і обидва можуть створювати захоплюючі та складні маршрути в іграх та інших застосунках. Вони мають свої унікальні особливості, переваги та недоліки, які роблять їх більш або менш підходящими залежно від конкретних потреб проєкту.

Якщо порівнювати швидкодію та складність, то алгоритм Краскала працює з усіма ребрами графа, які він спочатку сортує, що є часово вимогливою операцією. Складність Краскала в найгіршому випадку становить O(E log E), де E – кількість ребер у графі. Це робить Краскала відносно повільним для дуже великих графів, але дуже ефективним для розріджених графів, де кількість ребер значно менша за квадрат числа вершин.

Алгоритм Прима, у свою чергу, працює з вершинами графа та вимагає підтримки структури даних для швидкого визначення наступної вершини, що має бути додана до лабіринту. За допомогою бінарної кучі, складність алгоритму Прима може досягати O(E log V), де V – кількість вершин. Прима зазвичай швидший за Краскала у випадку густого графа, де майже кожна вершина з’єднана з кожною іншою [14].

Виходячи з цих даних дуже важко обрати потрібний алгоритм, бо у кожного є свою переваги і недоліки, саме тому вибір між Краскалом і Примом для створення лабіринту залежить від декількох факторів:

– розмір та щільність графа: для розріджених графів, де кількість ребер невелика у порівнянні з кількістю вершин, Краскал може бути ефективнішим. Натомість, Прим краще підходить для густо з’єднаних графів;

– специфіка реалізації: якщо лабіринт вимагає частих взаємодій та змін під час його побудови, Прим може забезпечити кращі можливості для оптимізації та адаптації до змін;

– ресурси системи: алгоритм Прима зазвичай використовує більше пам’яті через необхідність зберігання додаткових даних про стан кожної вершини, в той час як Краскал більше фокусується на ребрах і їх вагах.

І Краскал, і Прим мають свої переваги та можуть бути використані для створення інтригуючих лабіринтів. Вибір між ними залежить від конкретних вимог до алгоритму, включаючи типи входів, що обробляються, вимоги до продуктивності та доступні системні ресурси. У загальному випадку, Краскал підходить для ситуацій, коли потрібна більша гнучкість і рандомізація у виборі стін для видалення, тоді як Прим забезпечує більшу ефективність у щільних графах з великою кількістю вершин.

3 РЕАЛІЗАЦІЯ ПОСТАНОВКИ ЗАДАЧІ ТА ТЕСТУВАННЯ

3.1 Вибір системи розробки і обґрунтування вибору

Для розробки застосунку було використано інтегроване середовище розробки Android Studio, яке є офіційним IDE для платформи Android. Це рішення було обрано з кількох причин, кожна з яких важлива для забезпечення ефективності та якості процесу розробки.

Android Studio забезпечує тісну інтеграцію з Android Software Development Kit (SDK), що дозволяє розробникам легко доступатися до останніх API та інструментів, необхідних для розробки сучасних Android-додатків. Це включає доступ до компонентів, таких як Android Jetpack, які містять готові до використання компоненти та бібліотеки, що значно пришвидшують розробку та покращують якість додатку. Прикладом інтерфейсу середовища є рисунок 3.1, на ньому видно велику кількість можливостей, що надає студія в інтеграції з Android SDK.

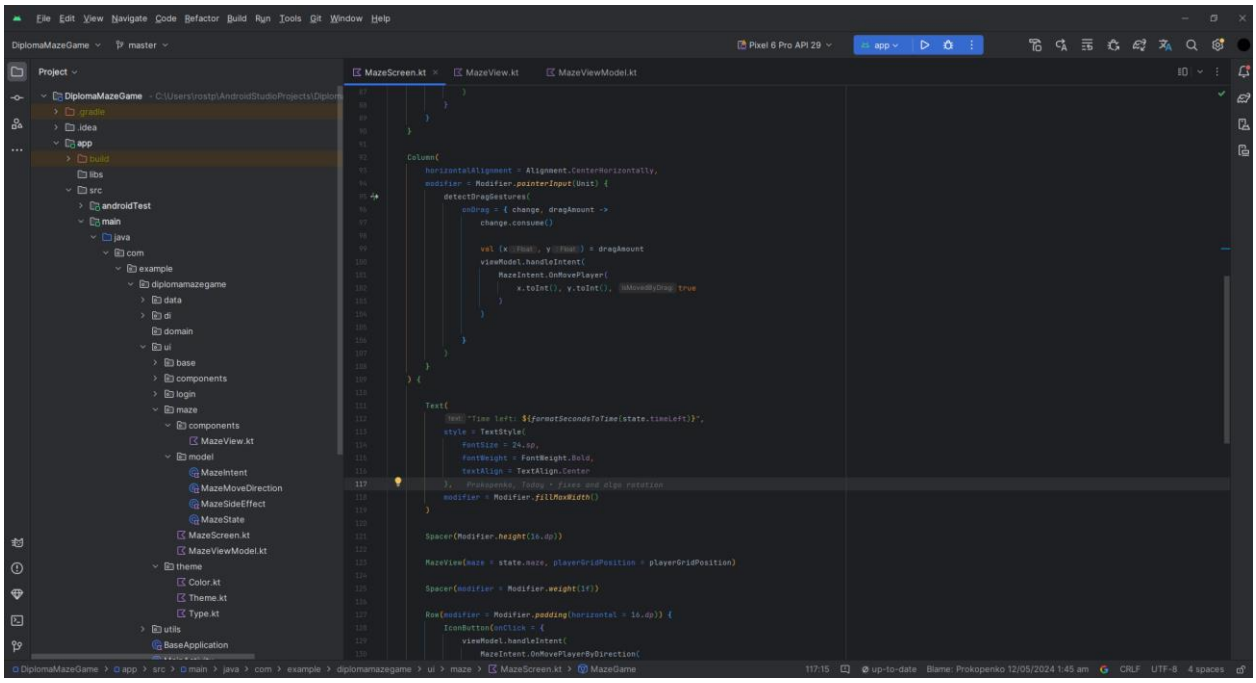

Рисунок 3.1 – Інтерфейс Android Studio

Також, середовище пропонує розширені можливості для профілювання додатків та дебагінгу, що дуже важливо при розробці вимогливих за ресурсами додатків, таких як ігри. Вбудовані інструменти, такі як Android Profiler, дозволяють розробникам моніторити використання ЦПУ, споживання пам’яті та мережеву активність у реальному часі, що сприяє оптимізації та виявленню потенційних проблем на ранніх стадіях розробки.

Android Studio надає повну підтримку для Kotlin, яка є офіційною мовою розробки Android, а також для Jetpack Compose. Це середовище дозволяє розробникам використовувати всі переваги Kotlin. Це можна побачити на рисунку 3.2, де показано інтеграцію з Jetpack Compose, а саме, моментальне відображення побудованого елементу інтерфейсу без необхідності запускати застосунок, що значно полегшує і пришвидшує розробку [15].

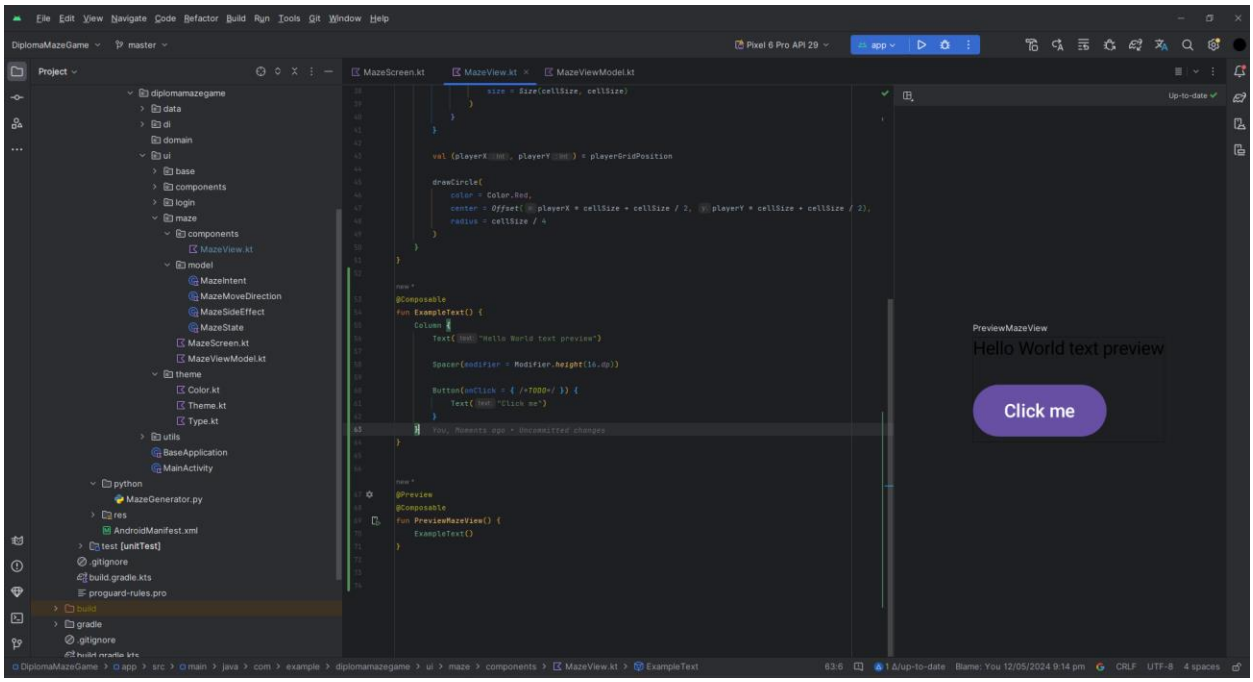

Рисунок 3.2 – Приклад інтеграції Android Studio з Jetpack Compose

Android Studio підтримується великою спільнотою розробників та компанією Google, що забезпечує регулярні оновлення інструментів та

платформи. Спільнота надає численні ресурси, плагіни, бібліотеки та кращі практики, які доступні розробникам для використання у своїх проєктах.

Для розробки інтерфейсу застосунку було обрано Figma, сучасний інструмент для дизайну інтерфейсів, який стає все популярнішим серед дизайнерів і розробників. Figma пропонує широкий спектр можливостей для створення, прототипування та колаборації, роблячи її ідеальним інструментом для команд, які прагнуть ефективно співпрацювати над дизайном продукту (рис. 3.3).

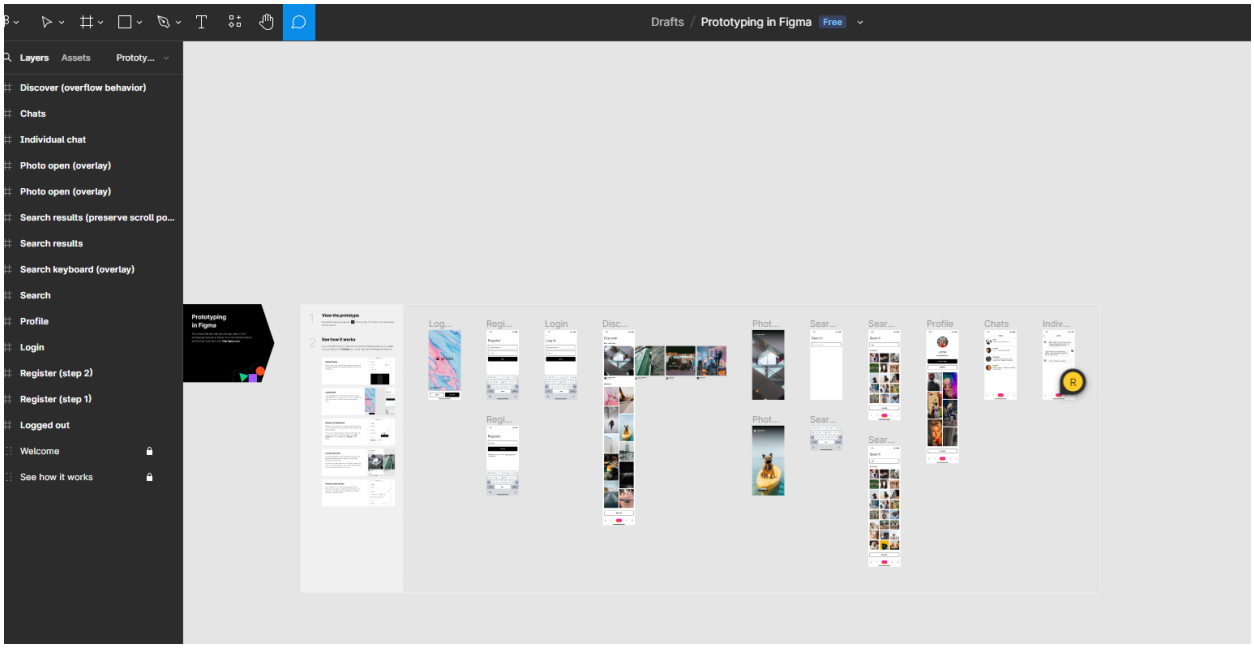

Рисунок 3.3 – Інтерфейс Figma

Figma – це базований на хмарних технологіях інструмент, який дозволяє дизайнерам створювати високоякісні дизайни інтерфейсів та прототипи без необхідності встановлення будь-якого програмного забезпечення. Це забезпечує легкий доступ до проєктів з будь-якого пристрою, що підключений до Інтернету, і спрощує процес внесення змін і обговорення дизайну в реальному часі.

Figma пропонує велику кількість плагінів та інтеграцій, що дозволяє дизайнерам підключити різні інструменти та сервіси, які вони вже використовують, щоб поліпшити свій робочий процес. Це включає

інструменти для автоматизації рутинних завдань, імпорту зовнішніх даних, а також засоби для кращого тестування і відображення прототипів.

Figma має інтуїтивно зрозумілий інтерфейс і надає багато ресурсів для навчання, включаючи підручники, відео та спільноти, де користувачі можуть обмінюватися порадами і кращими практиками. Це робить Figma доступною не тільки для досвідчених дизайнерів, але й для новачків, які тільки починають свій шлях у дизайні.

Однією з ключових переваг Figma є її здатність підтримувати співпрацю в реальному часі. Дизайнери та розробники можуть одночасно працювати над одними і тими ж файлами, бачити зміни один одного миттєво та коментувати елементи дизайну безпосередньо в самому інструменті, це продемонстровано на рисунку 3.4. Це рішення ідеально підходить для швидкої ітерації та узгодження ідей у великих командах, скорочуючи час від ідеї до реалізації [16].

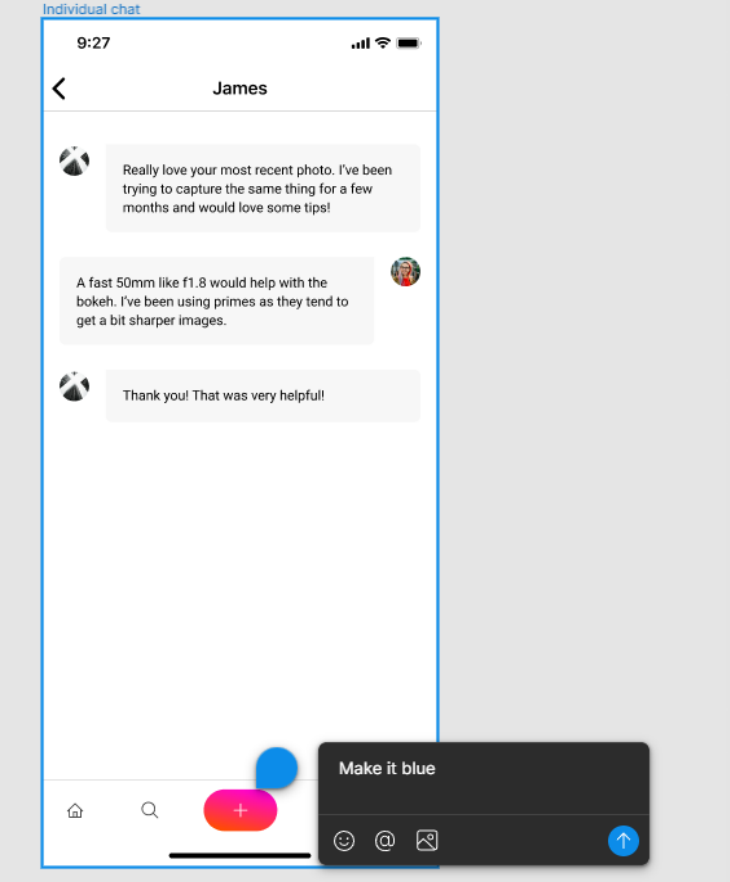

Рисунок 3.4 – Приклад коментаря до макету

Для збереження глобальних результатів гравців і формування рейтингу в застосунку було вибрано Firestore Database, частину платформи Firebase від Google, яка спеціалізується на розробці мобільних та вебзастосунків. Firestore є NoSQL базою даних, що забезпечує динамічне зберігання і синхронізацію даних в реальному часі, що є ідеальним для додатків з активною взаємодією користувачів [17].

Основною перевагою Firestore є її здатність до масштабування, що дозволяє витримувати великі навантаження і обробляти великі обсяги даних без втрати продуктивності. Це робить Firestore відмінним вибором для ігор, де рейтинг гравців постійно оновлюється та де необхідне миттєве відображення оновлень у всіх користувачів застосунку (рис. 3.5). Додатково, ця платформа пропонує інтуїтивно зрозумілі інструменти для управління даними, що дозволяють легко реалізовувати складні запити та інтеграцію з іншими сервісами Firebase для розширення функціональності.

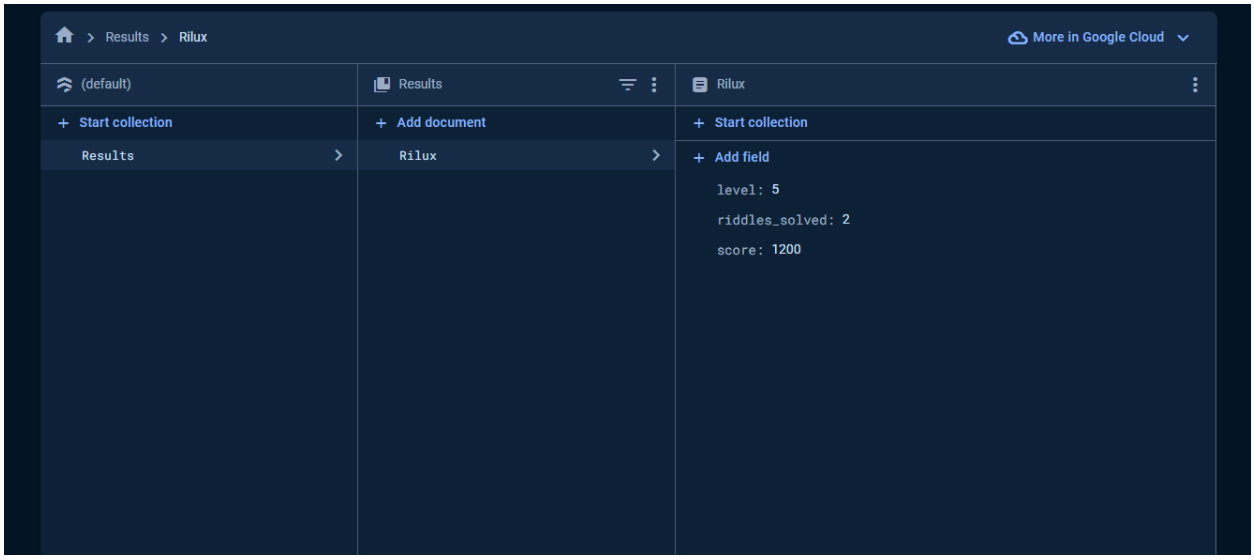

Рисунок 3.5 – База даних результатів гравця

Firestore також підтримує гнучкість в управлінні даними, дозволяючи розробникам ефективно організувати дані у вигляді документів і колекцій. Кожен документ може містити різні типи даних, що спрощує роботу з складними даними та їх структурами. Такий підхід дозволяє швидко

адаптуватися до змін у вимогах до додатку без необхідності перебудови всієї бази даних.

Обрання Firestore як основного засобу для збереження даних у застосунку забезпечує не тільки технічну ефективність, але й значно покращує досвід користувачів, надаючи їм надійний та безперервний доступ до своїх даних і результатів інших гравців в реальному часі.

3.2 Програмна реалізація застосунку гри-лабіринту

Для написання коду гри було обрано мову програмування Kotlin. Цей вибір був зумовлений тим, що Kotlin ідеально поєднується з інструментами Android SDK та підтримує використання останніх бібліотек для розробки програмного забезпечення на базі операційної системи Android. На відміну від Java, яка поступово визнається застарілою в контексті сучасних технологічних трендів, Kotlin внесений як офіційна мова розробки для Android за її сучасні можливості, включно з покращеною обробкою нульових значень (null safety) і підтримкою функціонального програмування. Kotlin також пропонує більш чистий і стислий синтаксис порівняно з Java, що робить код легшим для читання та підтримки.

Також варто зазначити що для написання усього ui було використано Jetpack Compose. Ця бібліотека дозволяє швидко і легко створювати модульні макети, які можна перевикористовувати і масштабувати без особливих перепон.

3.2.1 Налаштування проєкту

З самого початку варто описати головні налаштування проєкту і як було підключено головні бібліотеки. Головне налаштування проєкту

відбувалось у Gradle файлах. Gradle – це потужна система автоматизації збірки, яка широко використовується в розробці Android-застосунків. Вона дозволяє розробникам ефективно керувати залежностями, конфігураціями та різними середовищами збірки (наприклад, розробка, тестування, випуск). Gradle використовує гнучкий DSL (Domain-Specific Language) на базі Groovy або Kotlin для опису та налаштування процесів збірки, що дозволяє налаштовувати збірку під конкретні потреби проєкту [18]. В контексті Android, Gradle спрощує процес компіляції коду, ресурсів, і пакування застосунків у APK або AAB формати. У проєкті було використано багато комплексних бібліотек та плагінів, більшість з них інтегрувалась у проєкт як наведено на прикладі нижче (рис. 3.6, 3.7).

```
    implementation("com.jakewharton.timber:timber:5.0.1")

    implementation("io.github.raamcosta.compose-destinations:core:1.10.2")
    ksp("io.github.raamcosta.compose-destinations:ksp:1.10.2")

    implementation("org.orbit-mvi:orbit-core:7.1.0")   Prokopenko, 11/05/2024 9:25 pm • added timer, refactore
    implementation("org.orbit-mvi:orbit-viewmodel:7.1.0")
    implementation("org.orbit-mvi:orbit-compose:7.1.0")

    val room_version = "2.6.1"

    implementation("androidx.room:room-runtime:$room_version")
    annotationProcessor("androidx.room:room-compiler:$room_version")
    ksp("androidx.room:room-compiler:$room_version")
    implementation("androidx.room:room-ktx:$room_version")
    implementation("androidx.room:room-rxjava2:$room_version")
    implementation("androidx.room:room-rxjava3:$room_version")
    implementation("androidx.room:room-guava:$room_version")
    testImplementation("androidx.room:room-testing:$room_version")
    implementation("androidx.room:room-paging:$room_version")

}
```

Рисунок 3.6 – Основні бібліотеки, що були використані в проєкті

```
dependencies {

    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.7.0")
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:2.7.0")
    implementation("androidx.activity:activity-compose:1.9.0")

    implementation(platform("androidx.compose:compose-bom:2023.03.00"))
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-graphics")
    implementation("androidx.compose.ui:ui-tooling-preview")
    implementation("androidx.compose.material3:material3")
    implementation("androidx.compose.ui:ui-graphics-android:1.6.7")
    androidTestImplementation(platform("androidx.compose:compose-bom:2023.03.00"))
    debugImplementation("androidx.compose.ui:ui-tooling")
    debugImplementation("androidx.compose.ui:ui-test-manifest")
    implementation("androidx.compose.ui:ui-text-google-fonts:1.6.7")

    implementation("com.google.firebase:firebase-firestore:25.0.0")
    implementation("com.google.firebase:firebase-auth:23.0.0")

    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")

    implementation("com.google.dagger:hilt-android:2.51")
    ksp("com.google.dagger:dagger-compiler:2.51") // Dagger compiler
    ksp("com.google.dagger:hilt-compiler:2.51")
    implementation("androidx.hilt:hilt-navigation-compose:1.2.0")
    implementation("androidx.hilt:hilt-navigation:1.2.0")

    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.8.0")
    implementation("com.squareup.retrofit2:retrofit:2.9.0")
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.3")
    implementation("com.jakewharton.retrofit:retrofit2-kotlinx-serialization-converter:1.0.0")
    implementation("com.squareup.okhttp3:okhttp:4.12.0")
    implementation("com.squareup.okhttp3:logging-interceptor:4.12.0")
    implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.6.0-RC.2")

```

Рисунок 3.7 – Основні бібліотеки, що були використані в проєкті

Але є і бібліотеки з особливим способом інтеграції, серед них одна з найголовніших для проєкту – Chaquopy. Ця бібліотека підключається окремим методом, перш за все вона підключається в плагінах (рис. 3.8), та окремо підключається метод для визначення які системні архітектури підтримуються (рис. 3.9), це необхідно в першу чергу через те, що бібліотека працює за рахунок трансляції коду Python в C++, аби той міг бути

скомпільований за допомогою Android NDK, а він підтримує обмежену кількість архітектур.

```
plugins {
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
    id("com.google.devtools.ksp")
    id("com.google.dagger.hilt.android")
    id("org.jetbrains.kotlin.plugin.serialization")
    id("com.chaquo.python")   Prokopenko, 13/03/2024 10:16 pm
    id("kotlin-parcelize")
    id("com.google.gms.google-services")
}
```

Рисунок 3.8 – Підключення основних плагінів в проєкті, серед яких Chaquopy

```
ndk {
    abiFilters.addAll(listOf("arm64-v8a", "x86_64"))
}
```

Рисунок 3.9 – Визначення основних архітектур, які підтримуються проєктом

3.2.2 Процес аутентифікації в застосунку

При вході в застосунок система перевіряє наявність автентифікованого акаунту і веде на головну сторінку, якщо такий присутній (лістинг 3.1).

Лістинг 3.1 Реалізація перевірки на наявність автентифікованого облікового запису:

```
private fun loadDataAboutUser() {
    val user = firebaseAuth.currentUser
    if (user == null) {
        postSideEffect(MazeSplashSideEffect.ShowLogin)
```

```
        } else {
            postSideEffect(MazeSplashSideEffect.ShowMenuScreen)
        }
    }
```

Якщо при перевірці буде знайдено дійсний акаунт, то гравця автоматично переведе в меню гри, якщо ж обліковий запис не буде знайдено, то користувача буде перенесено на екран логіну для проходження подальшого процесу аутентифікації.

На сторінці логіну користувач або може пройти процес логіну, ввівши свою пошту і пароль (рис. 3.10), на які було зареєстровано обліковий запис, або пройти процес реєстрації. При введені даних для входу буде відбуватись перевірка на їх правильність і можливість відправки даних на вхід відкривається тільки якщо пошта і пароль задовольняють умовам, це значно економить трафік для сервера, так як ліквідує запити з завчасно відомим негативним результатом.

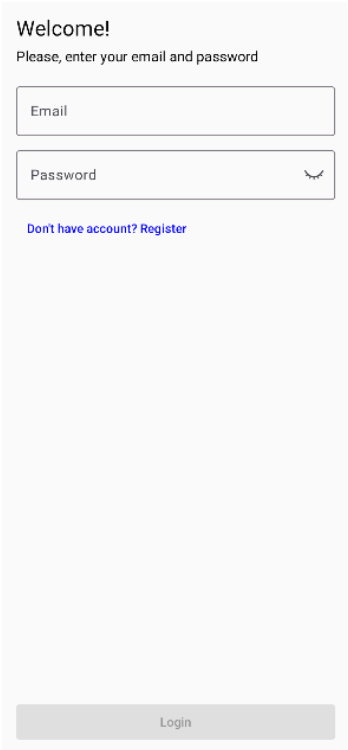

Рисунок 3.10 – Екран логіну у застосунок

Лістинг 3.2 Реалізація перевірки на відповідність формату пошти та відповідність пароля правилам:

```
val isEmailValid = remember(email) { PatternsCompat.EMAIL_ADDRESS.matcher(email).matches() }
val isPasswordValid = remember(password) { password.length > 6 && password.any { it.isDigit() } && password.any { it.isLetter()
}
}
```

За умови що обидва поля мають валідні значення, користувач має змогу натиснути на кнопку логіну, відбувається запит на Firebase Auth [19] і за успішного результату користувача переводить на екран меню гри, у випадку невдалого запиту користувача повідомляють про це.

Якщо користувач не має облікового запису, чи йому не вдалось увійти в наявний, то він може натиснути на кнопку «Don’t have account? Register», ця текстова кнопка переведе користувача на екран реєстрації.

На екрані реєстрації (рис. 3.12) гравцю необхідно буде вказати пошту, ім’я, пароль. Після цього натиснути на кнопку реєстрації, ця кнопка буде доступна для натискання лише за умови, що всі необхідні поля заповнені і підходять під вимоги і формат.

Лістинг 3.3 Реалізація перевірки на відповідність формату пошти та відповідність пароля правилам при реєстрації:

```
val isEmailValid = email.matches("^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\\.[a-zA-Z0-9-.]+$".toRegex())
val isPasswordValid =
password.length >= 6 && password.any { it.isDigit() } && password.any { it.isLetter() }
val isFormValid =
```

email.isNotBlank() && nickname.isNotBlank() && password.isNotBlank() && isEmailValid && isPasswordValid

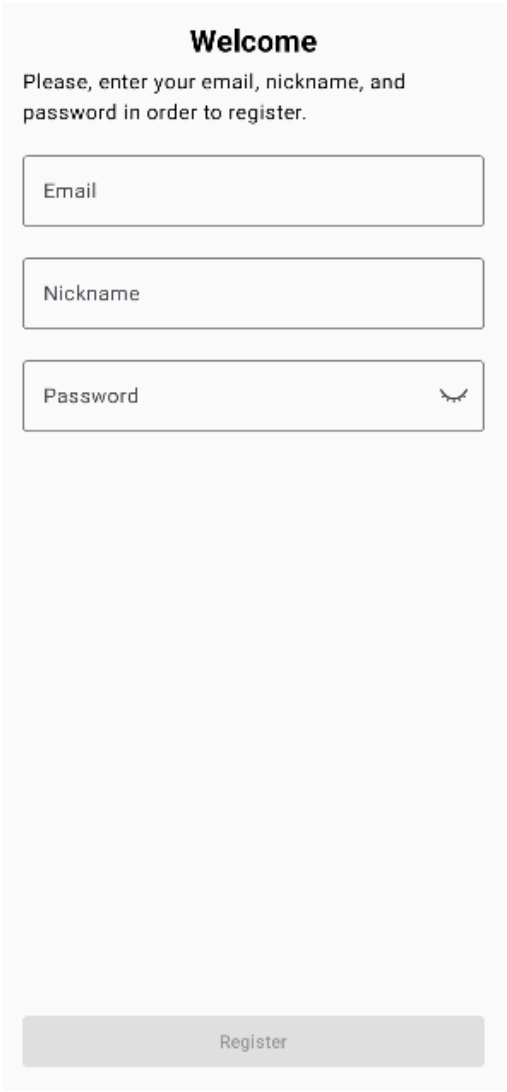

Рисунок 3.11 – Екран реєстрації в застосунку

Після натискання на реєстрацію дані відправляються в Firebase Auth, де створюється акаунт по пошті з представленим паролем, а також створюється запис у Firestore Database, де записується ім’я гравця і його пошта.

Лістинг 3.4 Реалізація реєстрація користувача:

```
private fun registerUser(
    email: String,
    nickname: String,
```

```
    password: String,
) {
    viewModelScope.launch {
        try {
            // Create user with email and password
            auth.createUserWithEmailAndPassword(email, password).await()
            // Add extra user data to Firestore
            val user = hashMapOf(
                «nickname» to nickname,
                «email» to email
            )
            db.collection(«users»).document(email).set(user).await()
            postSideEffect(RegistrationSideEffect.OnNavigateToMenu)
        } catch (e: Exception) {
            e.printStackTrace()
            postSideEffect(RegistrationSideEffect.OnShowError)
        }
    }
}
```

За умови успішної реєстрації користувача навігує на головний екран, якщо виникла помилка, то гравець бачить повідомлення про помилку (рис. 3.12).

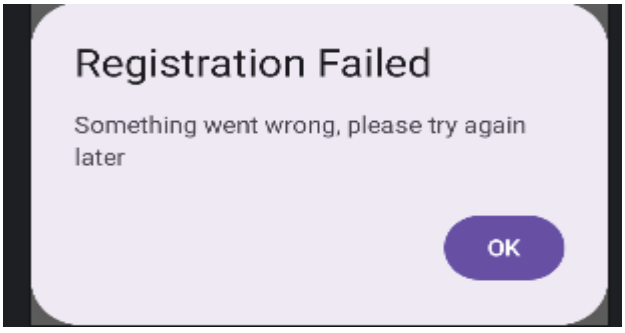

Рисунок 3.12 – Повідомлення про помилку під час реєстрації

3.2.3 Головний екран застосунку і супутні до нього екрани

Головний екран виступає основним маршрутизатором для користувача, який веде на усі допоміжні екрани, та на екран гри. Саме на головний екран потрапляє користувач після успішного проходження процесу аутентифікації.

На головному екрані гравець бачить кнопки, які ведуть на екрани гри, налаштувань та таблиці лідерів (рис. 3.13)

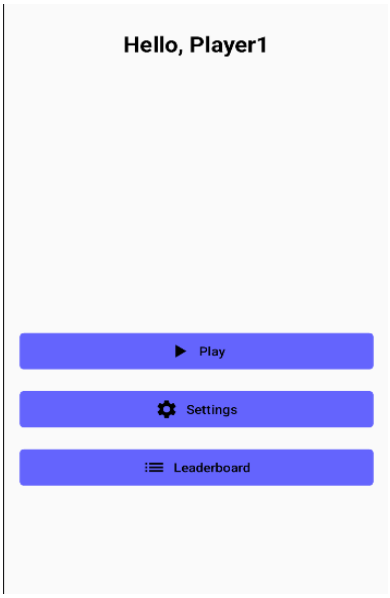

Рисунок 3.13 – Екран головного меню

При переході на екран налаштувань користувач потрапляє на екран, де він буде бачити поле з його ім’ям, яке можна змінити та кнопку, яка надає можливість вийти з поточного облікового запису (рис. 3.14).

Рисунок 3.14 – Екран налаштувань

Логіка зміни імені доволі проста, виконується запит на оновлення документу з ключем по пошті гравця.

Лістинг 3.5 Реалізація зміни імені користувача:

```
private fun updateUserName(newName: String) {
    val userRef = db.collection(«users»).document(currentUser!!.email!!)
    viewModelScope.launch {
        try {
            userRef.update(«nickname», newName).await()
            postSideEffect(SettingsSideEffect.ShowSuccessDialog)
        } catch (e: Exception) {
            postSideEffect(SettingsSideEffect.ShowFailureDialog)
        }
    }
}
```

Вихід з акаунту відбувається за допомогою інтегрованих сервісів Firebase Auth, єдине що залишається застосунку – це викликати необхідний метод і провести навігацію на екран логіну.

Лістинг 3.6 Реалізація виходу з облікового запису:

```
private fun logout() {
    auth.signOut()
    postSideEffect(SettingsSideEffect.NavigateToLoginScreen)
}
```

З головного екрану також можна перейти на екран таблиці лідерів «Leaderboard», екран представляє з себе список гравців з відображенням: загальної кількості зароблених очок, рівня, кількості вирішених загадок (рис. 3.15). Сортування списку відбувається по найбільшій кількості очок.

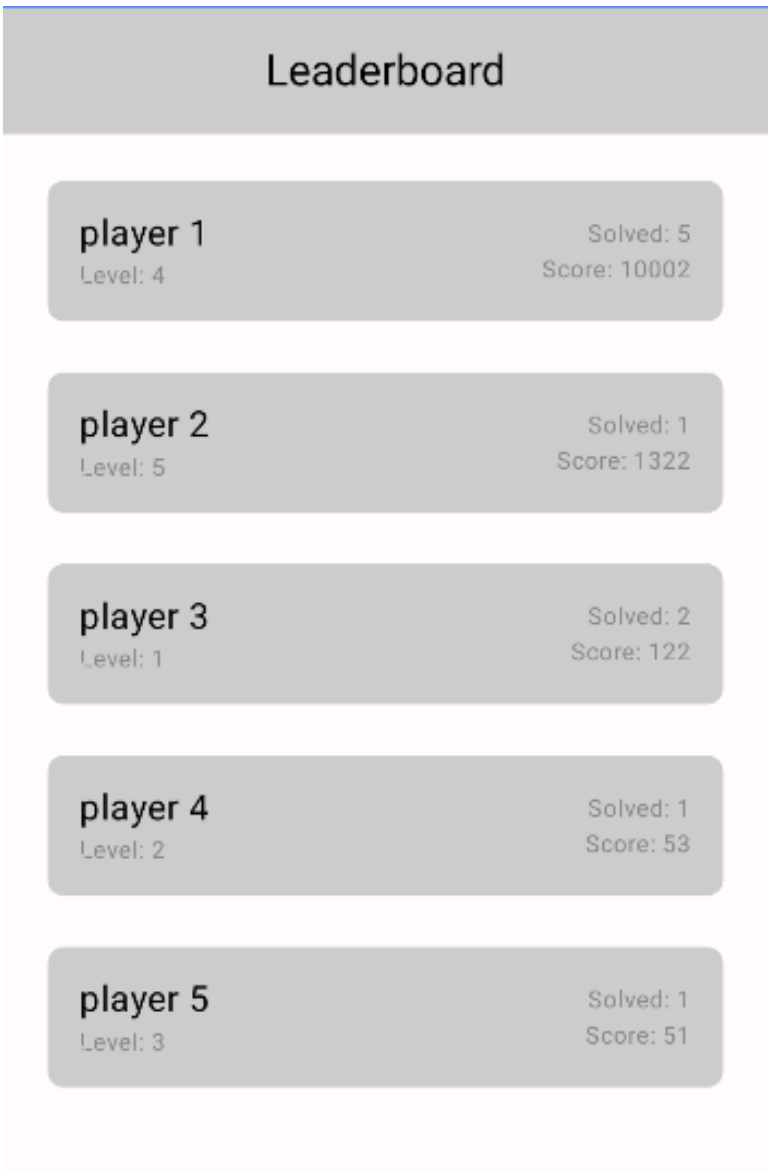

Рисунок 3.15 – Екран таблиці лідерів

Дані для таблиці завантажуються з Firestore Database, куди вони були додані під час гри гравця, що буде розглянуто далі. Після завантаження дані сортуються за кількістю очок і відправляються на екран для відображення списку.

Лістинг 3.7 Реалізація завантаження даних для таблиці лідерів і сортування цих даних:

```
private fun loadData() {

    updateState {
```

```
            it.copy(isLoading = true)
        }

        viewModelScope.launch {
            firestore.collection(«Results»)
                .get()
                .addOnSuccessListener { documents ->
                    val resultsList = documents.map { doc ->
                        doc.toObject<PlayerRank>()
                            .copy(name = doc.id) // Assuming doc.id is the player's name
                    }

                    updateState {
                        it.copy(score = resultsList.sortedByDescending { it.score },
isLoading = false)
                    }
                }
        }
    }
```

3.2.4 Екран лабіринту і загальна логіка гри

Загальна ідея лабіринту в тому, аби підлаштовуватися під проходження гравця, тобто, підлаштовуватись під його швидкість проходження і поточні дані [20]. В залежності від швидкості проходження лабіринт вирішує чи перевести користувача на новий рівень, або змінити алгоритм побудови лабіринту на більш комплексний, або просто додати гравцю загадок у лабіринт, аби він проходив його трохи повільніше, затримуючись на загадку.

Сам лабіринт складається з: стін, проходів, старту, фінішу, точки гравця та загадок (рис. 3.16). Лабіринт у коді представлений як матриця чисел, де кожне число відповідає за певний елемент на його місці [21]. На

рисунку видно чорним стіни, зеленим видно старт, червоним фініш та синім задачу. Гравець завжди починає в лівому нижньому куті і повинен дійти до фінішу, тобто, червоного квадрату.

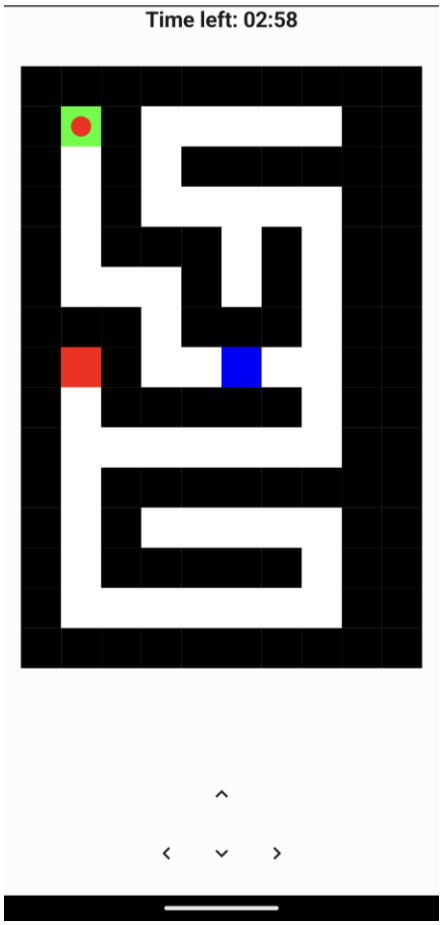

Рисунок 3.16 – Екран застосунку під час гри

Гравець може проходити крізь усі елементи окрім стін, у випадку проходження крізь синій квадрат гравцю буде показано опціональне математичне рівняння (рис. 3.17), яке він може вирішити і отримати додаткові очки за це, а система прийме це до уваги на наступному рівні [22]. Також у майбутньому планується додавання інших типів загадок, головним чином вони будуть орієнтуватися на когнітивні і графічні Це буде якесь невелике завдання, яке треба обдумати, або це буде якась невеличка задачка де треба намалювати щось схоже на поставлену задачу, або обрати правильну фігуру, логіка доволі схожа на проходження «капчі» при перевірці що користувач не робот.

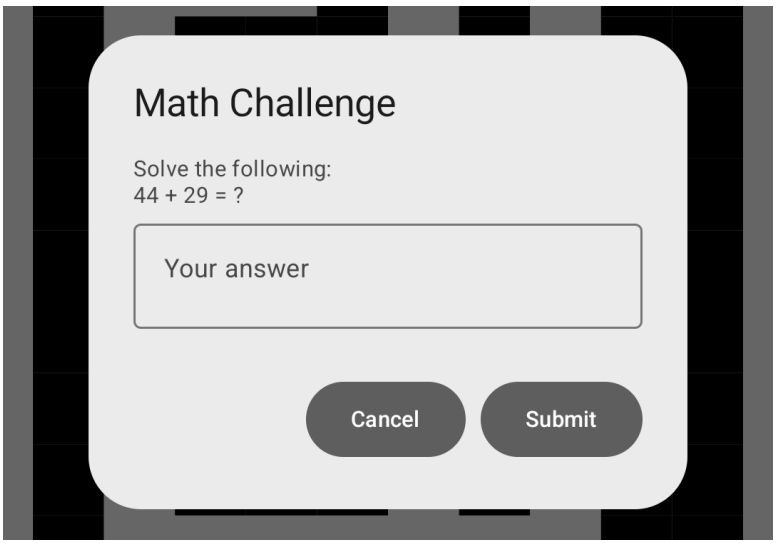

Рисунок 3.17 – Математична задачка під час проходження лабіринту

Побудова лабіринту відбувається за допомогою двох алгоритмів: Краскала і Прима [23]. Алгоритм Прима використовується першим, так як лабіринти побудовані ним вважаються простіше, тоді як алгоритм Краскала будує алгоритми складніше. Самі алгоритми написані на мові Python в окремому ру файлі, а використання цих методів відбувається за допомогою бібліотеки Chaquopy. З цікавого варто виділити що перед поверненням матриці згенерованого лабіринту проводиться ще одна перевірка на можливість проходження лабіринту і на шляху гравця проставляються математичні загадки.

Логіка побудови така, що в залежності від рівня лабіринту обирається алгоритм для побудови, а після цього викликається необхідний метод (рис. 3.18) для побудови лабіринту, зараз їх два, але в майбутньому може бути їх розширення. Також є певна особливість щодо роботи бібліотеки, усі методи Python повертають лише примітиви, тому навіть якщо це матриця, то вона повернеться в Kotlin код як строка, і надалі необхідно конвертувати її в матрицю, що зробити не так легко, так як структура строки матриці в мові Python відрізняється від такої в Kotlin, тому необхідно писати власний перетворювач, що і було реалізовано у методі «convertPythonMatrixToKotlin».

```kotlin
private fun generatePythonMaze() {
    updateState { it.copy(isLoading = true) }

    val py : Python = Python.getInstance()
    val module : PyObject = py.getModule( name: "MazeGenerator")
    val numsLength : PyObject? = if (currentAlgorithm == 0) {
        module["generate_maze"]
    } else {
        module["generate_kraskal_maze"]
    }

    val mazeWidth : Int =
        (currentLevel + 1) * 10 / if (currentAlgorithm == 1) 2 else 1


    val mazeHeight : Int =
        (currentLevel + 1) * 15 / if (currentAlgorithm == 1) 2 else 1

    val res : String = numsLength?.call(mazeWidth, mazeHeight).toString()
        .trimIndent()
    val result : Array<Array<Int>> = convertPythonMatrixToKotlin(res)
    val pos : List<Pair<Int, Int>> = findIndexesOfNumber(result, number: 2)
    pos.firstOrNull()?.let {
        playerGridPosition.value = it
    }

    updateState { it.copy(maze = result.convertArrayToArrayOfList(), isLoading = false) }
}


private fun convertPythonMatrixToKotlin(input: String): Array<Array<Int>> {
    val rows : List<String> = input.drop( n: 2).dropLast( n: 2).split( ...delimiters: "], [")

    return Array(rows.size) { rowIndex ->
        rows[rowIndex].split( ...delimiters: ", ").map { it.toInt() }.toTypedArray()
    }
}
Prokopenko, 18/04/2024 9:09 pm • fixed py matrix converter
```

Рисунок 3.18 – Реалізація методу виклику алгоритмів побудови лабіринту

Пересуватися лабіринтом користувач може як за допомогою стрілок, так і за допомогою «свайпів» по екрану, ця логіка фактично веде в одне місце, де цифра гравця переміщується по матриці в потрібному напрямку, а вже на екрані відображається її нове положення. Єдина різниця між методами пересування в тому, що при пересуванні «свайпом» існує невеликий алгоритм обмеження швидкості руху, інакше гравець би не встигав за рухами

(рис. 3.19). В коді нижче можна побачити що рух відбувається моментально якщо це був рух по кнопці чи якщо дозволяється рух жестом, рух жестом дозволяється кожні 300 мілісекунд, це якраз і убезпечує від хаотичного пересування персонажа [24].

```
private fun movePlayer(
    deltaX: Int,
    deltaY: Int,
    isMovedByGesture: Boolean = false,
) = intent {
    val maze : List<List<Int>> = state.maze

    viewModelScope.launch {
        if (isAbleToMove || !isMovedByGesture) {
            isAbleToMove = false
            val (currentX : Int , currentY : Int ) = playerGridPosition.value
            val nextX : Int  = (currentX + deltaX).coerceIn(0, maze[0].size - 1)
            val nextY : Int  = (currentY + deltaY).coerceIn(0, maze.size - 1)

            // Move player if the next cell is not a wall
            if (maze[nextY][nextX] != 1) {
                playerGridPosition.value = Pair(nextX, nextY)
            }

            if (maze[nextY][nextX] == 3) {
                postSideEffect(MazeSideEffect.ShowVictoryDialog( pointsReceived: (currentLevel + 1) * 100))
            }

            if (maze[nextY][nextX] == 4) {
                postSideEffect(MazeSideEffect.ShowMathDialog)
            }

            delay(0.3.seconds)
            isAbleToMove = true
        }

    }

}
```

Рисунок 3.19 – Реалізація логіки руху гравця

Головна логіка адаптації гри полягає в тому, що залежно від того, наскільки швидко гравець проходить лабіринт і чи розв’язав він загадку, застосунок обирає, чи переводити гравця на новий рівень [25]. Якщо часу залишилось більше 30% від початкового, то переводити, або якщо користувач вирішив задачку і прийшов до фінішу вчасно. Якщо ж користувач прийшов невчасно і не вирішив задачу, то він «понижується» в рівні, що впливає на його позицію в таблиці лідерів і змушує починати знову. Також

гра може визначити, чи підвищувати складність алгоритму [26]. Принцип доволі схожий: якщо користувач прийшов вчасно, але не вирішив загадку, то гра може не підняти рівень, а просто підняти складність алгоритму. Подібний підхід двох незалежних змінних дозволяє тримати гравця в тонусі і мотивувати його на досягнення найкращих результатів.

Кожний новий рівень для гравця означає не тільки підвищення кількості очок за успішне проходження, а й збільшення розміру рівня, що створює новий виклик. За кожне успішне проходження лабіринту користувачу нараховується певна кількість бонусів (рис. 3.20), яка розраховується залежно від поточного алгоритму та рівня: чим вищий рівень і чим складніший алгоритм, тим більше очок отримає гравець.

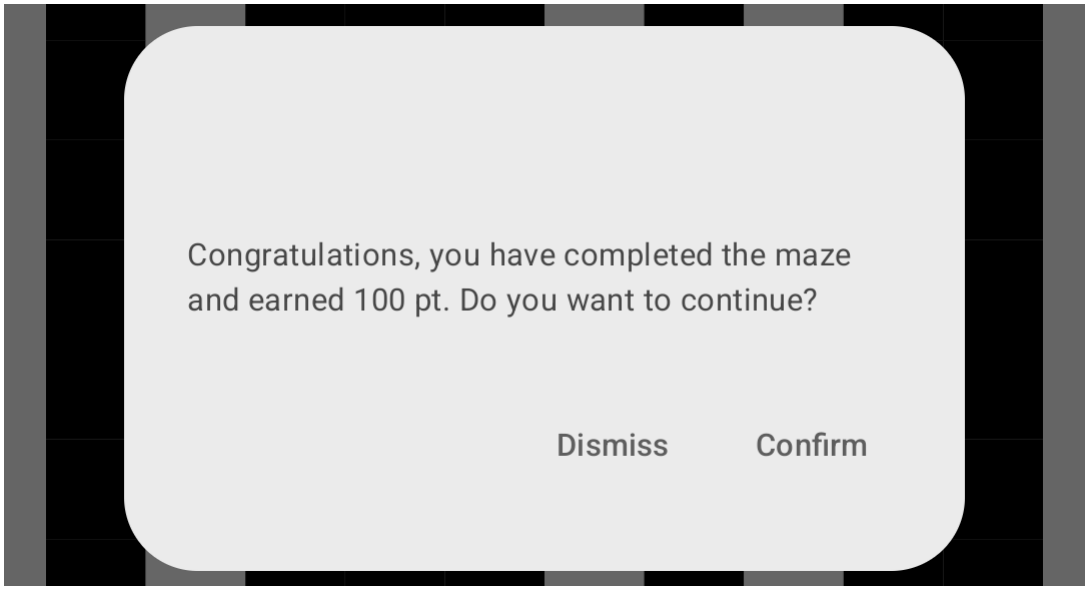

Рисунок 3.20 – Діалогове вікно по кінцю рівня

Всього для гравця доступно 5 рівнів, та 2 алгоритми, тобто загалом 10 комбінацій, які можна розіграти, якщо додати до цього задачки, кількість яких можна також збільшувати, то варіантів комбінацій стає в рази більше і все буде залежати лише від кількості таких задачок [27].

Варто також додати, що в майбутньому планується додавання нових типів задачок: на логіку, історію, програмування. Таким чином, є простір для створення своєрідного лабіринту-вікторини.

3.3 Тестування застосунку

Тестування є ключовим етапом у процесі розробки будь-якого програмного забезпечення, особливо коли мова йде про ігрові застосунки, як гра-лабіринт. Цей процес забезпечує не тільки виявлення та виправлення помилок та багів перед випуском продукту, але й допомагає переконатися, що гра відповідає всім заданим вимогам та є придатною для користувача. Тестування допомагає забезпечити високу якість ігрового досвіду, оптимізацію продуктивності та стабільність роботи застосунку на різних пристроях і платформах [28].

У контексті гри-лабіринту, тестування охоплює кілька ключових аспектів: перевірку логіки побудови лабіринту, інтерактивності елементів ігрового процесу, візуального представлення, користувацького інтерфейсу та загальної інтеграції всіх компонентів застосунку. Це дозволяє виявити потенційні проблеми, які могли б не бути очевидні на етапі розробки [29].

Тестування застосунку гри-лабіринту включає різноманітні методики та підходи, такі як модульне тестування, інтеграційне тестування, системне тестування, а також тестування користувацького досвіду. Кожен з цих підходів має свої цілі та методи, які разом формують повний цикл перевірки програми на предмет відповідності очікуванням та вимогам.

Unit тестування є фундаментальною частиною процесу забезпечення якості програмного забезпечення, зокрема у складних проєктах, таких як ігрові застосунки. Воно полягає у написанні тестів для окремих компонентів або «юнітів» коду, які перевіряють їхню функціональність в ізольованому середовищі. Це дозволяє розробникам швидко ідентифікувати та виправляти баги на ранніх етапах розробки, підвищуючи загальну надійність і стабільність застосунку.

Особливостями Unit тестування у грі-лабіринті є тестування алгоритмів генерації лабіринтів, інтерактивних елементів та геймплейної логіки. Оскільки коректна побудова лабіринту є критично важливою для ігрового

процесу, Unit тести можуть перевіряти правильність алгоритмів Прима та Краскала, зокрема, що алгоритми створюють лабіринт без недосяжних областей або ізольованих секцій.

Важливо забезпечити, що всі інтерактивні елементи гри, такі як двері, ключі або перемикачі, працюють відповідно до очікувань, і Unit тести можуть автоматично перевіряти реакцію цих елементів на дії гравця. Також до особливостей Unit тестування входить перевірка геймплейної логіки, що включає перевірку правил гри, таких як оновлення рахунку гравця, умови перемоги чи поразки. Ці тести допомагають забезпечити, що геймплей відповідає задокументованим специфікаціям.

Якщо говорити про підходи до написання Unit тестів, то вони розподіляються на такі основні частини:

– ізольованість: кожен тест перевіряє тільки один аспект функціональності та не залежить від інших тестів або зовнішніх систем;

– повторюваність: тести можна запускати багаторазово в однакових умовах без зміни результатів;

– автоматизованість: тести автоматично перевіряють умови та повертають результати без втручання людини;

– швидкість: тести виконуються швидко, щоб не затримувати процес розробки.

У контексті Android розробки, популярними інструментами для Unit тестування є JUnit для написання тестів та Mockito для створення мок-об’єктів. Android Studio забезпечує вбудовану підтримку для цих інструментів, що дозволяє розробникам легко інтегрувати тестування в їхній робочий процес [30].

Завдяки цим підходам і інструментам, Unit тестування є невід’ємною частиною процесу розробки гри-лабіринту, забезпечуючи високу якість кінцевого продукту та позитивний досвід для користувачів.

Прикладом такого тестування може послужити Unit тест написаний для класу калькулятор, який виконує роль підрахунку відповіді гравця під час

гри. На рисунку 3.21 показано самі тести, які були написані, а на рисунку 3.22 результат їх виконання.

```
class CalculatorTests {
    private val calculator = Calculator()

    @Test
    fun testAddition() {
        assertEquals( expected: 4, calculator.add( a: 2,  b: 2))
        assertEquals( expected: 0, calculator.add( a: -1,  b: 1))
        assertEquals( expected: -3, calculator.add( a: -1,  b: -2))
    }

    @Test
    fun testSubtraction() {
        assertEquals( expected: 0, calculator.subtract( a: 2,  b: 2))
        assertEquals( expected: -2, calculator.subtract( a: -1,  b: 1))
        assertEquals( expected: 1, calculator.subtract( a: -1,  b: -2))
    }
}
You, A minute ago • Uncommitted changes
```

Рисунок 3.21 – Unit тест для класу «Calculator»

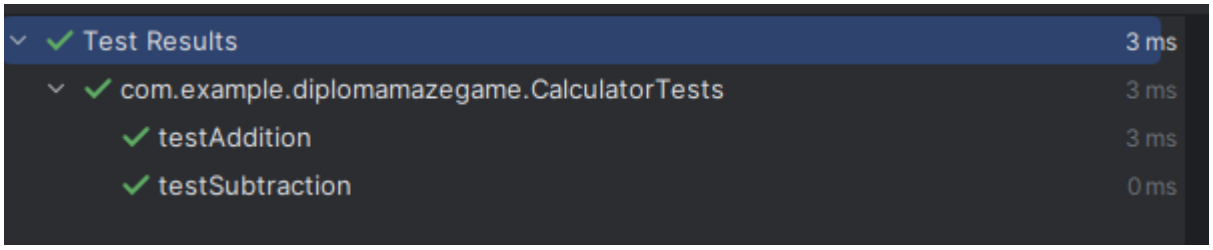

Рисунок 3.22 – Результат виконання Unit тестів

ВИСНОВКИ

У рамках кваліфікаційної роботи було розроблено і реалізовано гру-лабіринт з елементами самонавчання і ускладненням проходження враховуючи дії гравця. Серед головних переваг застосунку можна виділити його простоту інтерфейсу, який не перевантажено зайвими елементами, цікаву основу гри, яка будується на постійному ускладненні і постійних спробах стрибнути вище і далі, ніж в попередній раз і поєднання цієї гри з перспективною ідеєю вікторин.

Для цього були вирішені такі завдання:

– проведено аналіз існуючих алгоритмів побудови лабіринту;

– проведено аналіз існуючих ігор-лабіринтів задля визначення елементів, які можуть бути використані;

– розробити алгоритми побудови лабіринту у матричному вигляді;

– реалізовано алгоритм переведення матричного лабіринту у графічний, з використанням базових інструментів та бібліотек Android розробки;

– реалізовано алгоритм ускладнення лабіринту та тасування алгоритмів;

– реалізовано алгоритм таймінгу та випадкових ускладнень при проходженні;

– реалізовано логіку підрахунку винагороди за рівні та загальну їх статистику.

Для подальшої популяризації застосунку і його успішного розвитку необхідно покращення інтефейсу, створення нових рівні та розширення загального функціоналу. Головними варіантами є додавання нових видів вікторин на різні тематики, додавання нових режимів гри, наприклад мережевої гри, де виграє той, хто отримає найбільшу кількість очок за найменший час.

Додатково можна інтегрувати режими з ворогами, аби вони перешкоджали проходженню гри, і необхідно було б швидко знаходити спосіб їх знешкодити аби зачинити. Саме ці режими відкривають найбільші горизонти до розвитку, бо можна інтегрувати дуже комплексні варіації штучного інтелекту, який буде вчитись на діях гравця і адаптуватися до них.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Kinoshenko D., Kobylin O., Mashtalir S., & Stolbovyi M. (2019, March). Metric video retrieval speedup by irrelevant data elimination. In Eleventh International Conference on Machine Vision (ICMV 2018) (Vol. 11041, pp. 176-183). SPIE.

2. Unraveling the curious history of videogame mazes and labyrinths. URL: https://www.pcgamer.com/unravelling-the-curious-history-of-videogame-mazes-and-labyrinths/ (дата звернення 01.04.2024)

3. 44 different types of mazes and labyrinths. URL: https://www.doyoumaze.com/blog/36-different-types-of-mazes-and-labyrinths (дата звернення 01.04.2024)

4. The Witness Review. URL: https://www.ign.com/articles/2016/01/25/the-witness-review (дата звернення 02.04.2024)

5. Kotlin overview. URL: https://developer.Android.com/kotlin/overview (дата звернення 12.04.2024)

6. Тітов С.В., Тітова О.В., Чорна О.С. (2022). Опис нескоротних наборів ознак в приблизних множинах з використанням систем числення. Збірник наукових праць Харківського національного університету Повітряних Сил. № 1(71), с. 106-110.

7. Jetpack Compose UI App Development Toolkit. URL: https://developer.android.com/develop/ui/compose (дата звернення 14.04.2024)

8. Кобилін О. А., & Творошенко І. С. (2021). Методи цифрової обробки зображень.

9. Fun With Python #1: Maze Generator. URL: https://medium.com/swlh/fun-with-Python-1-maze-generator-931639b4fb7e (дата звернення 20.04.2024)

10. Chaquopy 15.0. URL: https://chaquo.com/chaquopy/doc/current/ (дата звернення 21.04.2024)

11. Prim’s Algorithm – Explained with a Pseudocode Example. URL: https://www.freecodecamp.org/news/prims-algorithm-explained-with-pseudocode/ (дата звернення 22.04.2024)

12. Kruskal’s Minimum Spanning Tree (MST) Algorithm. URL: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/ (дата звернення 22.04.2024)

13. Dubnitskiy, V., Kobylin, A., Kobylin, O., Kushneruk, Y., & Khodyrev, A. (2023). Обчислення значень функції Харрінгтона (функції бажаності) при інтервальному визначенні її аргументів. Advanced Information Systems, 7(1), 71-81.

14. Difference between Prim’s and Kruskal’s algorithm for MST. URL: https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/ (дата звернення 25.04.2024)

15. Daradkeh Y.I., Tvoroshenko I., Gorokhovatskyi V., Latiff L.A., and Ahmad N. (2021). Development of Effective Methods for Structural Image Recognition Using the Principles of Data Granulation and Apparatus of Fuzzy Logic. IEEE Access, 9, pp. 13417-13428.

16. Beginner 1: Explore ideas. URL: https://help.figma.com/hc/en-us/articles/4405269899287--Beginner-1-Explore-ideas (дата звернення 28.04.2024)

17. Cloud Firestore. URL: https://firebase.google.com/docs/firestore (дата звернення 01.05.2024)

18. Gradle Kotlin DSL Primer. URL: https://docs.gradle.org/current/userguide/Kotlin_dsl.html (дата звернення 03.05.2024)

19. Firebase Authentication. URL: https://firebase.google.com/docs/auth (дата звернення 05.05.2024)

20. Руденко, Д., Лотвінова , В., & Безверха , Є. (2023). Навчання нейронної мережі в задачах обробки даних. Collection of Scientific Papers «SCIENTIA», (September 22, 2023; Singapore, Singapore), pp. 94–95.

21. Tvoroshenko, I., & Almakaieva, A. (2020). Application of procedural generation of game content using software algorithms.

22. Дубницький, В. Ю., Кобилін, А. М., & Кобилін, О. А. (2021). Виконання на мобільних пристроях арифметичних операцій з використанням аксіом класичного та нестандартного інтервального аналізу.

23. Kobylin O., Vyskrebentseva S., & Petrova R. (2019). Обробка даних, що містять пропуски в задачах кластеризації. Системи управління, навігації та зв'язку. Збірник наукових праць, 5(57).

24. Mustafa, S. K., Kopot, M., Ahmad, M. A., Lyubchenko, V., & Lyashenko, V. (2020). Interesting Applications of Mobile Robotic Motion by using Control Algorithms. International Journal, 9(3).

25. Kuzomin, O., & Lyashenko, V. (2022). Key Elements of a Specialized Complex for Solving Modeling Problems.

26. Yousef Ibrahim Daradkeh & Irina Tvoroshenko (2020), Technologies for Making Reliable Decisions on a Variety of Effective Factors using Fuzzy Logic. International Journal of Advanced Computer Science and Applications(IJACSA), pp. 11(5),

27. Ahmad, M. A., Tvoroshenko, I., Baker, J. H., & Lyashenko, V. (2019). Computational complexity of the accessory function setting mechanism in fuzzy intellectual systems.

28. Tvoroshenko, I. S., & Maksimenko, H. (2021). To the question of analysis of existing mechanisms of web application testing.

29. Yanholenko, O., Cherednichenko, O., Yakovleva, O., & Arkatov, D. (2020). A Model for Estimating the Security Level of Mobile Applications: A Fuzzy Logic Approach (Doctoral dissertation).

30. Build local unit tests. URL: https://developer.android.com/training/testing/local-tests (дата звернення 14.05.2024)