МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки
Центр післядипломної освіти
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження методів оптимізації та прискорення рендера елементів FlatList у мобільному додатку

Виконав:
студент 2 курсу, групи ІПЗздм-22-1
Петроченков П. М.
(прізвище, ініціали)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-наукова
Керівник доц. Кириченко І.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. Кафедри _____ З.В. Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Центр	Центр післядипломної освіти
Кафедра	Програмної інженерії
Рівень вищої освіти	другий (магістерський)
Спеціальність	121 – Інженерія програмного забезпечення
	(код і повна назва)
Тип програми	освітньо-наукова програма
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«___» _____202_ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Петроченкова Павла Михайловича

(прізвище, ім'я, по-батькові)

1. Тема роботи «Дослідження методів оптимізації та прискорення рендера елементів FlatList у мобільному додатку»

затверджена наказом університету від « 22 » квітень 2024 р. № 60Стз

2. Термін подання студентом роботи до екзаменаційної комісії « 11» червня 2024 р.

3. Вихідні дані до роботи *оптимізація продуктивності компонента FlatList у мобільних додатках, розроблених за допомогою React Native, шляхом застосування різних модифікацій і підходів для покращення рендерингу та ефективності використання ресурсів.*

4. Перелік питань, що необхідно опрацювати в роботі *вступ, аналіз предметної галузі, методи вирішення проблеми, проведення практичного експерименту та аналіз отриманих результатів, опис прототипу додатку, висновки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	12.02 – 28.02.24	Виконано
2	Аналіз та вибір моделей для дослідження	20.02 – 14.03.24	Виконано
3	Аналіз та моделювання предметноїобласті	17.02 – 28.03.24	Виконано
4	Планування експериментів	25.02 – 28.04.24	Виконано
5	Програмна реалізація інтеграції API	25.02 – 01.0.5.24	Виконано
6	Експериментальні дослідження	02.04 – 15.05.24	Виконано
7	Аналіз результатів експериментальних досліджень та розробка рекомендацій	20.04 – 20.05.24	Виконано
8	Написання та оформлення статті та тез доповіді	17.04 – 23.05.24	Виконано
9	Підготовка пояснювальної записки	01.04 – 26.05.25	Виконано
10	Підготовка презентації та доповіді	26.05 – 02.06.24	Виконано
11	Підготовка презентації та доповіді	03.05 – 08.06.24	Виконано
12	Рецензування	08.05 – 14.06.24	Виконано
13	Занесення диплома в електронний архів	15.06.2024	Виконано
14	Попередній захист	15.06.2024	Виконано
15	Допуск до захисту у зав. кафедри	17.06.2024	Виконано

Дата видачі завдання « 12 » лютого 2024 р.

Студент _____ Петроченков П.М.
(підпис)

Керівник роботи _____ доц. Кириченко І.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 78 с., 12 рис., 15 табл., 33 джерел.

АЛГОРИТМИ КЕШУВАННЯ, АЛГОРИТМИ РЕНДЕРУ, FLATLIST, REACT NATIVE, RENDER OPTIMIZATION, VIRTUALIZATION.

Об'єктом дослідження є FlatList в контексті мобільної розробки на платформі React Native, зокрема його вплив на продуктивність додатків при відображенні великих наборів даних на мобільних пристроях.

Метою роботи є аналіз особливостей FlatList у React Native, вивчення його ефективності у віртуалізації та кешуванні даних, а також розробка оптимізаційних стратегій для підвищення продуктивності мобільних додатків з великими обсягами інформації.

Методами розробки та проектування є:

- експериментальне дослідження: тестування продуктивності FlatList з різними обсягами даних на різних типах пристроїв;
- технічний аналіз: вивчення внутрішньої реалізації FlatList та механізмів віртуалізації та кешування;
- розробка прототипів: створення додаткових функцій та оптимізаційних патчів для існуючої реалізації FlatList, з метою покращення її ефективності.

Ця структура дасть можливість глибоко аналізувати та оцінити потенційні переваги та недоліки FlatList, а також розробити методики для їх вирішення чи покращення.

CACHING ALGORITHMS, RENDERING ALGORITHMS, FLATLIST, REACT NATIVE, RENDER OPTIMIZATION, VIRTUALIZATION.

The object of research is FlatList in the context of mobile development on the React Native platform, in particular, its impact on the performance of applications when displaying large data sets on mobile devices.

The purpose of the work is to analyze the features of FlatList in React Native, to study its effectiveness in virtualization and data caching, as well as to develop optimization strategies to improve the performance of mobile applications with large amounts of information.

The development and design methods are:

- experimental research: testing the performance of FlatList with different volumes of data on different types of devices;
- technical analysis: studying the internal implementation of FlatList and mechanisms of virtualization and caching;
- development of prototypes: creation of additional functions and optimization patches for the existing implementation of FlatList, in order to improve its efficiency.

This framework will provide an opportunity to deeply analyze and evaluate the potential advantages and disadvantages of FlatList, as well as to develop techniques to solve or improve them.

Умови публікації пояснювальної записки

Я, Петроченков Павло Михайлович ,
студент групи ІПЗздм-22-1 здобувач вищої освіти на другому (магістерському) рівні кафедра програмної інженерії ,
(повна назва кафедри)
заявляю: моя кваліфікаційна робота на тему
Дослідження методів оптимізації та прискорення рендера елементів FlatList у мобільному додатку ,
(назва роботи)
що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

У наш час, коли життєвий темп невпинно зростає, мобільні застосунки відіграють ключову роль у нашій повсякденному взаємодією зі світом технологій. Вони не просто забезпечують негайний доступ до потрібної інформації та розваг, але й служать посередником до безмежних можливостей сучасного цифрового суспільства, пропонуючи все від онлайн-банкінгу до віртуальних зустрічей, від дистанційної освіти до підтримки здорового способу життя через фітнес-додатки.

У цьому світі мобільних інновацій, оптимізація продуктивності застосунків не просто бажана, але й абсолютно необхідна. Продуктивність стає вирішальним фактором у успіху додатка, оскільки користувачі очікують швидкої та гладенької реакції від своїх застосунків без відстрочень або затримок. Спроможність ефективно управляти великими обсягами даних, особливо в спискових структурах, є фундаментальною вимогою для забезпечення високої продуктивності та рівня задоволення користувачів.

Детальне дослідження механізмів, які лежать в основі компонента FlatList в React Native, стає не тільки актуальним, але й стратегічно важливим завданням. Розуміння тонкощів управління пам'яттю, рендерингу елементів, які перебувають в зоні видимості, та стратегій лінивого завантаження може відкрити шляхи до оптимізації, які підуть на користь кожного аспекту мобільної взаємодії.

Таким чином обрана тема є актуальна як для розробників, так і для розвитку науки, адже результати даної роботи дозволять приймати більш якісні рішення у виборі методів для роботи зі списками та можуть бути гарною відправною точкою у подальших дослідженнях.

Метою роботи є визначення властивостей FlatList які мають вплив на перфоманс та швидкодію списку.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– зробити аналіз та порівняння існуючі методи покращення роботи списку;
– визначити метрики, за допомогою яких буде проведено оцінювання;
– розробити застосунок для проведення експерименту;
– виміряти та підрахувати значення обраних метрик для кожного з методів;

– надати рекомендації щодо використання кожного з методів;

– зробити аналіз впливу групи методів.

Об’єктом дослідження є швидкодія списку, який розроблені за допомогою бібліотеки React Native.

Предметом дослідження є властивості та методи для покращення швидкодії списку.

Методами дослідження є проведення вимірів часу завантаження списку, кількості пікселей які пропускаються при швидкій прокрутці утворюючи так звану “blank area”, які базуються на цих даних.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

У динамічному ритмі сучасного цифрового світу мобільні додатки стали незамінним засобом комунікації та доступу до інформації, пропонуючи небачену раніше зручність та мобільність. Смартфони та планшети, які щодня використовуються мільярдами людей, стали носіями не тільки індивідуального способу життя, але й ключових елементів сучасного бізнесу, навчання та розваг.

Завдяки розширенню можливостей мобільного зв'язку, компанії нині залучають клієнтів за допомогою інноваційних застосунків, розширюючи ринок збуту та взаємодію з аудиторією. Мобільні ігри з їх вражаючим графічним та ігровим дизайном стали не просто засобом розваги, а й потужним сегментом ринку з величезним потенціалом доходу.

Це містить як переваги так і недоліки. В статті “Development of the Concept of the Quality Profile of Mobile Applications” [1] можна знайти основні проблеми мобільної розробки з яких можна виділити, наприклад, проблему безопасноті. Широке використання мобільних додатків створює не лише можливості для зручного доступу до інформації та розваг, але й підвищує ризики, пов’язані з надмірним використанням технологій та втратою особистої інформації через несанкціонований доступ або витоки даних.

Тим не менш, потенціал інновацій у мобільних застосунках безперечно великий [2]. Вони сприяють розвитку телемедицини, дистанційної освіти, фінансових технологій та багатьох інших галузей, пропонуючи нові форми взаємодії між людьми та установами. Розробники мобільних додатків продовжують досліджувати штучний інтелект і машинне навчання, щоб створювати все більш персоналізовані та інтелектуальні рішення, які можуть значно покращувати якість життя користувачів.

Не менш важливу роль відіграють застосунки у сферах, які вимагають надійності та точності: охорона здоров'я, освіта, фінанси та логістика. Використовуючи комплексні алгоритми та інтегровані системи, мобільні додатки

пропонують рішення, які спрощують складні процеси та роблять повсякденні задачі більш ефективними.

Проте, мобільні додатки відрізняються від традиційного програмного забезпечення рядом специфічних характеристик. Як показано у дослідженні “Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts” [3] мобільні додатки мають свої особливості:

- мобільність: створені для користування "на ходу", ці додатки мають бути оптимізовані для використання на пристроях з обмеженою робочою площею екрану та мінімалістичним дизайном інтерфейсу;
- API платформи: мобільні додатки активно користуються можливостями пристроїв, такими як камера, GPS, сенсори та сповіщення, для надання більш глибокого та інтуїтивного досвіду користувача;
- оптимізація ресурсів: враховуючи обмежені ресурси мобільних пристроїв, застосунки повинні використовувати пам'ять, процесор і батарею якомога економніше;
- офлайн-функціональність: важливість можливості використання додатків без сталого підключення до Інтернету стимулює розробників створювати механізми локального зберігання даних і синхронізації;
- дистрибуція: присутність в централізованих магазинах додатків вимагає від розробників відповідати специфічним вимогам та керівним принципам кожної платформи.

Сфера мобільної розробки невпинно розвивається, пропонуючи розробникам широкий спектр інструментів і технологій, які можуть задовольнити різноманітні потреби сучасного ринку додатків. Ці інструменти розрізняються за мовами програмування, фреймворками, бібліотеками, інтеграцією API, інтерфейсами розробника та платформами розгортання, кожен з яких має свої унікальні особливості та переваги.

Існує декілька основних мов програмування та фреймворків які пов’язані з мобільною розробкою:

- Swift і Objective-C для iOS [4]: Apple надає ці мови програмування для розробки додатків для своїх пристроїв. Swift є новішою мовою, яка спрощує розробку та покращує безпеку і продуктивність додатків;
- Java і Kotlin для Android [5]: Kotlin, що в даний час є основною мовою розробки на Android, пропонує більшу чистоту коду та безпеку в порівнянні з Java, але обидві мови продовжують бути популярними в розробці додатків для Android;
- C# з Xamarin: завдяки Xamarin [6], розробники можуть використовувати C# для створення кросплатформних мобільних додатків, які мають доступ до практично всіх API і можливостей нативних платформ;
- JavaScript/TypeScript з React Native та Ionic [7]: ці фреймворки дозволяють розробникам використовувати знання веб-розробки для створення кросплатформних додатків, які мають нативний вигляд і відчуття.

Також для кожної системи існує свої робочі середовища та інструменти:

- Android Studio: офіційне інтегроване середовище розробки (IDE) від Google для розробки Android-додатків, яке надає комплексні інструменти для проектування, тестування та дебагу;
- Xcode: IDE від Apple для розробки для iOS, macOS, watchOS і tvOS, що містить комплект інструментів для розробки і тестування додатків під всі продукти Apple;
- Visual Studio з Xamarin: потужне середовище від Microsoft, яке дозволяє розробникам створювати кросплатформні додатки з використанням Xamarin і .NET.

На даний момент існує дві платформи розгортання – по одній на кожну систему:

- App Store та Google Play: головні платформи для дистрибуції мобільних додатків, які забезпечують розробникам доступ до широкої аудиторії по всьому світу;
- TestFlight і Google Play Beta: сервіси для розгортання тестових версій додатків для збору відгуків від користувачів перед повноцінним релізом.

Серед широкого спектру інструментів для розробки, React Native виділяється завдяки своїм унікальним особливостям [8]:

- кросплатформеність: React Native дозволяє одночасно створювати додатки для Android та iOS, використовуючи спільну кодову базу, що значно оптимізує процес розробки;
- спільнота та екосистема: масштабна та активна спільнота підтримує новачків, ділиться досвідом та надає численні розширення та плагіни для вирішення будь-яких завдань;
- використання JavaScript: одна з найпопулярніших мов програмування, JavaScript, робить вхідний поріг для розробників низьким та дозволяє використовувати сучасні веб-технології для створення мобільних додатків.

React Native відрізняється швидкою адаптацією до змінних вимог ринку та підтримкою від Facebook, що робить його одним з найкращих рішень для сучасної мобільної розробки.

React Native надає широкий набір API [9], який дозволяє розробникам створювати мобільні додатки з використанням JavaScript та React. Фреймворк надає доступ до таких інструментів як FlatList, ScrollView, View, SectionList і т.д.

Компонент FlatList у React Native — це втілення передових технологій для створення динамічних та відзивчивих списків у мобільних додатках. Його ключова особливість — це ефективний механізм рендерингу, який обмежує обробку та відображення елементів до тих, що потрапляють у зону видимості користувача. Ця оптимізація значно покращує продуктивність та зменшує споживання ресурсів, особливо при роботі з великими наборами даних.

Аналізуючи FlatList, важливо ідентифікувати його поточні обмеження у контексті широкого спектру сценаріїв використання. Згідно з дослідженням “Boosting UI Rendering in Android Applications” [10] До таких обмежень може належати обробка величезної кількості елементів, затримки при динамічному оновленні даних, а також складність інтеграції з іншими компонентами інтерфейсу користувачах

Провевши порівняльний аналіз FlatList з іншими компонентами, такими як ScrollView або спеціалізованими бібліотеками сторонніх розробників, допомагає розробникам об'єктивно оцінити переваги та недоліки кожного підходу. Тестування може включати в себе експерименти з різними обсягами даних, варіації відображення (наприклад, списки проти сіток) та випробування різних методів оптимізації продуктивності.

Розглянемо стратегії оптимізації для FlatList базуючись на дослідженні “Critical Review and Fine-Tuning Performance of Flutter Applications” [11] де було проаналізовано мобільний додаток написаний на мові Flutter [12]. Flutter є кросплатформенною мовою програмування та має багато спільного з React Native.

Оптимізація пам'яті:

– зменшення споживання пам'яті може бути досягнуто через ретельне управління компонентами та даними, які FlatList відображає. Це може включати в себе кешування об'єктів, що використовуються повторно, та мінімізацію кількості анонімних функцій та зв'язків у рендер-функціях.

Управління станом [13]:

– ефективне управління станом, особливо в контексті динамічного оновлення даних, є ключовим для підтримки високої продуктивності. Використання таких інструментів як Redux або Context API в React може допомогти в організації стану додатка та забезпеченні його консистентності.

Оптимізація інтерфейсу користувача:

– поліпшення інтерфейсу користувача зосереджується на плавності прокрутки та швидкості відгуку. Техніки, такі як динамічне завантаження зображень, асинхронне отримання даних та оптимізація анімацій, можуть значно покращити сприйняття додатка кінцевими користувачами.

Реалізація вищезгаданих стратегій оптимізації вимагає глибокого розуміння як внутрішньої архітектури FlatList, так і загальних принципів розробки мобільних додатків у React Native. Важливо також зосередитися на тестуванні оптимізацій у

реальних сценаріях використання, щоб гарантувати, що вони не лише покращують продуктивність, але й не впливають негативно на інші аспекти додатка.

Постійний розвиток методів оптимізації і вдосконалення компонентів, таких як FlatList, є вирішальним для досягнення високої продуктивності та задоволеності користувачів у динамічному світі мобільної розробки.

1.2 Аналіз існуючих реалізацій

У сучасному світі мобільної розробки, компоненти, такі як FlatList та SectionList в React Native, відіграють ключову роль у створенні ефективних та водночас динамічних інтерфейсів користувача. Вони дозволяють розробникам легко впорядковувати та відображати списки даних з високою продуктивністю та оптимізацією.

При проведенні аналізу компонента FlatList було виявленно, що FlatList – це комплексний компонент, розроблений для оптимального відображення довгих списків даних [14]. Його інтелектуальна система рендерингу відображає лише елементи, які потрапляють у поле зору користувача, що зменшує навантаження на пам'ять пристрою та покращує загальну швидкодію додатка.

Основні характеристики FlatList включають:

- віртуалізація даних: автоматичний рендер лише тих елементів, які видимі на екрані, забезпечує оптимальне використання ресурсів;
- лінива загрузка: можливість поступово завантажувати дані або динамічно додавати елементи при прокрутці допомагає уникнути затримок в інтерфейсі користувача;
- гнучкість: FlatList підтримує горизонтальну прокрутку, відступи між елементами, витягування для оновлення та багато інших можливостей, що робить його незамінним для будь-якого типу списку;
- оптимізація продуктивності: використання ключів для ефективного перерендерингу та можливість кастомізації кожного елемента списку за допомогою різних методів рендерингу.

Проаналізувавши SectionList у порівнянні з FlatListб отримали, що SectionList є розширенням FlatList [15] з додатковою можливістю групування елементів у секції з власними заголовками. Це ідеально підходить для створення структурованих списків, де дані потрібно організувати у логічні групи.

Наведімо особливості SectionList:

- групування даних: легко створюйте розділені секції з власними заголовками для кращої організації контенту;
- індексація: можливість швидкого переходу між секціями забезпечує зручну навігацію по довгим спискам;
- кастомізація: кожна секція може бути кастомізована незалежно, надаючи розробникам гнучкість у візуальному представленні даних.

В свою чергу VirtualizedList є базовим компонентом, на якому побудовані FlatList та SectionList [16]. Він пропонує низькорівневий API для віртуалізації списків, даючи розробникам більше контролю та гнучкості для створення власних компонентів відображення списків.

При порівнянні API яке надає React Native вибір між FlatList та SectionList залежить від специфічних вимог до додатка. Для простих списків, де не потрібна додаткова організація в секції, FlatList пропонує простоту та швидкодію. У випадках, коли дані краще представляти у груповаій формі, SectionList надає додаткові можливості для структурування та категоризації контенту.

Обидва компоненти є потужними інструментами в арсеналі розробника React Native, кожен з яких має свої унікальні переваги для створення високопродуктивних мобільних додатків.

Також можна підкреслити, що FlatList відіграє вирішальну роль, так як SectionList є часним способом вирішення певної проблеми. Однак, незважаючи на свою універсальність та ефективність, FlatList не позбавлений певних обмежень, які можуть вплинути на досвід користувача та загальну продуктивність додатку.

Проаналізувавши FlatList можна знайти такі недоліки [17]:

- виклики з оптимізацією пам'яті: незважаючи на оптимізований механізм рендерингу, який обробляє лише елементи, що відображаються на екрані,

FlatList може стикатися з викликами, пов'язаними з управлінням пам'яттю, особливо при роботі з величезними наборами даних. Це може призвести до збільшення споживання пам'яті та затримок у роботі додатку;

- ключові вимоги: ефективність FlatList значною мірою залежить від коректного використання унікальних ключів для елементів списку. Неналежне управління ключами може спричинити проблеми з рендерингом та непередбачувані оновлення інтерфейсу;
- обмежена підтримка макетів: хоча FlatList підтримує базові макети списку, створення складних макетів, таких як сітки або мозаїки, може вимагати додаткових зусиль та кастомізації, що збільшує складність розробки;
- ручне управління відображенням: для реалізації специфічних взаємодій або анімацій може знадобитися більше ручного керування рендерингом елементів, що може ускладнити розробку та підтримку коду;
- обмеження в оптимізації зображень: вбудована підтримка оптимізації зображень в FlatList обмежена, що може вимагати інтеграції зовнішніх бібліотек або реалізації власних рішень для кешування та асинхронного завантаження зображень.

Існує декілька альтернативних рішень які мають ті чи інші покращення та недоліки у порівнянні з FlatList:

- RecyclerListView [18]: ця бібліотека вводить концепцію повторного використання елементів та ефективного управління ресурсами, пропонуючи покращену продуктивність для відображення великих списків або сіток. RecyclerListView підтримує різноманітні макети та кастомізації, роблячи його гнучким рішенням для різних вимог до макету;
- FlashList від Shopify [19]: цей компонент був розроблений як вдосконалена альтернатива FlatList, з акцентом на підвищену продуктивність та плавність прокрутки. Завдяки оптимізованому управлінню ресурсами та підтримці автоматичного вимірювання елементів, FlashList пропонує покращену продуктивність для складних або об'ємних списків;

– react-native-largelist [20]: спеціалізований на оптимізації відображення об'ємних даних, цей компонент пропонує розширені можливості для роботи з великими списками, включаючи підтримку складних макетів та ефективне управління пам'яттю для забезпечення високої продуктивності.

У глибокому аналізі сучасних підходів до відображення списків у мобільних додатках на платформі React Native, стає очевидною значущість вибору оптимального компоненту. FlatList, як вбудований компонент React Native, забезпечує розробникам зручний інструментарій для створення відзивчивих та продуктивних списків, проте він не є універсальним рішенням для всіх сценаріїв. Розглянемо детальніше аналіз альтернатив FlatList, виходячи з даних GitHub API та статистики завантажень за останній тиждень.

На рис.1 наведено графік з порівнянням аналогів FlatList.

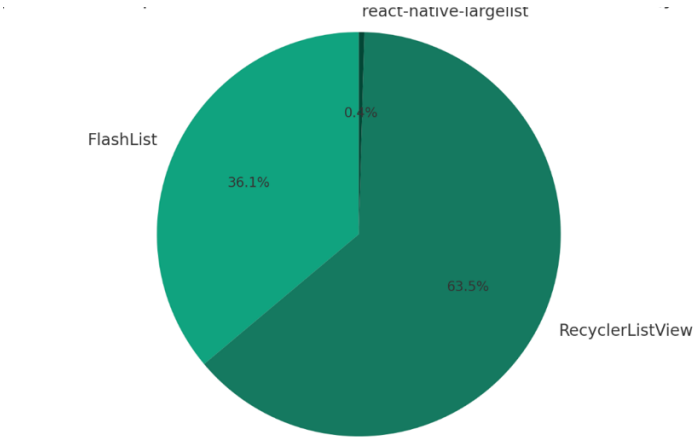

Рисунок 1 – Графік порівняння завантажень аналогів FlatList (рисунок виконаний самостійно)

Зробимо аналіз альтернатив FlatList та проаналізуємо кількість завантажень то популярність даних альтернатив.

RecyclerListView - лідер за популярністю. За даними аналізу, RecyclerListView займає лідируючу позицію з часткою в 63.5% (за даними [18] 350,505 згадок), що свідчить про його широке визнання та впровадження у спільноті розробників React Native. Ця популярність може бути обґрунтована його високою адаптивністю та ефективністю при роботі з великими обсягами даних та складними макетами, надаючи розробникам гнучкі можливості для створення

інтуїтивно зрозумілих і водночас продуктивних користувацьких інтерфейсів.

FlashList - новаторське рішення від Shopify. FlashList, розроблений командою Shopify, демонструє значний інтерес у спільноті з часткою в 36.1% (за даними [19] 199,417 згадок), незважаючи на свою відносно нову появу на ринку. Це підкреслює швидке прийняття цього компоненту спільнотою, завдяки його зосередженню на оптимізації продуктивності списків. FlashList пропонує значні поліпшення у плавності прокрутки та ефективності використання ресурсів, роблячи його відмінним вибором для проектів, де критично важлива висока продуктивність.

React-native-largelist - нішеве рішення для спеціалізованих вимог. З іншого боку, react-native-largelist має значно меншу кількість згадок, лише 0.4% (за даними [20] 2,400 згадок), що може вказувати на його спеціалізоване застосування або меншу відомість в спільноті. Однак, для певних проектів, де особливості react-native-largelist ідеально відповідають технічним вимогам, цей компонент може стати незамінним інструментом, надаючи розробникам необхідну функціональність та оптимізацію.

Проаналізувавши FlatList та його аналізи можна зазначити, що вибір між FlatList та його альтернативами має бути заснований на детальному аналізі специфічних потреб проекту та можливостей кожного компоненту. FlatList залишається надійним вибором для загальних потреб відображення списків, пропонуючи легку інтеграцію та велику гнучкість. Однак, у випадках, коли необхідно досягти вищої продуктивності або реалізувати складні взаємодії, RecyclerListView та FlashList пропонують важливі переваги, які можуть бути критично важливими для успіху проекту. Враховуючи це, за допомогою аналізу вже існуючих рішень та комбінації вдосконалень можна знайти найбільш ефективні способи для оптимізації FlatList компонунта.

1.3 Постановка задачі

Завданням цього дослідження є підвищення продуктивності мобільних застосунків, створених на базі React Native, із фокусом на оптимізацію спискових компонентів. Основна увага приділяється аналізу FlatList — стандартного

інструменту для відображення списків у React Native — та виявленню шляхів для поліпшення його ефективності.

Задачі дослідження даної роботи полягають:

- дослідити основні характеристики FlatList, які впливають на продуктивність: віртуалізацію, горизонтальне та вертикальне відображення, кешування, та підтримку лінивого завантаження;
- виявити основні недоліки FlatList, такі як управління ключами елементів, неефективність при змінних даних, та залежність продуктивності від технічних параметрів пристроїв;
- розробити стратегії для ефективного управління пам'яттю, вдосконаленої віртуалізації елементів та оптимізованого рендерингу, щоб досягнути вищої продуктивності порівняно зі стандартним використанням FlatList.

Наведемо методи дослідження нижче:

- теоретичний аналіз поточних методів оптимізації та існуючих рішень у мобільній розробці;
- проведення експериментального порівняння FlatList із іншими можливими альтернативами;
- розробка та тестування прототипів з впровадженням запропонованих оптимізаційних підходів.

Результатом даної роботи буде:

- формулювання комплексного набору рекомендацій для оптимізації відображення списків у застосунках React Native;
- розробка демонстраційного застосунку, який ілюструє впровадження запропонованих технік і стратегій;
- внесок у спільноту розробників у вигляді документації, кращих практик та вихідного коду оптимізованих компонентів.

Дослідження буде спрямовано на забезпечення покращеного досвіду користувача, надійності мобільного застосунку та ефективності використання ресурсів пристрою, спираючись на сучасні методи та інновації в області програмування та технічної оптимізації.

2 ОПИС ВИРІШЕННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Аналіз FlatList

FlatList - це компонент у React Native, який використовується для ефективного відображення довгих списків даних [21]. Він оптимізований для великих датасетів і може динамічно завантажувати та вивантажувати елементи, коли вони з'являються або зникають з області перегляду, зменшуючи використання пам'яті та підвищуючи продуктивність. Проте, не зважаючи на ці оптимізації, існують певні аспекти FlatList, які можуть бути вдосконалені[22].

Розглянемо основні характеристики FlatList, що впливають на продуктивність [14]:

- віртуалізація - FlatList використовує віртуалізацію для обмеження кількості елементів, які рендеряться в DOM в будь-який момент часу. Однак, алгоритми віртуалізації можуть бути покращені для більш ефективного використання ресурсів, особливо при швидкому прокручуванні;
- горизонтальне та вертикальне відображення - Підтримка обох орієнтацій є важливою, але горизонтальне відображення може вимагати додаткових оптимізацій для забезпечення плавності прокрутки;
- кешування - механізми кешування можуть бути використані для швидшого відновлення відображених раніше елементів, але потребують ретельного управління, щоб уникнути надмірного споживання пам'яті;
- підтримка лінивого завантаження - хоча FlatList підтримує ліниве завантаження елементів, алгоритми для передбачення та оптимізації завантаження можуть бути покращені.

Спираючись на дослідження в статті можна виділити деякі недоліки [23]:

- управління ключами елементів: неоптимальне управління ключами може призводити до перерендерингу більшої кількості компонентів, ніж необхідно. Використання стабільних, унікальних ключів для кожного елемента списку може покращити продуктивність;

– неефективність при змінних даних: при частих оновленнях даних FlatList може виявитися неефективним. Використання компонентів, що мемоізуються, та оптимізація рендерингу елементів може зменшити кількість непотрібних перерендерингів;
– залежність продуктивності від технічних параметрів пристроїв: на слабких пристроях продуктивність може бути значно гіршою. Розробка адаптивних алгоритмів, які адаптують поведінку компонента залежно від можливостей пристрою, може покращити ситуацію.

2.2 Оптимізація кешування

Оптимізація кешування в контексті компонента FlatList у React Native — це процес покращення ефективності зберігання та повторного використання вже завантажених даних або компонентів, з метою зменшення кількості запитів до джерела даних та зниження навантаження на рендеринг.

Кешування зображень – це одне з найважливіших питань в сфері кешування пов’язаного з FlatList. Зображення часто є одними з найбільш ресурсозатратних елементів у спискових структурах.

Головною пропозицією є використання нативних рішень для обробки Image для завантаження та кешування даних. Пропонується використовувати бібліотеку react-native-fast-image яка використовую SDWebImage для IOs та Glide для Android що забезпечує найкращу роботу з картинками. Також ця бібліотека забезпечує ефективне зберігання зображень та їх повторне використання.

Для зменшення кількості операций рендеринга можна використовувати мемоізацію на рівні компонентів списку. React.memo або useMemo може бути використаний для обгортання елементів FlatList. Це запобігає їхньому повторному рендерингу, при умові, що властивості компонентів не змінялися.

Наведемо приклад використання React.memo нижче:

```
const SomeComponent = memo(function SomeComponent(props) {
// ...
});
```

Коли додаток виконує повторні запити до одних і тих же даних, ефективне кешування відповідей від сервера може значно покращити продуктивність. Бібліотеки, такі як React Query, GraphQL, etc, дозволяють автоматизувати процес кешування та управління даними, зменшуючи кількість необхідних мережевих запитів.

Для даних, що рідко оновлюються, можна використовувати локальне зберігання (наприклад, MMKV в React Native) для кешування даних між сесіями [24]. Як зазначено в документації – завдяки використанню нативної частини написаної на C++ це покращую швидкодію з взаємодією даних та дозволяє швидко відновити стан без необхідності здійснення мережевого запиту при повторному запуску додатка.

В React Native зазвичай використовується AsyncStorage для зберігання між сесійних даних, але при розгляді MMKV можна побачити, що даний інструмент зберігає файли в форматі mmap, що забезпечую більше ефективне використання пам’яті, що особливо важливо для мобільної розробки [25].

При роботі з великими колекціями даних важливо ефективно використовувати структури даних і алгоритми для їх обробки. Наприклад, використання іммутабельних структур даних може допомогти зменшити кількість непотрібних перерендерингів.

Наведемо приклад неоптимізованого підходу роботи з колекціями:

```
const data = [
  { id: 1, name: "Item 1", active: true },
  { id: 2, name: "Item 2", active: false },
  // Припустимо, що тут тисячі об'єктів...
];

// Неоптимізовано: спочатку фільтруємо, потім мапимо
const activeItemsNames = data
  .filter(item => item.active)
  .map(activeItem => activeItem.name);
```

Наведемо приклад оптимізованого підходу роботи з колекціями:

```
// Оптимізовано: використання одного проходу з reduce
const activeItemsNamesOptimized = data.reduce((acc, item) => {
  if (item.active) {
    acc.push(item.name);
  }
```

```
    return acc;
  }, []);
```

У неоптимізованому підході кожна операція (filter, map) генерує новий масив, що призводить до додаткових витрат пам'яті та часу на обробку кожного проміжного кроку, особливо при роботі з великими обсягами даних.

Оптимізований підхід з reduce дозволяє зменшити кількість проходжень по масиву та проміжних структур даних, забезпечуючи більш ефективне використання ресурсів, що особливо важливо для високопродуктивних застосунків або при обробці великих масивів даних.

Оптимізація кешування в FlatList є одним з ключових факторів для підвищення продуктивності мобільних додатків на React Native. Використання вищевказаних стратегій дозволить оптимізувати та покращити швидкість роботи FlatList .

2.3 Оптимізація рендеру

Оптимізація рендеру в контексті компоненту FlatList у React Native є дуже важливим аспектом для підвищення продуктивності та забезпечення плавної роботи додатку. Розглянемо декілька стратегій які було наведенно в дослідженні рендерінгу на Android системі [26], які можуть допомогти оптимізувати процес рендеринга.

Мінімізація елементів, які рендеряться. Використання віртуалізації [27]: FlatList автоматично віртуалізує елементи, рендерячи лише ті, що знаходяться у вікні перегляду. Для покращення перфомансу потрібно задати властивості, як-от initialNumToRender та maxToRenderPerBatch, для контролю кількості елементів, які рендеряться спочатку та при прокрутці відповідно. Це забезпечить кількість елементів для рендеру, це потрібно для того бо існують різні девайси та різні розміри екранів.

Оптимізація компонентів елементів. Потрібно уникати анонімних функцій у рендері. Анонімні функції можуть бути більш удобні для використання але треба

памятати, що вони створюються кожний раз при кожному рендері, спричиняючи повторні рендери дочірніх компонентів.

Наведемо приклад Arrow function та Regular function:

```
// анонімна функція яка не має назви
() => {
  console.log(arguments)
}
// звичайна функція
function print() {
  console.log(arguments)
}
```

Ефективне використання ключів. API FlatList надає змогу додати та використовувати ключі для елементів, які допомагають React визначати, які елементи змінилися, додалися або видалилися, оптимізуючи процес рендеринга. Для цього потрібна використовувати властивість keyExtractor.

Наведено інтерфейс keyExtractor:

```
(item: ItemT, index: number) => string;
```

Використовується для отримання унікального ключа для даного елемента за вказаним індексом. Ключ використовується для кешування та як ключ реакції для відстеження перевпорядкування елементів. Стандартний екстрактор перевіряє item.key, потім item.id, а потім повертається до використання індексу, як це робить React.

Також потрібно зазначити, що важливим аспектом є дані та їх попередня обробка. Обробляйте дані до передачі їх у FlatList, замість того, щоб робити це в рендер-функції кожного елемента. Це зменшує кількість обчислень під час рендеринга. Потрібно намагатися уникати зміни даних після рендеру списку як умога сильніше.

Іншою рекомендацією є використовувати ліниве завантаження зображень та даних. Використовуйте ліниве завантаження зображень, щоб вони завантажувались лише тоді, коли потрапляють у вікно перегляду. Для цього можна використовувати react-native-fast-image як і було запропоновано раніше.

Також потрібно враховувати які дані будуть використовуватися та намагатися попередньо завантажити дані. Для даних, які можуть знадобитися користувачу далі, можна використовувати попереднє завантаження, щоб зменшити час очікування.

Для оптимізації рендеру потрібно використовувати легковагові компоненти. Намагайтеся використовувати легковагові компоненти та уникайте важких бібліотек або фреймворків, які можуть сповільнити рендеринг. Потрібно уникати бізнес логіки компонентів та внутрішнього стану який є привязаний саме до компонента. Тобто компонент має бути “Dumb” компонентом.

Застосування цих стратегій дозволить значно покращити продуктивність рендеру в FlatList.

2.4 Оптимізація управління памяттю

Оптимізація управління пам'яттю є критично важливою для підтримки високої продуктивності та стабільності додатків, розроблених за допомогою React Native, особливо при використанні компонентів, як-от FlatList, для відображення великих обсягів даних. Розглянемо декілька стратегій, які допоможуть оптимізувати управління пам'яттю.

Для оптимізації пам’яті можна використовувати додаткові властивості FlatList. FlatList надає властивості, такі як removeClippedSubviews, initialNumToRender, та maxToRenderPerBatch, які можуть допомогти зменшити споживання пам'яті [28], обмежуючи кількість елементів, що зберігаються в пам'яті та рендеряться одночасно. Також слід зазначити, що використання лінивого завантаження даних замість завантаження всього обсягу даних спочатку може значно зменшити використання пам'яті.

Зображення мають великий вплив на використання пам'яті, нажаль нативна реалізація Image має багато недоліків які впливають майже на все [29]. Для роботи з зображеннями використовуйте бібліотеки, такі як react-native-fast-image, для оптимізованого кешування та завантаження зображень. Також потрібно зазначити, що потрібно уникати використання зображень високої роздільної здатності, які не

адаптовані до екрану пристрою. Використання зображень відповідного розміру може значно зменшити споживання пам'яті. Цього можно добитися додавши media властивостіх [30].

Потрібно слідкувати за уникненням витоків пам'яті. Для цього упевніться, що всі події та таймери скасовуються у методі життєвого циклу componentWillUnmount для уникнення витоків пам'яті. Скасуйте усі підписки eventEmitter та eventListener якщо такі використовуються.

Використовуйте стан обережно, уникайте зберігання великих об'єктів у стані компонента, оскільки це може призвести до зайвого використання пам'яті. Для збереження великих обьемів даних потрібно використовувати сховище на нативній частині системи, для цього можна використати MMKV.

Мемоізація обчислень також навантажує пам'ять. Потрібно використову React.memo і useMemo, для уникнення непотрібних рендерів та обчислень, використовуйте React.memo для компонентів та useMemo для обчислень, які не повинні повторюватися при кожному рендері.

Потрібно перевикористовувати компоненти для рендерингу, тобто потрібно використовувати компоненти високого порядку (HOC) та компоненти-обгортки: Розробляйте перевикористовувані компоненти для зменшення кількості дубльованого коду та об'єктів у пам'яті.

HOC, або Higher-Order Component (компонент вищого порядку), — це техніка в React для повторного використання логіки компонентів. HOC дозволяє вам обгорнути один компонент іншим, тим самим додавши або модифікувавши його поведінку без необхідності змінювати сам компонент.

Основні характеристики HOC:

– абстракція та повторне використання: HOC дозволяють абстрагувати спільну функціональність і використовувати її в кількох компонентах;
– композиція: замість успадкування, у React для повторного використання компонентів рекомендується використовувати композицію, де HOC виступають як обгортки навколо існуючих компонентів;

– контракт пропсів: HOC передають пропси до обгортаного компонента, можливо додаючи або модифікуючи їх. Важливо уважно стежити за тим, щоб не виникало конфліктів між іменами пропсів.

Нижче наведемо приклад HOC компонент:

```
function wrapper(WrappedComponent) {
  return class extends React.Component {
    componentDidMount() {
      console.log(`Компонент ${WrappedComponent.name} змонтовано`);
    }

    componentWillUnmount() {
      console.log(`Компонент ${WrappedComponent.name} демонтовано`);
    }

    render() {
      // Передаємо всі пропси в обгортаний компонент
      return <WrappedComponent {...this.props} />;
    }
  };
}

// Використання HOC
const LoggedComponent = withLogging(SomeComponent);
```

Застосування цих стратегій може значно покращити управління пам'яттю в додатках React Native, зменшивши споживання ресурсів та підвищивши загальну продуктивність і відгук додатка.

2.5 Оптимізація ручного управління ключами елементів

Оптимізація ручного управління ключами елементів у FlatList в React Native є важливим аспектом для забезпечення ефективного рендеринга та оновлення списку. Правильне використання ключів може значно покращити продуктивність застосунку, зокрема швидкість прокрутки та час відгуку інтерфейсу. Ось кілька стратегій для оптимізації управління ключами елементів:

Можна зробити генерацію ключів на основі вмісту. Якщо елементи містять унікальний ідентифікатор (наприклад, ID з бази даних), використовуйте його як ключ. Це забезпечує найкращу стабільність та унікальність ключа. ID - це найкращий вибір для ключа елементів. У випадках, коли унікальний ID відсутній,

можна генерувати ключі, використовуючи хеш-функцію від значущих властивостей елемента.

Одна з найрозповсюдженіших помилок є використання індексів елементів як ключів. Індекси масиву можуть бути спокусою використати як ключі, особливо при відсутності унікальних ідентифікаторів. Проте, це може призвести до помилок і непотрібних перерендерів, особливо при зміні порядку або додаванні/видаленні елементів.

Потрібно звернути увагу на ключи при динамічних списках, бо це є дуже важливою частиною при додаванні або видаленні елементів. При додаванні або видаленні елементів забезпечте, щоб ключі для існуючих елементів залишались незмінними. Це дозволяє React ефективно перевикористати компоненти елементів.

Правильне управління ключами елементів у FlatList є важливим для оптимізації продуктивності рендеринга та оновлення списків. Використання унікальних та стабільних ключів, базованих на вмісті елемента, допоможе зменшити непотрібні перерендери та покращити загальну продуктивність додатку.

2.6 Оптимізація динамічного списку

Оптимізація динамічних списків в React Native, зокрема використовуючи компонент FlatList, вимагає ретельного планування та впровадження найкращих практик для забезпечення плавності прокрутки, швидкості завантаження та ефективності пам'яті. Розглянемо декілька стратегій, які можуть допомогти оптимізувати динамічні списки:

Для досягнення данної оптимізації потрібно використовувати FlatList у найбільш ефективний спосіб [31]. Для цтого потрібно задати такі властивості як initialNumToRender, maxToRenderPerBatch та windowSize. Де initialNumToRender – контролює кількість елементів, які рендеряться спочатку, для зменшення навантаження при початковому завантаженні, maxToRenderPerBatch – регулює скільки нових елементів буде рендеритися за кожен цикл обробки подій та windowSize - налаштуйте розмір "вікна" віртуалізації, щоб контролювати, скільки

контенту поза видимою областю рендериться. Саме ці три властивості мають найбільший вплив на продуктивність FlatList.

Для оптимізація елементів списку можна використати попередні рекомендації щодо використовування React.memo для обгортання елементів списку, щоб уникнути їх непотрібного перерендеру та уникнення анонімних функцій.

Слід звернути увагу на уникнення витоків пам'яті. Для цього потрібно відписуватися від подій та таймерів та обмежети кількості слухачів подій, бо занадто багато слухачів подій може сповільнити додаток та призвести до витоків пам'яті.

Виконуючи ці рекомендації, ви зможете значно покращити продуктивність динамічних списків у React Native, забезпечивши користувачам вашого застосунку високий рівень відгуку та плавність прокрутки.

2.7 Підведення висновків аналізу

Було проаналізовано та наданно рекомендації та було виконано комплексну оптимізацію компонента FlatList у React Native, що включає кілька ключових аспектів:

- віртуалізація та оптимізація відображення - ефективне використання віртуалізації дозволило обмежити кількість елементів, що рендеряться в DOM, зокрема при швидкій прокрутці. Це значно зменшило використання пам'яті та підвищило продуктивність;
- підтримка обох орієнтацій та оптимізація горизонтального відображення забезпечили плавність прокрутки та вдосконалили користувацький досвід;
- впровадження механізмів кешування для зображень та елементів списку дозволило швидше відновлювати відображені раніше елементи, зменшуючи час завантаження та оптимізуючи загальну продуктивність;

- використання нативних рішень для кешування зображень, таких як react-native-fast-image, покращило роботу з картинками та знизило вплив на продуктивність;
- використання стабільних та унікальних ключів для кожного елемента списку мінімізувало непотрібні перерендеринги та сприяло ефективному оновленню списку;
- мемоізація компонентів та оптимізація передачі пропсів знизили кількість непотрібних рендерів;
- попередня обробка даних та використання лінивого завантаження забезпечили швидше відображення вмісту та покращили реакцію на взаємодію користувача;
- вдосконалення управління пам'яттю через правильне використання властивостей FlatList, зокрема removeClippedSubviews, та застосування стратегій локального кешування даних між сесіями зменшило загальне споживання ресурсів.

Виконані оптимізації FlatList в React Native сприяли значному підвищенню продуктивності та плавності прокрутки, забезпечуючи високий рівень відгуку інтерфейсу та поліпшений користувацький досвід. Ці підходи демонструють важливість комплексної оптимізації, врахування специфіки даних та уваги до деталей управління пам'яттю та ресурсами, що є ключовими для розробки високопродуктивних мобільних додатків.

3 СТВОРЕННЯ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ ДОСЛІДЖЕННЯ

3.1 Функціональні вимоги

Для визначення впливу різних підходів по оптимізації та модифікації FlatList, було розроблено тестовий мобільний додаток з використанням React Native.

До розробленого застосунку було визначено наступні функціональні вимоги. Інтерфейс користувача:

- екран з налаштуваннями: містить меню з опціями для тестування різних конфігурацій FlatList;
- екран списку: відображає список елементів з можливістю прокручування. Повинно бути реалізовано кілька різних варіантів списків для тестування (наприклад, стандартний FlatList, FlatList з оптимізаціями, альтернативні компоненти списків).

Конфігурації FlatList:

- стандартний FlatList - без додаткових оптимізацій, використовується як базова лінія для порівняння;
- оптимізований FlatList - застосування різних стратегій оптимізації (наприклад, використання мемоізації, кастомних компонентів рендерингу);
- альтернативні компоненти - тестування інших бібліотек та компонентів списків, наприклад, RecyclerListView.

Тестові сценарії:

- віртуалізація - тестування продуктивності списків з різною кількістю елементів (наприклад, 100, 1000, 10000);
- горизонтальне та вертикальне відображення - перевірка впливу на продуктивність при горизонтальному та вертикальному відображенні списків;
- ліниве завантаження - тестування ефективності лінивого завантаження даних.

Метрики продуктивності:

– швидкість рендерингу - час, необхідний для рендерингу списку;
– використання пам'яті - споживання пам'яті під час роботи додатку;
– плавність прокручування - частота кадрів (FPS) при прокручуванні списку.

Збір даних та аналітика: логування метрик - запис метрик продуктивності для кожного тестового сценарію.

Додаткові функції:

– перемикання між конфігураціями - можливість перемикання між різними конфігураціями списків через інтерфейс додатку;
– вибір налаштувань - налаштування кількості елементів у списку, розміру кешу, параметрів віртуалізації тощо.

Презентаційна частина програми має бути розроблена за допомогою бібліотеки React та має залишатися незмінною під час дослідження, тоді як управління станом буде реалізується окремо за допомогою обраних інструментів.

3.2 Розробка презентаційної частини

Для створення застосунку було використано React Native. Ця бібліотека передбачає, що інтерфейс формується як композиція із різних компонентів.

Для того щоб правильно сформулювати усі компоненти для реалізації даного застосунку було створено макет за допомогою сервісу Moqups [32]. Moqups — це онлайн-інструмент для створення прототипів, макетів та схем користувацького інтерфейсу. Він використовується дизайнерами, розробниками та командами для швидкого створення візуальних уявлень майбутніх додатків або веб-сайтів.

Основний екран застосунку передбачає відображення списку елементів. Кожний елемент може містити декілька текс елементів та зображення.

Приклад відображається на рисунку 2.

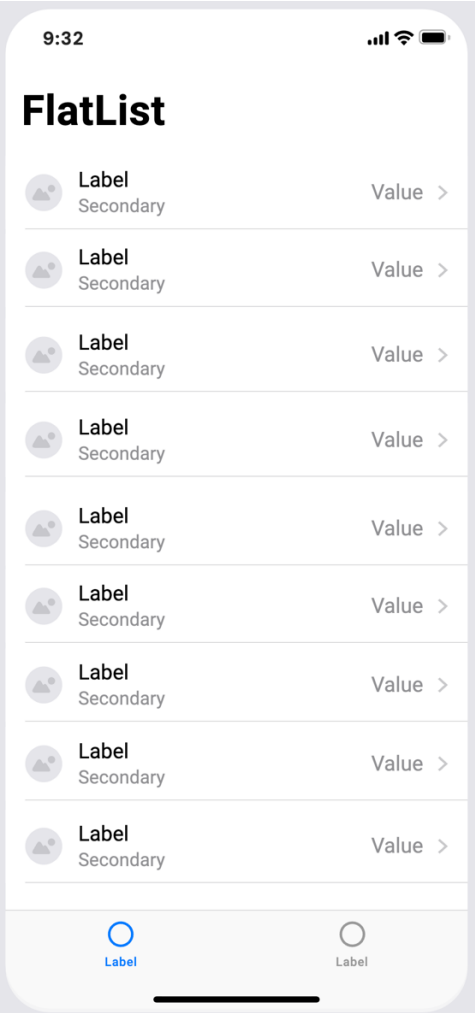

Рисунок 2 – Макет головного екрану (рисунок виконаний самостійно)

На зображенні представлено екран мобільного додатку з використанням компонента FlatList в React Native. Ось детальний опис елементів, які можна побачити на екрані:

Заголовок - великий текст "FlatList" у верхній частині екрану, який слугує заголовком сторінки.

Список елементів - кожен елемент списку складається з декількох підкомпонентів. Зображення це кругла іконка (імовірно аватарка або іконка) зліва від тексту. Основний текст це надпис "Label" великим шрифтом, що знаходиться праворуч від зображення. Підтекст - надпис "Secondary" меншим шрифтом під основним текстом та інше.

Нижня панель навігації - дві кнопки з текстом "Label" і "Label", кожна з яких має іконку зверху. Кнопки розташовані на нижній панелі та використовуються для перемикання між різними екранами або функціональностями в додатку.

Також додаток має другий екран який є екраном налаштування та конфігурації компоненту. Приклад даного екрану зображено на рисунку 3.

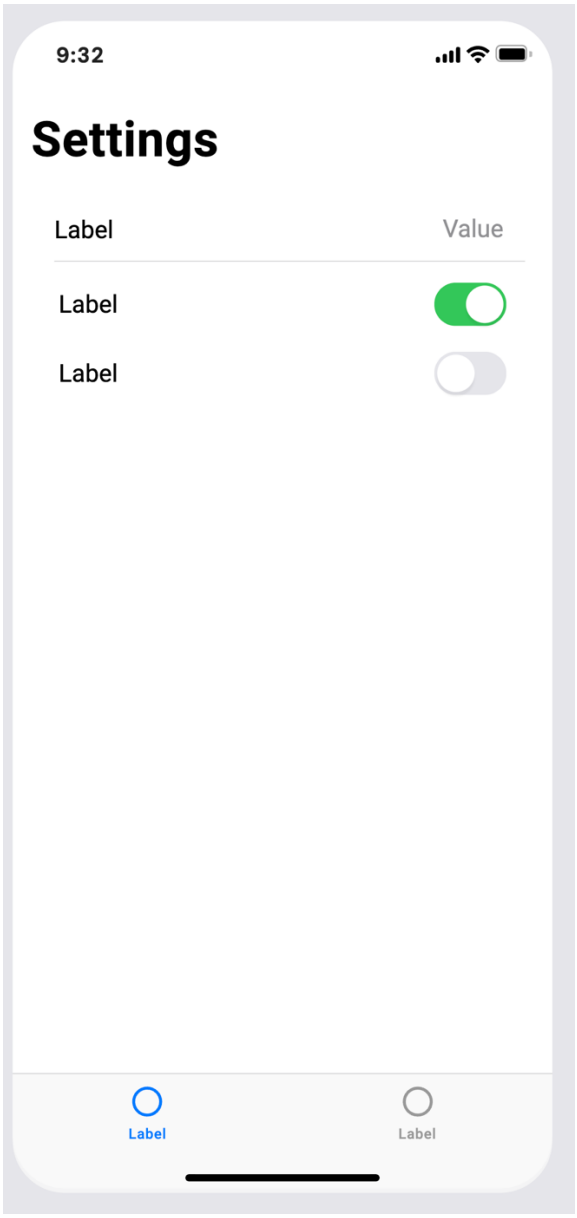

Рисунок 3 – Макет екрану налаштувань (рисунок виконаний самостійно)

На зображенні представлено екран налаштувань мобільного додатку. Ось детальний опис елементів, які можна побачити на екрані:

Заголовок - великий текст "Settings" у верхній частині екрану, який слугує заголовком сторінки налаштувань.

Список налаштувань - кожен елемент списку складається з основного тексту, що знаходиться зліва та перемикача (Switch) - розташований праворуч, дозволяє включати або вимикати певну функцію.

Нижня панель навігації така сама як на головній сторінці.

Після створення макетів було виділено основні компоненти які потрібно створити для даного застосунку:

- App;
- Navigation tab;
- Icon;
- Text;
- Row;
- List;
- Checkbox;
- TextInput;
- Image;
- Title.

Компоненти було реалізовано за допомогою JSX синтаксису. JSX це синтаксичне розширення для JavaScript, яке дозволяє писати HTML-подібний код всередині JavaScript. Він використовується в React для опису інтерфейсу користувача (UI). JSX дозволяє розробникам легко створювати React компоненти, які генерують UI.

Для додавання навігації між сторінками було використано react-navigation [33]. React Navigation — це популярна бібліотека для додавання навігації в React Native додатки. Вона забезпечує простий спосіб перемикання між різними екранами в додатку. Також саме ця бібліотека була використана для створення bottom tab navigation.

Для стилізації компонентів було вирішено не використовувати ніякі додаткові бібліотеки як Tailwind або Styled components – а використовувати звичайні StyledSheed які надає API React Native. Так як головною метою додатку є перевірка роботи Flatlist - було вирішено розробити простий та зрозумілий UI/UX

Для реалізації можливості налаштування FlatList у самому додатку було створено окрему сторінку з налаштуваннями. Для даного функціоналу використовується бібліотека Zustand, яка надає змогу зробити глобальний сховище. Zustand це нова бібліотека яка працює на базі Redux, але є більше простою та легкою у використанні у порівнянні з Redux Saga або Redux Thunk. Даний інструмент розрахований на маленькі та середні проекти, що ідеально підходить нашим вимогам. Головна перевага даної бібліотеки це простата у використанні. Нижче наведено приклад створення сховища.

```
const useBearStore = create((set) => ({
  bears: 0,
  increasePopulation: () => set((state) => ({ bears: state.bears + 1
})),
  removeAllBears: () => set({ bears: 0 }),
}))
```

Для перевірки перфомансу FlatList було використано API FlashList, а саме властивості onLoad та onBlankArea.

Подія onLoad виникає, коли список виводить елементи на екран. Він також повідомляє elapsedTimeInMs, тобто час, який знадобився для малювання елементів. Це потрібно, оскільки FlatList не відображає елементи в першому циклі. Елементи малюються після вимірювання в кінці першого рендерингу. Для відображення часу було створено функцію наведену нижче.

```
const onLoadListener = useCallback(({elapsedTimeInMs}) => {
    console.log(`FlatList loaded in ${elapsedTimeInMs}ms`);
  }, []);
```

Подія onBlankArea у свою чергу повертає дві властивості maxBlankArea та cumulativeBlankArea.

– CumulativeBlankArea — це загальна порожня область, яку бачив користувач під час прокручування списку. Він вимірюється в пікселях і є сумою всіх проміжків між елементами, які ще не були відрендерені. Менше значення означає менше пустого місця та кращу продуктивність;
– MaxBlankArea — максимальна порожня область, яку бачив користувач під час прокручування списку. Він також вимірюється в пікселях і є найбільшим розривом між елементами, які ще не були відрендерені.

Менше значення означає менш помітний порожній простір і кращий досвід користувача.

Для заміру даного функціоналу було використано хук useBlankAreaTracker. Нижче наведено приклад використання даного хука.

```
function MyListComponent(){
   const ref = useRef(null)
   const [blankAreaTrackerResult, onBlankArea] =
useBlankAreaTracker(ref)

    // Only when the component will unmount then you willl see the
output
    // As we set the console in cleanUp function
    // It will show you then the latest output of Blank Area when
unmount
    useEffect(() => {
        return () => {
         console.log('On blank area: ', blankAreaTrackerResult)
        }
     }, [])

    return (
      <FlatList
       {...props}
       ref={ref}
       onBlankArea={onBlankArea}
      />
    )
}
```

Наведемо приклад головного екрану який вдалося розробити на рисунку 3.

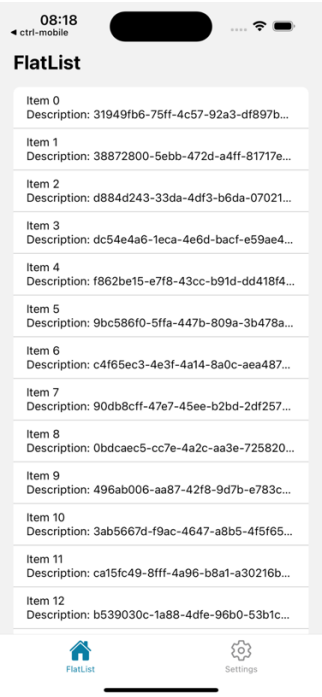

Рисунок 4 – Головний екран (рисунок виконаний самостійно)

Наведемо приклад екрану налаштувань списку який вдалося розробити на рисунку 4.

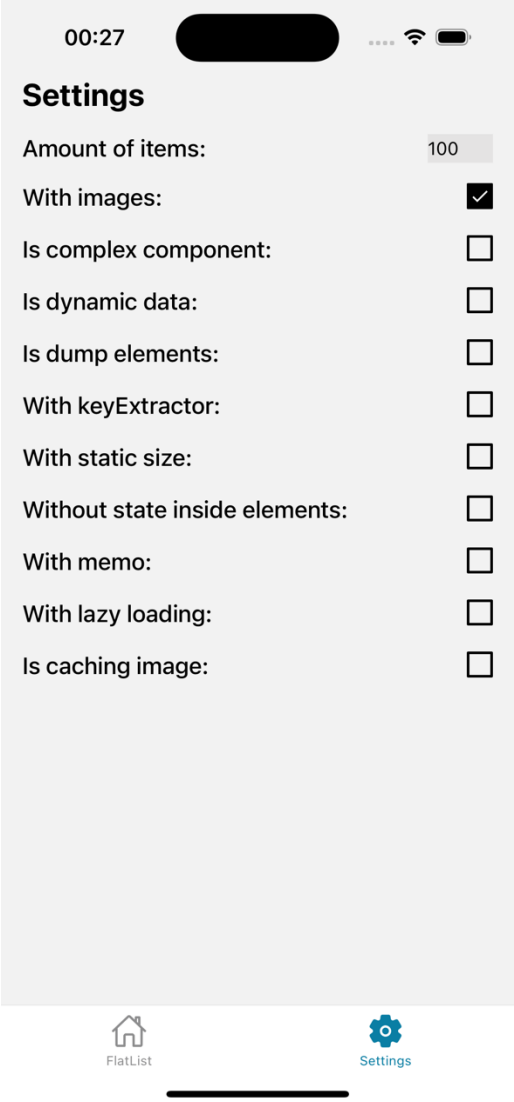

Рисунок 5 – Екран налаштувань (рисунок виконаний самостійно)

У застосунку було реалізовано всі функціональні вимоги, необхідні для коректного проведення дослідження.

3.3 Реалізація управління FlatList

Як було сказано раніше для управління FlatList було створено окрему сторінку в додатку “Settings”. Ця сторінка містить декілька окремих компонентів які відповідають за налаштування. Усі налаштування зберігаються у глобальному сховищі. Налаштування можна умовно поділити на дві частини. Одна частини – це налаштування які змінюють складність списку та складність рендеру елементу списку. Друга частина – це налаштування які мають вплив на перфоманс то можуть

змінювати час рендеру та інші параметри при незмінних налаштуваннях складності рендеру списку та елементів списку.

Для налаштування складності рендеру списку можна виділити декілька властивостей які наведено нижче:

- Amount of Items - налаштування кількості елементів в списку є одним з найочевидніших частин налаштування додатку. Саме змінна кількості даних найбільше. Впливає на час рендеру повного обсягу інформації, саме тому було обрано це налаштування. Для налаштування кількості елементів було створено окремий TextInput який надає змогу глобальна змінювати сховище. Користувач може ввести любе значення, яке буде типом Number та більше 100;
- Image – на сторінці налаштувань є можливість додати до елементу списку зображення. Є декілька опцій зображень. Перша опція це додати просте зображення використавши API самого фреймворку React Native, тобто не як не оптимізоване зображення. Друга опція це використання додаткової бібліотеки для покращення процесу завантаження та зберігання зображень в пам’яті. Тобто можна буде побачити різницю між використанням списку без зображень та з різними зображеннями як оптимізованими так і не оптимізованими;
- Complex component – дана властивість відповідає за складність рендеру компоненту. Тобто якщо даний пункт увімкнено то елементи матимуть додаткові стилі які будуть навантажувати UI thread при рендері, що в свою чергу буде навантажувати рендер;
- Dynamic data – дана властивість відповідає за дані які рендерить список. Якщо dynamic data включена то дані будуть генеруватися наново кожну хвилину. Це допоможе перевірити як список реагує на ререндер компонентів.

Перелічемо властивості які будуть мати вплив на перфоманс самого списку:

- Is dumb elements – властивість яка відповідає за JS thread при рендері елементів списку. Тобто якщо даний опшин ввімкнино, при рендері

елменту списку буде додано штучне навантаження на JS thread за допомогою додаткових математичних вичеслень при рендері елементу;

- With keyExtractor – властивість відповідає за наявність ідентифікатора при рендері елементів. Має прямий вплив на FlatList при рендері великої кількості елементів;
- With static size – даний рядок показую чи використовуємо ми властивість estimatedItemSize. EstimadItemSize — це одне числове значення, яке повідомляє список про приблизний розмір елементів перед їх відтворенням. Потім список може використовувати цю інформацію, щоб вирішити, скільки елементів йому потрібно намалювати на екрані перед початковим завантаженням і під час прокручування;
- With recycling way of rendering – вмикає режим при якому змінюється підходи до рендеру елементів. Коли елемент виходить із вікна перегляду, замість того, щоб бути знищеним, компонент повторно рендериться з іншим атрибутом елемента. Дана властивість має великий вплив на перфоманс так як змінює один з головних принципів рендеру елементів та буде застосовано завжди так як це змінює поведінку спуску;
- Without state inside elements – для відображення списку рекомендується робити stateless компоненти. Дана властивість відповідаю за додання у кожний елемент компоненту Checkbox який має свій стан незалежний від глобального сховища;
- With memo – memorization це дуже розповсюджена практика в React та React Native. Ця властивість відповідаю за запам’ятовування елементів та функцій рендеру у пам'ять;
- With lazy loading – API FlatList надає змогу вимкнути ліниве завантаження, це надає змогу протестувати список з та без лінивого завантаження;
- Is caching image - кешування зображень. Увімкнення кешування зображень для підвищення продуктивності. Ця частина була реалізована за допомогою бібліотеки react-native-fast-image. Це нативне рішення для

React Native яке маю продвинуті алгоритми кешування та завантаження зображень на IOS та Android.

Сторінка Settings має багато різних налаштувань які змінюють як і спосіб роботи так і складність самого списку, що дає змогу проаналізувати різні ситуації та сценарії для більш комплексного аналізу роботи списку.

Розроблений додаток дозволяє комплексно досліджувати продуктивність FlatList у різних конфігураціях та умовах. Завдяки можливості налаштування параметрів списку та аналізу метрик продуктивності, можна визначити оптимальні підходи до використання FlatList у мобільних додатках. Реалізація додатку з використанням React Native та сучасних бібліотек забезпечує ефективність та зручність розробки, що робить його корисним інструментом для дослідження та оптимізації компонентів списків у мобільних додатках.

4 ОПИС ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

При проведенні дослідження було створено три конфігурації списку які будуть мати різницю у складності рендеру елементів списку.

Перша конфігурація – це список в якому усі елементи мають лише текст, та не мають ніяких додаткових ускладнень роботи

Друга конфігурація – це список, який має зображення та увімкненою властивість isComplexData, яке в свою чергу додає навантаження на UI thread. В даному випадку це досягаються завдяки властивостям shadows. У реальному проекті – це може бути як складні стилі так і анімації на компонентах списку.

Третя конфігурація – це список, такий самий як друга конфігурація, але з увімкненою властивістю isDynamicData. Що буде імітувати оновлення даних кожну секунду.

Для порівняння даних конфігурацій було зроблено заміри часу реакції без покращень з кількістю елементів у 1000. Таблиця 1 демонструє загальні результати.

Таблиця 1 – Таблиця із замірами властивостей для різних конфігурацій (таблиця виконана самостійно)

	maxBlankArea(px)	cumulativeBlankArea(px)	Loading time(ms)
Конфігурація 1	14431	20119.6	21
Конфігурація 2	20362	29034	25
Конфігурація 3	20394	36112	31

При розгляданні даної таблиці можна побачити, що найкращі результати у конфігурації 1, що в свою чергу є зрозумілим так як даний список є самим простим з усіх вище наведених.

Створимо графік на базі таблиці для більш детального порівняння (див. рис.6).

На наведених графіках представлені результати для трьох конфігурацій списків за параметрами maxBlankArea, cumulativeBlankArea та Loading time.

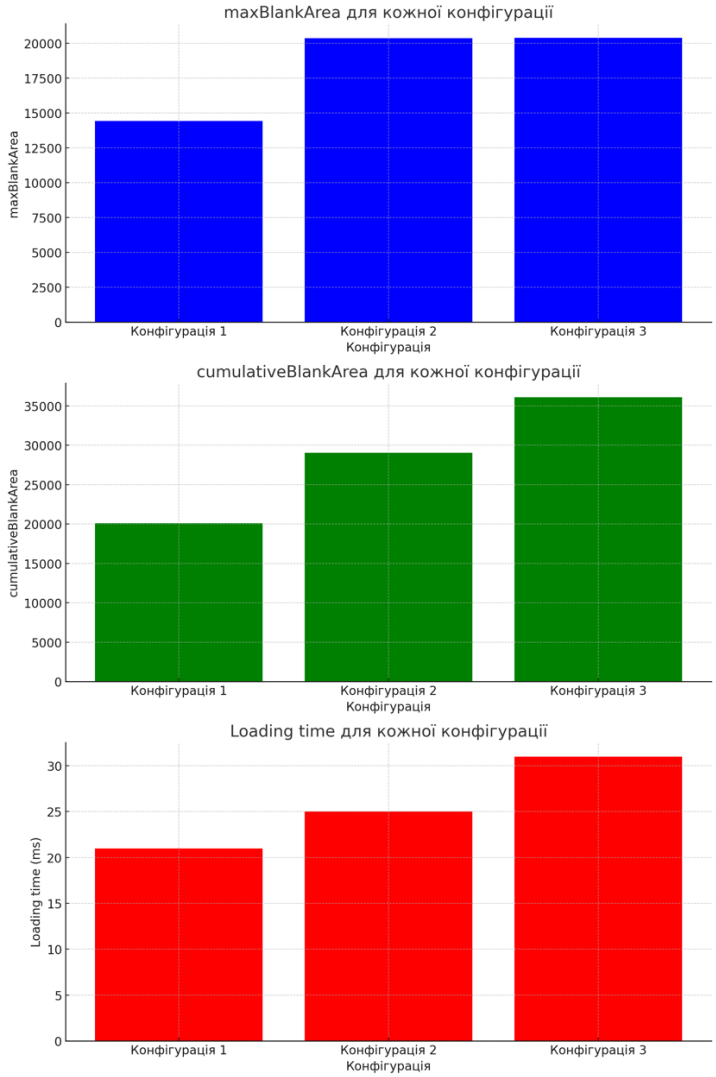

Рисунок 6 – Графік порівняння конфігурацій (рисунок виконаний самостійно)

Розглянемо детальніше властивість maxBlankArea:

– конфігурація 1 має значно меншу площу порожніх місць порівняно з іншими конфігураціями;
– конфігурації 2 та 3 мають подібні значення maxBlankArea, що вказує на схоже навантаження на UI thread.

Розглянемо детальніше властивість cumulativeBlankArea:

– конфігурація 1 демонструє найменше значення cumulativeBlankArea, що вказує на ефективніше використання простору;

– конфігурація 2 має більше cumulativeBlankArea, ніж конфігурація 1, але менше, ніж конфігурація 3;
– конфігурація 3 має найбільше cumulativeBlankArea через постійне оновлення даних.

Розглянемо детальніше властивість Loading time:

– конфігурація 1 має найшвидший час завантаження;
– конфігурація 2 показує збільшення часу завантаження через додаткові стилі та анімації;
– конфігурація 3 має найповільніший час завантаження через постійні оновлення даних.

Зосереджуючись на цих фактах було вирішено взяти за основу конфігурацію 2 для подальших випробувань властивостей непосереднє пов’язаних з швидкодією списку. Конфігурація 1 має найкращі результати, але не є репрезентативною для більшої кількості списків в реальних проектах, так як відображаються лише текстові дані. Конфігурація 3 має динамічні дані – тому це скоріше дуже рідкі випадки коли дані мають оновлюватися постійно.

Зважаючи на сказане раніше було обрано робити всі тестування за допомогою конфігурація 2.

Після того як було обрано конфігурацію списку для тестування продуктивності, було проведено замір властивостей з кожною модифікацію яка впливає на перфоменс окремо.

Результат наведено у таблиці 2.

Таблиця 2 – Заміри характеристик властивостей (таблиця виконана самостійно)

Характеристики / Властивості	maxBlankArea(px)	cumulativeBlankArea(px)	Loading time (ms)
1	2	3	4
Без покращень	20362	29034	25

Кінець табл.2

1	2	3	4
isDumpElement	15590	19322	24
withKeyExtractor	16053	35800	27
withStaticSize	15800	16424	21
withoutStateInside Elements	19800	27352	25
withMemo	16782	17613	23
isCachingImage	16339	19192	23

Створимо графіки порівняння maxBlankArea на базі таблиці для більш зручного та детального порівняння. На рисунку 7 можна побачити наглядне порівняння де абсциса x показує властивості списку та абсциса y – кількість пікселей.

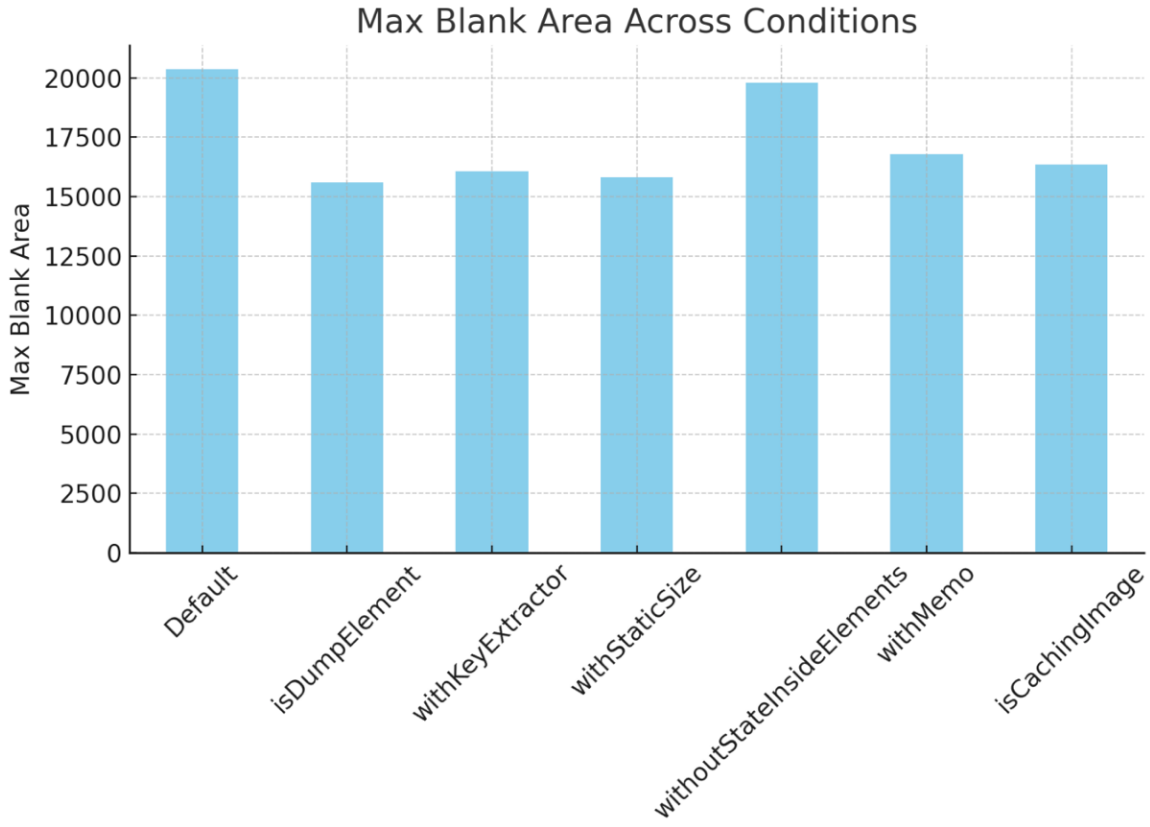

Рисунок 7 – Порівняння maxBlankArea (рисунок виконаний самостійно)

Можна побачити що усі властивості мають позитивний вплив на дану характеристику у порівнянні з Default. Тільки характеристика withoutStateInsideElements має незначне покращення.

Створимо графіки порівняння blankAreaConditions на базі таблиці для більш зручного та детального порівняння. На рисунку 8 можна побачити наглядне порівняння де абсциса x показує властивості списку та абсциса y – кількість пікселей.

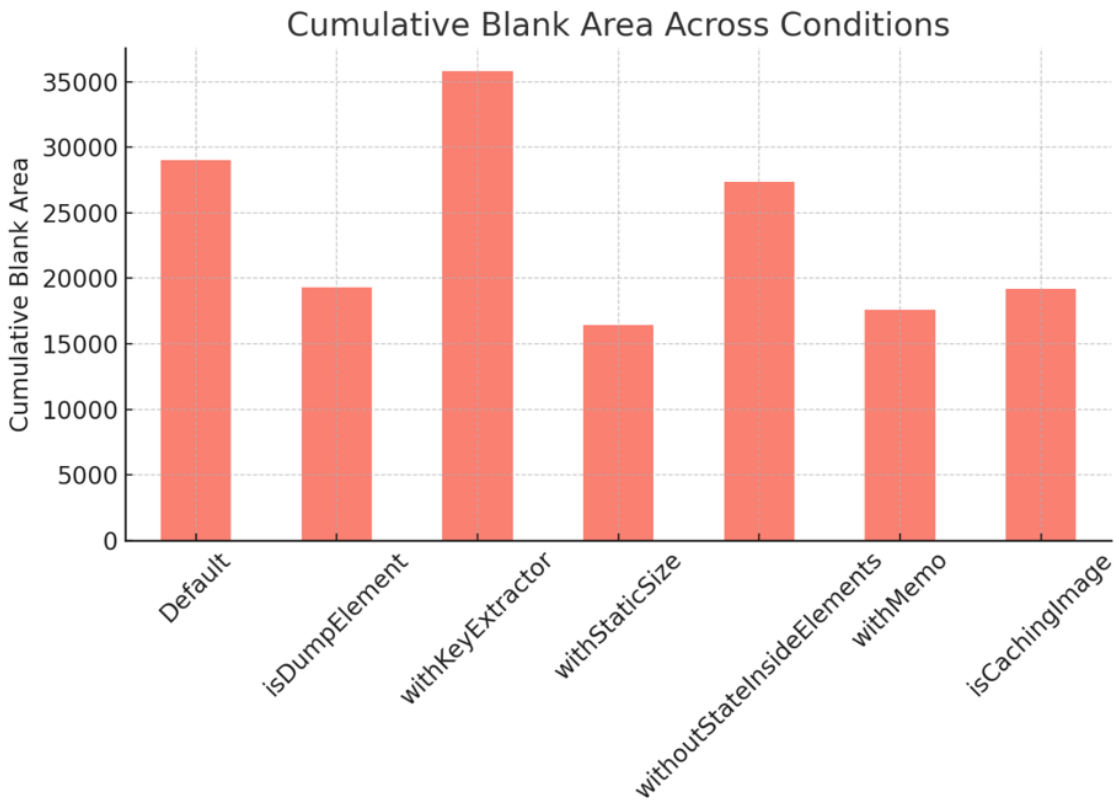

Рисунок 8 - Порівняння blankAreaConditions (рисунок виконаний самостійно)

На рисунку 8 видно, що усі властивості крім withKeyExtractor мають позитивний вплив на дану характеристику у порівнянні з Default. Слід виділити властивості isDumpElement та withStaticSize які мають найбільший вплив. Також слід звернути увагу на withKeyExtractor який зменшує продуктивність.

Створимо графіки порівняння loadingTime на базі таблиці для більш зручного та детального порівняння. На рисунку 9 можна побачити наглядне порівняння де абсциса x показує властивості списку та абсциса y – час у мс.

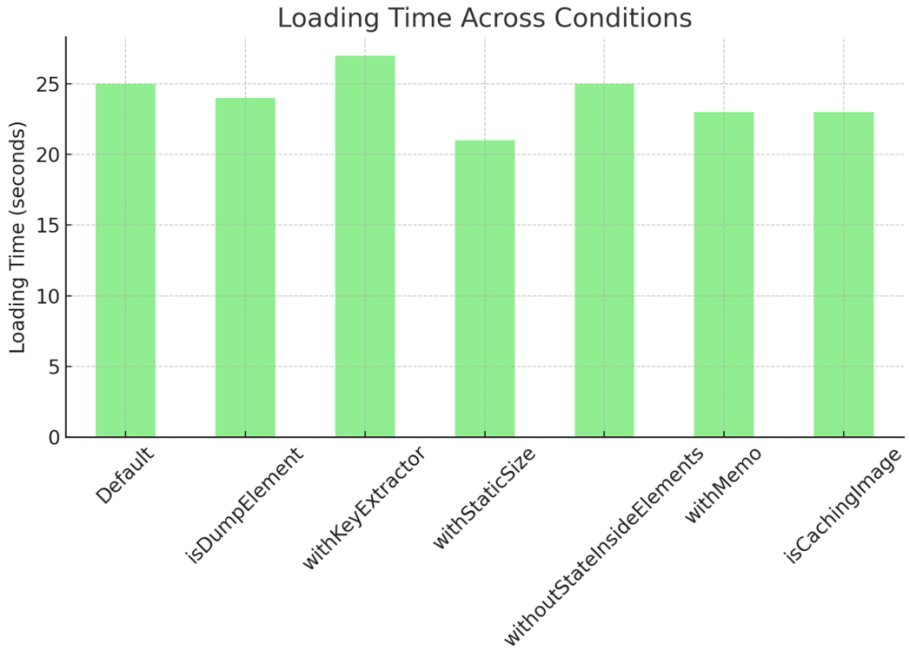

Рисунок 9 - Порівняння Loading time (рисунок виконаний самостійно)

Розглянемо, як кожна модифікація впливає на maxBlankArea, cumulativeBlankArea та Loading time у порівнянні з Default конфігурацією.

– Default - базове значення без модифікацій;
– isDumpElement - зменшення maxBlankArea і cumulativeBlankArea. Незначне зменшення Loading time. Що свідчить про те, що компоненти повинні бути більш простими та потрібно робити усі додаткові операції поза елементом;
– withKeyExtractor - зменшення maxBlankArea, але значне збільшення cumulativeBlankArea. Незначне збільшення Loading time. Це може свідчити про те що краще не використовувати Key Extractor якщо не потрібно оброблювати ключі елементів в кастомний спосіб, бо дефолтна робота keyExtractor є вже доволі оптимізованою;

- withStaticSize - значне зменшення maxBlankArea і cumulativeBlankArea. Найшвидший Loading time серед всіх модифікацій. Рекомендуються для використання;
- withoutStateInsideElements - незначне зменшення maxBlankArea і cumulativeBlankArea. Loading time залишається без змін. Дуже схожа за змістом з isDumpElement але різниця полягає у тому, що в даній властивості зберігаються значення та оброблюються всередині елементу;
- withMemo - зменшення maxBlankArea і значне зменшення cumulativeBlankArea. Loading time зменшується. Для досягнення найкращих результатів рекомендується помістити компоненти елементів та функції рендеру в cache системи.
- isCachingImage - зменшення maxBlankArea і cumulativeBlankArea. Loading time зменшується. При використанні розвинутого обробникам зображень можна побачити, що всі метрики було покращено.

Проаналізувавши кожну властивість, можна побачити що загалом усі властивості мають покращення, тому було вирішено вирахувати середнє покращення кожної властивості за формулою наведеною нижче:

$$P_c = \frac{v_d - v_c}{v_d} * 100\%$$

де V_d – це Default значення при увімкненої властивості (наприклад isDumpElement);

V_c – це значення Current властивості;

P_c – це процент покращення Current властивості відносно Default.

Наведемо таблицю у зміни значень у процентному еквіваленті (див. табл.3).

Будемо вважати що maxBlankArea, cumulativeBlankArea та Loading time мають однаковий вплив на систему, тобто коефіцієнти важливості будуть однакові.

Таблиця 3 – Таблиця порівнянь значень у процентному еквіваленті (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
isDumpElement	23.43%	33.45%	4%
withKeyExtractor	21.16%	-23.3%	-8%
withStaticSize	22.4%	43.43%	16%
withoutStateInsideElements	2.76%	5.79%	0%
withMemo	17.58%	39.34%	8%
isCachingImage	19.76%	33.9%	8%

Для більш наочного порівняння було вирішено прорахувати середній відсоток зміни за формулою наведеною нижче.

$$R = \frac{P_1 * \ldots * P_k}{k}$$

де P_k – це процентне значення зміни перфомансу;

K– це кількість властивостей;

R – середній відсоток зміни характеристик для властивості.

Нижче наведено таблиця 4 з середнім відсотком для кожної властивості.

Таблиця 4 - Таблиця порівнянь значень (Average performance change) (таблиця виконана самостійно)

	Average performance change
isDumpElement	20.29%
withKeyExtractor	-3.38
withStaticSize	27.28%
withoutStateInsideElements	2.85%
withMemo	21.64%
isCachingImage	20.55%

Проаналізувавши таблиці було зроблено наступні висновки:

- найбільше покращення продуктивності досягається за допомогою модифікації withStaticSize (27.28%). Це вказує на те, що фіксування розмірів елементів списку є найефективнішим підходом для оптимізації рендерингу;
- модифікації withMemo, isCachingImage та isDumpElement також показують значні покращення продуктивності (понад 20%), що робить їх корисними для оптимізації продуктивності UI;
- модифікація withoutStateInsideElements має незначне покращення продуктивності, що вказує на її обмежений вплив;
- модифікація withKeyExtractor показує зниження продуктивності (-3.38%), що свідчить про те, що її використання може бути контрпродуктивним для рендерингу списків.

Для подальшого аналізу та перевірки продуктивності було створено кілька груп модифікацій, щоб визначити, які комбінації надають найбільше покращення продуктивності. Наведемо групи, які було створено:

Перша комьынація включає в собі наступні покращення списку withStaticSize, withMemo та isCachingImage.

Комбінація модифікацій, що показали найбільше покращення продуктивності окремо. Нижче наведено таблиця 5 із замірами:.

Таблиця 5 – Дані по першій комбінації (таблиця виконана самостійно)

	maxBlankArea(px)	cumulativeBlankArea(px)	Loading time(ms)
Комбінація 1	14579	15130	24

Проаналізувавши таблицю можна побачити, що комбінація модифікацій withStaticSize + withMemo + isCachingImage показує значне покращення продуктивності в контексті зменшення maxBlankArea та cumulativeBlankArea.

Хоча час завантаження трохи збільшився до 24 мс, це незначне збільшення у порівнянні зі значними покращеннями в інших параметрах.

Загалом, ця комбінація модифікацій є ефективною для оптимізації рендерингу списку, зменшуючи незаповнений простір та покращуючи використання простору. У таблиці 6 наведена зміна продуктивності у процентах.

Таблиця 6 - Дані по першій комбінації у процентах (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
Комбінація 1	28.4%	47.8%	4%

Середній відсоток зміни для даної комбінації дорівнює 26.7%. Що свідчить про дуже гарне покращення та рекомендується використовувати дану комбінацію покращень.

Наступна комбінація, яка була розглянуто withStaticSize + isDumpElement. Оскільки ці дві модифікації показали значні покращення, їх комбінація може дати результати порівняно з попередньою комбінацією. Нижче наведено таблиця 7 з даними.

Таблиця 7 - Дані по другій комбінації (таблиця виконана самостійно)

	maxBlankArea(px)	cumulativeBlankArea(px)	Loading time(ms)
Комбінація 2	18243	23431	27

Комбінація withStaticSize + isDumpElement показує значне зменшення maxBlankArea та cumulativeBlankArea, що свідчить про покращення ефективності використання простору.

Однак, час завантаження збільшується у порівнянні з default значенням, що може бути проблемою для додатків, де важлива швидкість завантаження.

Загалом, ця комбінація є ефективною в контексті зменшення порожнього простору, але впливає на час завантаження, що потрібно враховувати при оптимізації.

В таблиці 8 наведено дані по другій комбінації у відсотках.

Таблиця 8 - Дані по другій комбінації у процентах (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
Комбінація 2	10.43%	19.31%	-8%

Середній відсоток зміни для данної комбінації дорівнює 7.24%. Цей результат є позитивним, але загальні зміни продуктивності не є великими.

Наступна комбінація буде включати усі доступні модифікації списку для покращення перфомансу. Нижче у таблиці 9 наведено результат заміру.

Таблиця 9 - Дані по третій комбінації(таблиця виконана самостійно)

	maxBlankArea(px)	cumulativeBlankArea(px)	Loading time(ms)
Комбінація 3	16531	37149	26

Комбінація 3 показує значне зменшення maxBlankArea, що свідчить про покращення ефективності використання простору.

Однак, cumulativeBlankArea значно збільшується, що може вказувати на більше загальне навантаження на інтерфейс користувача. Дана характеристика може бути пов’язана з властивістю withKeyExtractor.

Час завантаження незначно збільшується, що не є критичним, але може впливати на сприйняття швидкості роботи додатку.

Загалом, ця комбінація може бути корисною для зменшення незаповненого простору, але потрібно враховувати збільшення загального навантаження на UI thread.Нижче наведено таблицю 10 з змінною у відсотках.

Таблиця 10 - Дані по третій комбінації у процентах (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
Комбінація 3	18.83%	-27.97%	-4%

Середній відсоток зміни комбінації дорівнює -4.38%. Що свідчить про погіршення перфомансу списка. Так як потенційною проблемою даної комбінації є властивість withKeyExtractor, було вирішено зробити комбінацію з всіма властивостями, крім withKeyExtractor.

Остання комбінація включає усі покращення крім withKeyExtractor характеристики. Нижче у таблиці 11 наведено дані.

Таблиця 11 - Дані по четвертій комбінації (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
Усі доступні дані крім withKeyExtractor	14457	17897	24

Комбінація 4 показує значне покращення продуктивності за всіма параметрами:

– maxBlankArea зменшується на 14457, що вказує на меншу кількість незаповненого простору.
– cumulativeBlankArea зменшується на 17897, що вказує на ефективніше використання простору.

Загалом, ця комбінація є дуже ефективною для оптимізації продуктивності рендерингу списку, зменшуючи незаповнений простір, покращуючи використання простору та зменшуючи час завантаження.

Нижче наведено таблицю 12 з змінною у відсотках.

Таблиця 12 - Дані по четвертій комбінації у процентах (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
Усі доступні дані крім withKeyExtractor	29.01%	38.36%	4%

Середній відсоток зміни комбінації дорівнює 23.79%. Що є дуже гарним показником та було виявлено, що withKeyExtractor погіршую продуктивність списку.

Порівняємо усі комбінації та наведемо графіки для кожної характеристики.

На рисунок 10 зображено графік для максимальної порожньої області (maxBlankArea).

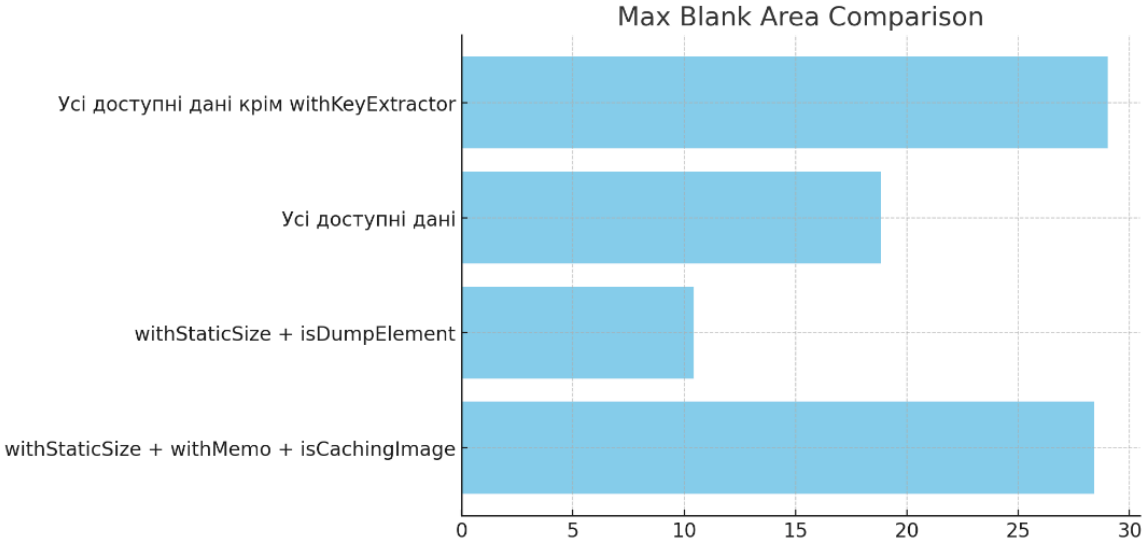

Рисунок 10 – Графік maxBlankArea (рисунок виконаний самостійно)

Конфігурація “Усі доступні дані крім withKeyExtractor” має найбільшу максимальну порожню область - 29.01%. “withStaticSize + isDumpElement” має найменшу максимальну порожню область - 10.43%. Різниця між “Усі доступні дані крім withKeyExtractor” та “withStaticSize + withMemo + isCachingImage” мінімальна.

Проаналізуємо сумарну порожню область (cumulativeBlankArea). Дивитися рисунок 11.

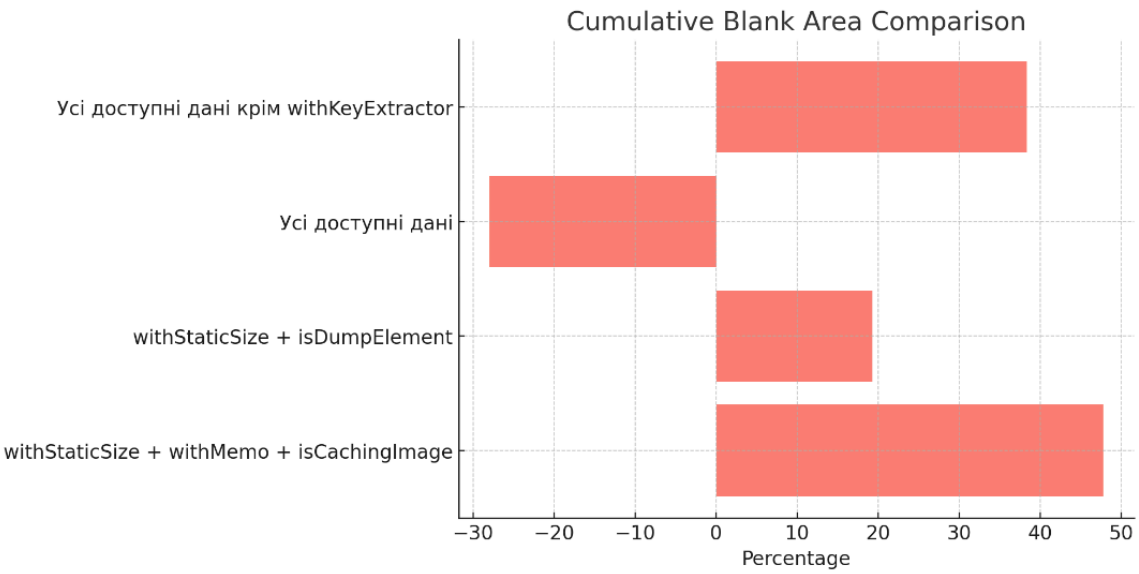

Рисунок 11 – Графік cumulativeBlankArea (рисунок виконаний самостійно)

“withStaticSize + withMemo + isCachingImage” має найбільшу сумарну порожню область - 47.8%. Конфігурація “Усі доступні дані” має негативну сумарну порожню область, що може свідчити про певну проблему з оптимізацією.

Зробимо порівняння часу завантаження. Дивитися рисунок 12.

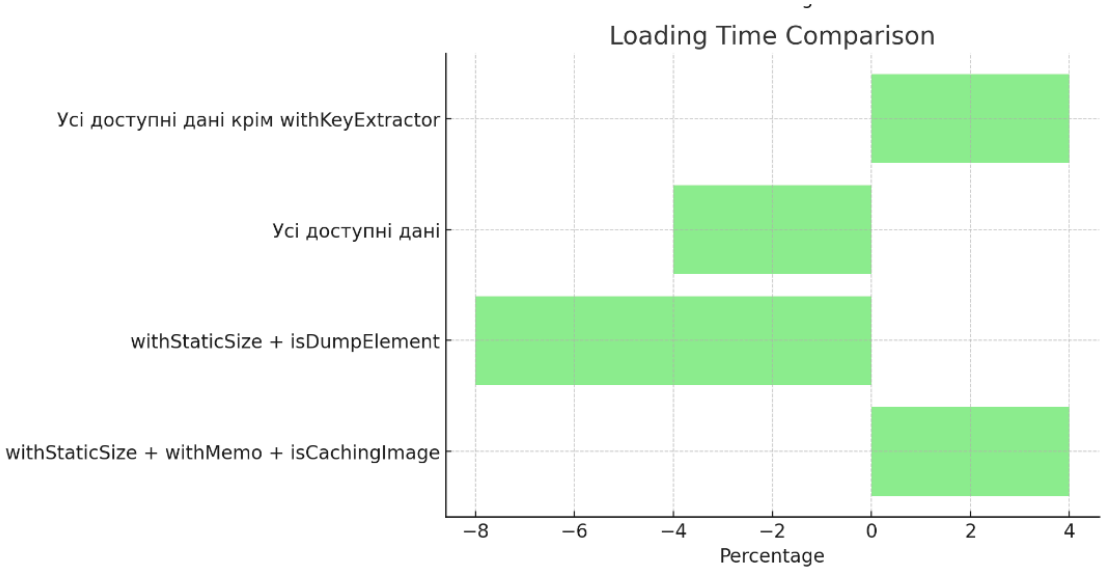

Рисунок 12 – Графік Loading Time (рисунок виконаний самостійно)

Як “withStaticSize + withMemo + isCachingImage”, так і “Усі доступні дані крім withKeyExtractor” мають однаковий час завантаження - 4%. “withStaticSize +

isDumpElement” показує найгірший час завантаження - -8%. “Усі доступні дані” також має негативний час завантаження - -4%.

Проаналізувавши усі комбінації у порівнні один з одним було зроблено наступні висновки:

– конфігурація “withStaticSize + withMemo + isCachingImage” загалом показує хороші результати з високою максимальною та сумарною порожньою областю, але з прийнятним часом завантаження.
– “withStaticSize + isDumpElement” має найменшу максимальну та сумарну порожню області, але страждає від поганого часу завантаження.
– “Усі доступні дані” показує аномалію з негативною сумарною порожньою областю та поганим часом завантаження, що може вказувати на потенційні проблеми. Як показують інші конфігурації, потенційна проблема даної конфігурації полягає у властивості withKeyExtractor.
– “Усі доступні дані крім withKeyExtractor” є збалансованою конфігурацією з трохи більшою максимальною порожньою областю, але зі значним покращенням сумарної порожньої області та прийнятним часом завантаження.

Для повної картини було зроблено аналіз аналогів покращення роботи списку.Нижче наведено таблицю 13 з замірами покращень для вже існуючих реалізацій.

Таблиця 13 – Абсолютні значення даних для вже існуючих реалізацій (таблиця виконана самостійно)

	maxBlankArea (px)	cumulativeBlankArea(px)	Loading time(ms)
FlashList	16731	27429	24
React Native Largelist	18432	26632	27
RecyclerListView	17834	26231	26

Вже існуючі реалізації роблять значне покращення відносто FlatList.

Нижче наведено таблицю 14 з замірами покращень для вже існуючих реалізацій.

Таблиця 14 - Дані для вже існуючих реалізацій (таблиця виконана самостійно)

	maxBlankArea	cumulativeBlankArea	Loading time
FlashList	17.8%	5.5%	4%
React Native Largelist	9.4%	8.2%	-8%
RecyclerListView	12.4%	9.6%	-4%

Проаналізувавши дані реалізації було зроблено висновок, що FlashList має найкращі результати у порівнянні з іншими реалізаціями.

Середній відсоток зміни списку при використанні FlashList дорівнює 9.1%. Що є дуже гарним показником. У порівнянні з нашими комбінаціями покращень можна виявити, що дана реалізація гірше приблизно в 2.5 рази. Рекомендується застосовувати дану реалізацію у зв’язці з нашими рекомендаціями.

Для більш детального аналізу зроблених покращень було проаналізовано додаткові заміри перфомансу FlatList для різного обьему даних. Для цього було створено три групи замірів початкового стану списку та покращеного де перша група має кількість елементів у розмірі 100, друга група – 1000 має звичайну кількість, яке і використовувалось впродовж даної роботи, та третя група – 10000 відповідно.

Нижче наведено таблицю 15 з замірами для різних списк з різною кількістю елементів.

Таблиця 15 - Дані з замірами для списків з різною кількістю елементів (таблиця виконана самостійно)

	maxBlankArea (px)	cumulativeBlankArea (px)	Loading time(ms)
Без покращень (100)	1023	2243	23
Комбінація (100)	426	694	23
Без покращень (1000)	20362	29034	25
Комбінація (1000)	14579	15130	24
Без покращень (10000)	219185	373017	30
Комбінація (10000)	208529	247142	27

Аналіз перфомансу включав заміри трьох груп списків із різною кількістю елементів: 100, 1000 та 10000. Результати показали, що застосування оптимізаційних підходів значно зменшує максимальну та кумулятивну площу пустих ділянок у списках, а також покращує час завантаження списків, особливо для великих обсягів даних.

Для списків із 100 елементами застосування оптимізацій дозволило зменшити максимальну площу пустих ділянок з 1023 до 426 пікселів, а кумулятивну площу з 2243 до 694 пікселів.

Для списків із 1000 елементами максимальна площа пустих ділянок зменшилась з 20362 до 14579 пікселів, а кумулятивна площа з 29034 до 15130 пікселів. Час завантаження списку також покращився з 25 до 24 мілісекунд.

Для найбільшого списку із 10000 елементами максимальна площа пустих ділянок зменшилась з 219185 до 208529 пікселів, а кумулятивна площа з 373017 до 247142 пікселів. Час завантаження списку знизився з 30 до 27 мілісекунд.

Дана оптимізація є рекомендованою для застосування незважаючи на кількість елементів списку.

ВИСНОВКИ

У цій роботі було проаналізовано ефективність різних підходів до оптимізації компонента FlatList у React Native. Розглянуто основні конфігурації та модифікації, які можуть впливати на продуктивність списків у мобільних додатках. Проведені дослідження охопили тестування трьох основних конфігурацій списків та декількох модифікацій для покращення їх продуктивності.

Можна виділити декілька модифікацій для покращення продуктивності, такі як використання властивості withStaticSize. Дана властивість показало найкращі результати (покращення на 27.28%), що вказує на ефективність фіксації розмірів елементів для оптимізації рендерингу. Властивість мемоізації (withMemo) та кешування зображень (isCachingImage) також показали значні покращення (понад 20%), що робить їх корисними для підвищення продуктивності UI. Також слід звернути увагу на властивість withKeyExtractor, яка показала зниження продуктивності (-3.38%), що свідчить про її негативний вплив на рендеринг списків.

Також було проаналізовано декілька комбінацій властивостей, та було виявлено, що комбінація “withStaticSize + withMemo + isCachingImage” властивостей має найкращий показник покращень середнього розміру списку, що складає 26.7% для середнього розміру списку.

Запропонований підхід та результати дослідження можуть бути застосовані для оптимізації спискових компонентів у мобільних додатках, розроблених за допомогою React Native. Це дозволить розробникам покращити продуктивність додатків, забезпечити плавність прокручування списків та зменшити навантаження на систему, що в свою чергу покращить користувацький досвід.

Результати роботи були апробовані та підтверджені через створення тестового мобільного додатка, що дозволяє комплексно досліджувати продуктивність FlatList у різних конфігураціях та умовах.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Mobile apps: How to evaluate mobile app design? [Електронний ресурс]. URL: https://merge.rocks/blog/how-to-evaluate-a-mobile-app-design (дата звернення: 6.04.2024).

2. P. Campamella, Future Mobile Learning, 2021 19th International Conference on Emerging eLearning Technologies and Applications (ICETA), Košice, Slovakia, 2021.

3. Ruben Horn, Abdellah Lahnaoni, Vadim Isakov, Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts, 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Melbourne, Australia, 2023.

4. Matt Neuberg, iOS 15 Programming Fundamentals with Swift: Swift, Xcode, and Cocoa Basics. 1st Ed., 2021 – 700 с.

5. Pierre-Olivier Laurence, Amanda Hinchman-Dominguez, Programming Android with Kotlin. Achieving Structured Concurrency with Coroutines. 1st Ed., 2021 – 336 с.

6. Ден Гермес, Німа Мазлумі, Building Xamarin.Forms Mobile Apps Using XAML: Mobile Cross-Platform XAML and Xamarin.Forms Fundamentals, 2019 – 426 с.

7. React Native [Електронний ресурс]. URL: https://reactnative.dev/ (дата звернення: 17.03.2024).

8. Roy Derks, React Projects: Build advanced cross-platform projects with React and React Native to become a professional developer, Packt Publishing, 2022.

9. Shpetim Kadrija; Agon Memeti, Development of mobile app through React Native hybrid framework, 2022 11th Mediterranean Conference on Embedded Computing (MECO), 2022.

10. Subrota Kumar Mondal, Yu Pei, Hong Ning Dai, Boosting UI Rendering in Android Applications. 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), Macau, China, 2020.

11. Jay Nanavati, Sanskruti Patel, Unnati Patel, Atul Patel, Critical Review and Fine-Tuning Performance of Flutter Applications, 2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI), 2024.

12. Flutter [Електронний ресурс]. URL: https://flutter.dev/ (дата звернення: 21.03.2024).

13. Daria Pronina, Iryna Kyrychenko. Comparison of Redux and React HooksMethods in Terms of Performance. 6th International Conference onComputational Linguistics and Intelligent Systems (COLINS-2022), 2022, Gliwice, Poland. – CEUR Workshop Proceedings 3171, Volume I: Main, PP. 791 – 800.

14. FlatList [Електронний ресурс]. URL: https://reactnative.dev/docs/flatlist (дата звернення: 21.03.2024).

15. SectionList [Електронний ресурс]. URL: https://reactnative.dev/docs/sectionlist (дата звернення: 24.03.2024).

16. VirtualizedList [Електронний ресурс]. URL: https://reactnative.dev/docs/virtualizedlist (дата звернення: 24.03.2024).

17. Optimizing FlatList [Електронний ресурс]. URL: https://reactnative.dev/docs/optimizing-flatlist-configuration (дата звернення: 24.03.2024).

18. RecyclerListView [Електронний ресурс]. URL: https://www.npmjs.com/package/recyclerlistview (дата звернення: 6.04.2024).

19. FlashList [Електронний ресурс]. URL: https://www.npmjs.com/package/@shopify/flash-list (дата звернення: 6.04.2024).

20. react-native-largelist [Електронний ресурс]. URL: https://www.npmjs.com/package/react-native-largelist (дата звернення: 6.04.2024).

21. Петроченков П. М. Дослідження методів оптимізації та прискорення рендера елементів FlatList у мобільному додатку. 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 6., – Харків: ХНУРЕ. 2024. – с.459-461. https://doi.org/10.30837/IYF.IIS.2024.459

22. Optimizing FlatList Performance in React Native [Електронний ресурс]. URL: https://medium.com/@ajkhatibi/optimizing-flatlist-performance-in-react-native-a83b1315ded9 (дата звернення: 6.04.2024).

23. Instant Performance Upgrade: From FlatList to FlashList [Електронний ресурс]. URL: https://shopify.engineering/instant-performance-upgrade-flatlist-flashlist (дата звернення: 6.04.2024).

24. MMKV [Електронний ресурс]. URL: https://github.com/mrousavy/react-native-mmkv (дата звернення: 6.04.2024).

25. MMKV vs AsyncStorage in React Native [Електронний ресурс]. URL: https://reactnativeexpert.com/blog/mmkv-vs-asyncstorage-in-react-native (дата звернення: 6.04.2024).

26. Xianfeng Li, Gengchao Li, Xiaole Cui, ReTriple: Reduction of Redundant Rendering on Android Devices for Performance and Energy Optimizations. 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, San Francisco, CA, USA

27. Daeun Heo, Daejin Park, Asynchronous Interaction Framework for Verilog Simulation Virtualization on Node.js. 2021 International Conference on Electronics, Information, and Communication (ICEIC), 2021, Jeju, Korea (South).

28. React Native View [Електронний ресурс]. URL: https://reactnative.dev/docs/view (дата звернення: 6.04.2024).

29. Smelyakov, K., Datsenko, A., Skrypka, V., Akhundov, A., The efficiency of images reduction algorithms with small-sized and linear details
Smelyakov, K., Datsenko, A., Skrypka, V., Akhundov, A.
2019 IEEE International Scientific-Practical Conference: Problems of Infocommunications Science and Technology, PIC S and T 2019 - Proceedings, 2019, стр. 745–750, 9061250 DOI 10.1109/PICST47496.2019.9061250

30. JeongJoon Yoo, Hybrid Curve Rendering Scheme for Mobiles, 2020 IEEE International Conference on Consumer Electronics (ICCE), 2020, Las Vegas, NV, USA.

31. Tereshchenko, I., Tereshchenko, A., Bilous, N., Shtangey, S., Warsza, Z.L. Risk Analysis Method by the Extreme Data of Dependent Exogenous Variables Tereshchenko, I., Tereshchenko, A., Bilous, N., Shtangey, S., Warsza, Z.L.

Journal of Automation, Mobile Robotics and Intelligent Systems, 2021, 2021(3), pp. 44–53

32. Moqups [Електронний ресурс]. URL: https://moqups.com/ (дата звернення: 6.04.2024).

33. React Navigation [Електронний ресурс]. URL: https://reactnavigation.org/ (дата звернення: 6.04.2024).