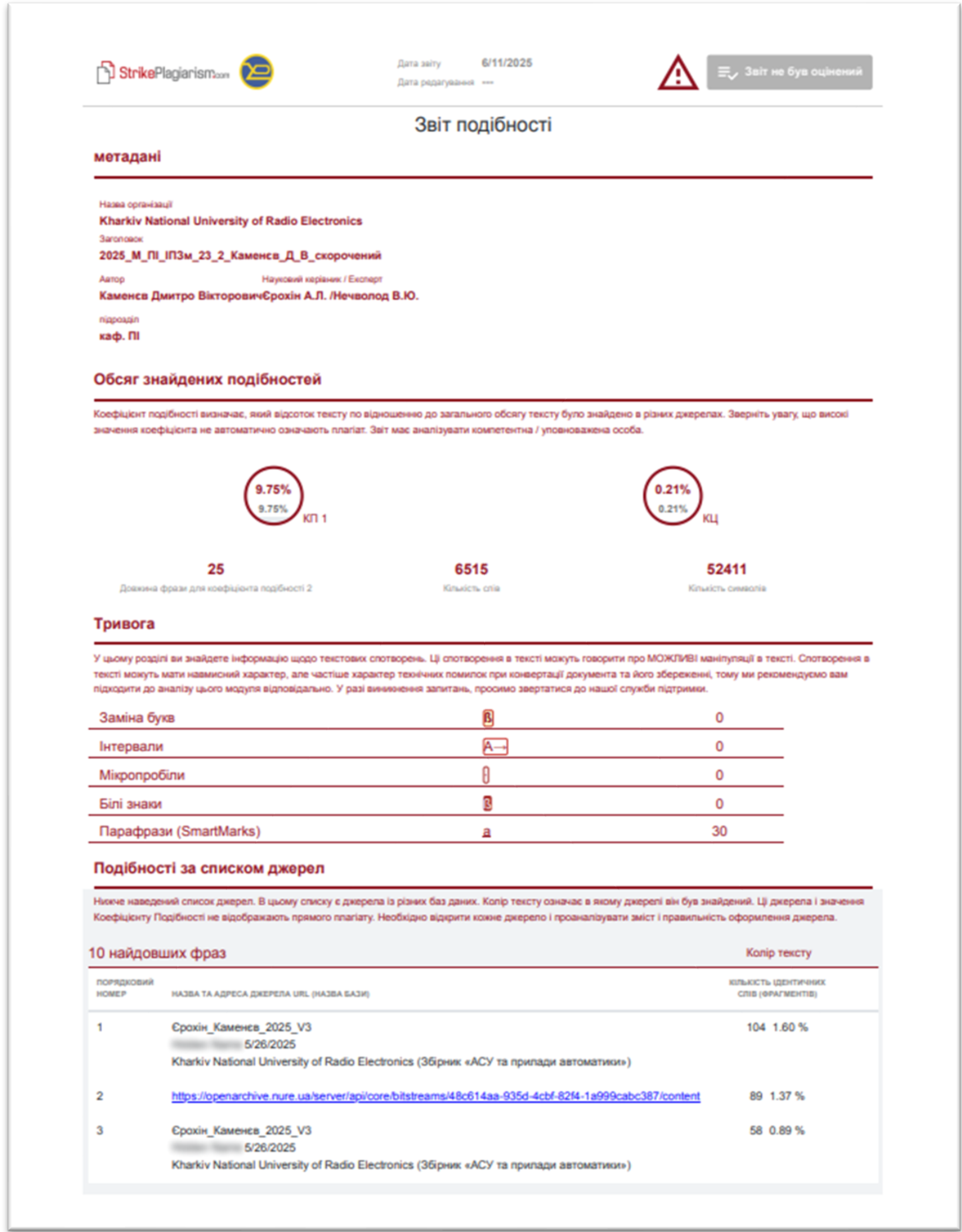
ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



StrikePlagiarism.com

Дата звіту 6/11/2025
Дата редагування ---

Звіт не був оцінений

Звіт подібності

метадані

Назва організації
Kharkiv National University of Radio Electronics

Заголовок
2025_М_ПІ_ІПЗм_23_2_Каменєв_Д_В_скорочений

Автор
Каменєв Дмитро Вікторович

Науковий керівник / Експерт
Єрохін А.Л. /Нечволод В.Ю.

підрозділ
каф. ПІ

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

9.75% 9.75% КП 1

0.21% 0.21% КЦ

25	6515	52411
Довжина фрази для коефіцієнта подібності 2	Кількість слів	Кількість символів

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	0
Інтервали	A→	0
Мікропробіли	0	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	30

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз — Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	104 1.60 %
2	https://openarchive.nure.ua/server/api/core/bitstreams/48c614aa-935d-4cbf-82f4-1a999cabc387/content	89 1.37 %
3	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	58 0.89 %

4	https://openarchive.nure.ua/server/api/core/bitstreams/48c614aa-935d-4cbf-82f4-1a999cabc387/content	53 0.81 %
5	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	46 0.71 %
6	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	39 0.60 %
7	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	39 0.60 %
8	https://openarchive.nure.ua/server/api/core/bitstreams/b02d9175-8a06-46d5-901e-c93d335ba777/content	34 0.52 %
9	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	22 0.34 %
10	Єрохін_Каменєв_2025_V3 5/26/2025 Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)	20 0.31 %

з бази даних RefBooks (0.00 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)

з домашньої бази даних (6.20 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Єрохін_Каменєв_2025_V3 5/26/2025 **Kharkiv National University of Radio Electronics (Збірник «АСУ та прилади автоматики»)**	**404 (14) 6.20 %**

з програми обміну базами даних (0.15 %)

ПОРЯДКОВИЙ НОМЕР	ЗАГОЛОВОК	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	Paper - Eugene Eremeev.docx 3/26/2025 National University "Zaporizhzhia Polytechnic" (Редакція "Радіоелектроніка, інформатика, управління")	10 (1) 0.15 %

з Інтернету (3.39 %)

ПОРЯДКОВИЙ НОМЕР	ДЖЕРЕЛО URL	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)
1	**https://openarchive.nure.ua/server/api/core/bitstreams/48c614aa-935d-4cbf-82f4-1a999cabc387/content**	**154 (3) 2.36 %**
2	**https://openarchive.nure.ua/server/api/core/bitstreams/b02d9175-8a06-46d5-901e-c93d335ba777/content**	**34 (1) 0.52 %**
3	https://www.answeroverflow.com/m/1129430705974296576	21 (2) 0.32 %
4	https://openarchive.nure.ua/bitstreams/087eb047-4bc5-479f-8c9e-f840b14a76c2/download	12 (1) 0.18 %

Список прийнятих фрагментів (немає прийнятих фрагментів)

ДОДАТОК Б

Слайди презентації

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Оптимізація повторного рендерингу у вебзастосунках: аналіз проблеми та рішення на основі React

Каменєв Дмитро Вікторович, ІПЗм-23-2
Науковий керівник: д.т.н, професор Єрохін Андрій Леонідович

16 червня 2025

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Мета:

- Дослідити проблеми ререндерингу в React-додатках.
- Розробити програмне рішення для зменшення непотрібних рендерів компонентів.
- Підвищити продуктивність React-додатків.

Об'єкт дослідження:

- Процес ререндерингу компонентів у React-додатках.

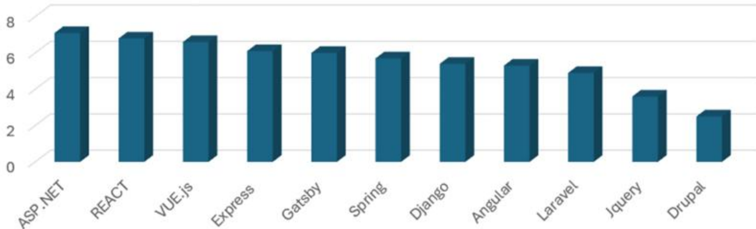
Предмет дослідження:

- Методи та інструменти оптимізації процесу рендерингу.

Для досягнення поставленої мети було визначено наступні задачі:

- ❑ Аналіз причин:
  Проаналізувати основні причини виникнення надмірного ререндерингу в React-додатках.
- ❑ Розробка утиліт:
  Розробити набір програмних утиліт для автоматичної оптимізації ререндерингу.
- ❑ Тестування рішення:
  Протестувати запропоноване рішення (PQM) на модельних прикладах.
- ❑ Оцінка ефективності:
  Оцінити ефективність розробленного рішення шляхом порівняння продуктивності додатків з її використанням та без нього.

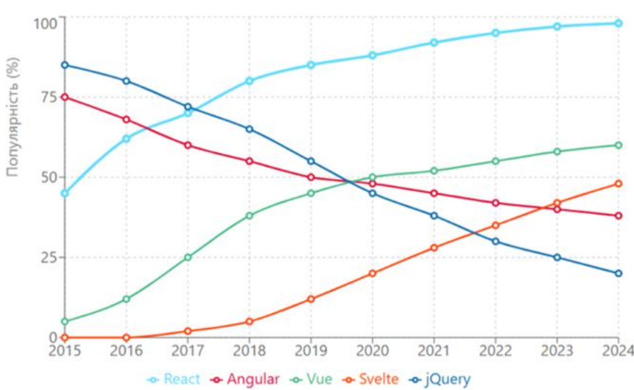
Дослідження № 1



Популярність серед фреймворків для веб розробки (2025)

Дані узято з сервісу StackOverflow.

Дослідження № 2



Дані узято з Google Developer Console.

5

Об'єкт дослідження

Було проаналізовано:

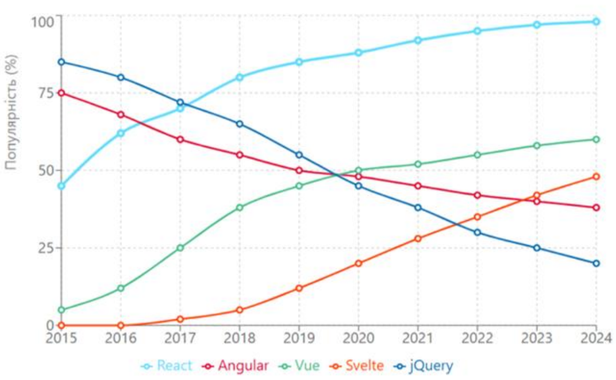
- Офіціальна документація бібліотеки React
- Наукові статті:
  - Comparative Analysis of Angular, React and Vue in SPA development
  - Modern Web Frameworks: Comparison of Rendering Performance
  - Server-Side Rendering vs Static Site Generation
- Статі на відомих форумах для розробників:
  - Стаття на “Medium”
  - Стаття на “GeeksForGeeks”
  - Стаття на “Dev.to”

Основні допоміжні методи (хуки) для оптимізації коду написанного на React:

- UseEffect
- UseState
- UseCallback
- UseMemo
- UseContext
- React.memo

6

Дослідження № 2



Дані узято з Google Developer Console.

5

Об'єкт дослідження

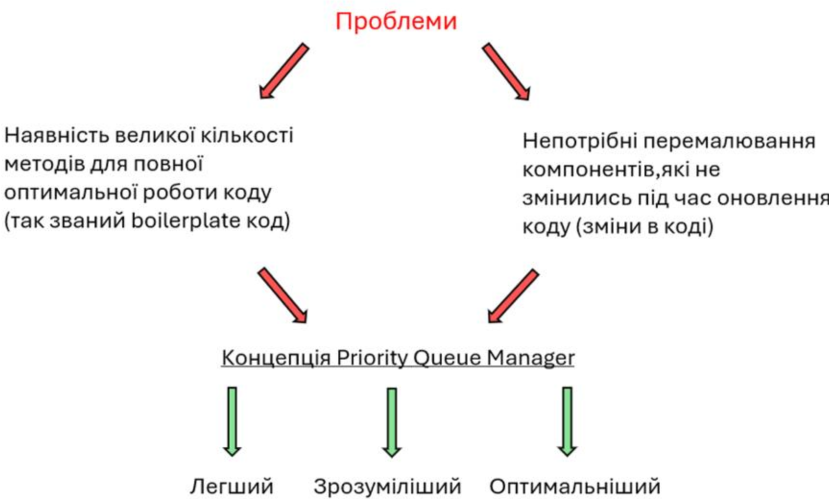
Було проаналізовано:

- Офіціальна документація бібліотеки React
- Наукові статті:
  - Comparative Analysis of Angular, React and Vue in SPA development
  - Modern Web Frameworks: Comparison of Rendering Performance
  - Server-Side Rendering vs Static Site Generation
- Статі на відомих форумах для розробників:
  - Стаття на "Medium"
  - Стаття на "GeeksForGeeks"
  - Стаття на "Dev.to"

Основні допоміжні методи (хуки) для оптимізації коду написанного на React:

- UseEffect
- UseState
- UseCallback
- UseMemo
- UseContext
- React.memo

6



7

Методологія пріорітезації

Формула виглядає наступним чином: $P(c) = \alpha * V(c) + \beta * I(c) + \gamma * D(c)$

де $V(c)$ – функція видимості,

$I(c)$ – індекс важливості,

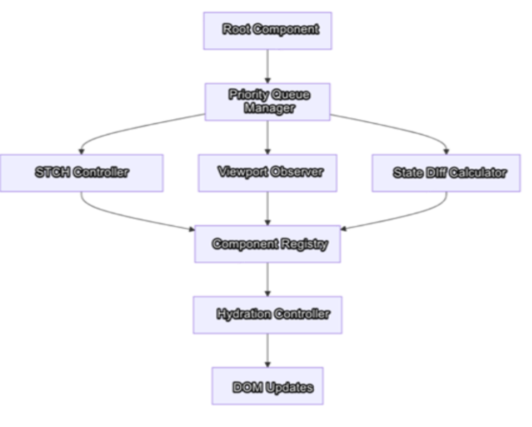
$D(c)$ – глибина в дереві,

α, β, γ – вагові коефіцієнти.

Формула дозволяє забезпечити оптимальну ефективність рендерингу в умовах обмежених ресурсів і мінімізувати затримки в оновленні інтерфейсу для користувач.

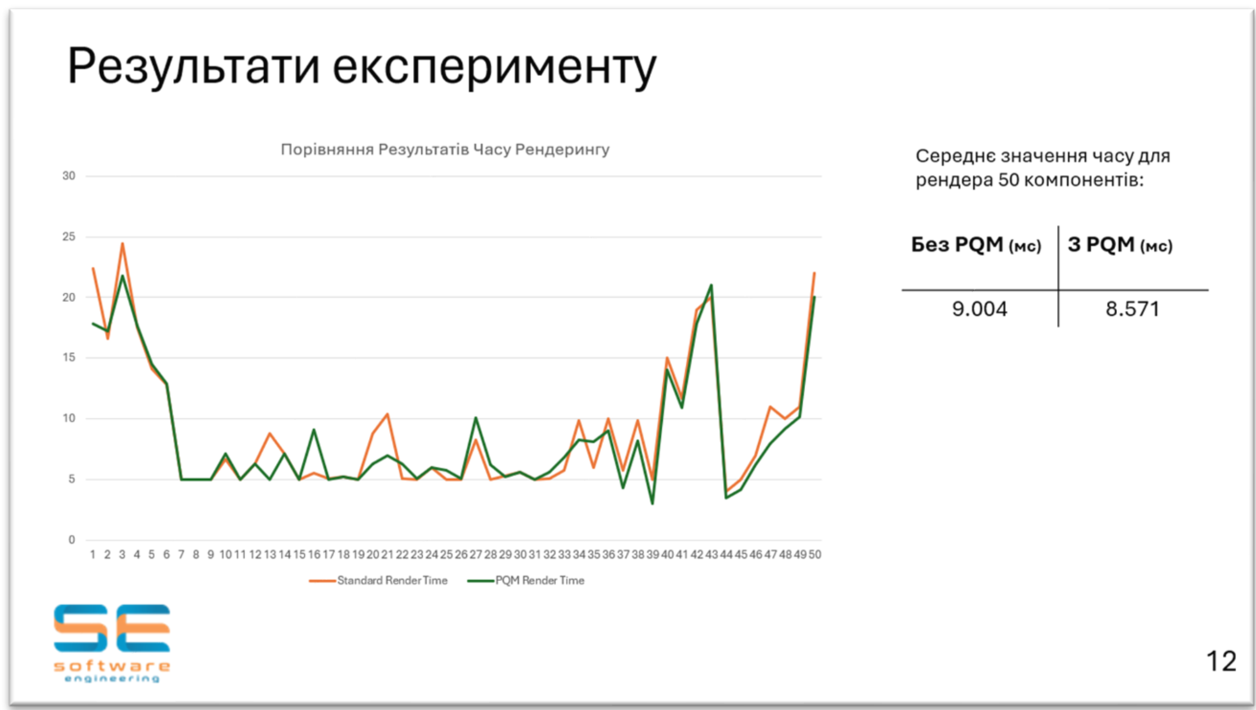
8

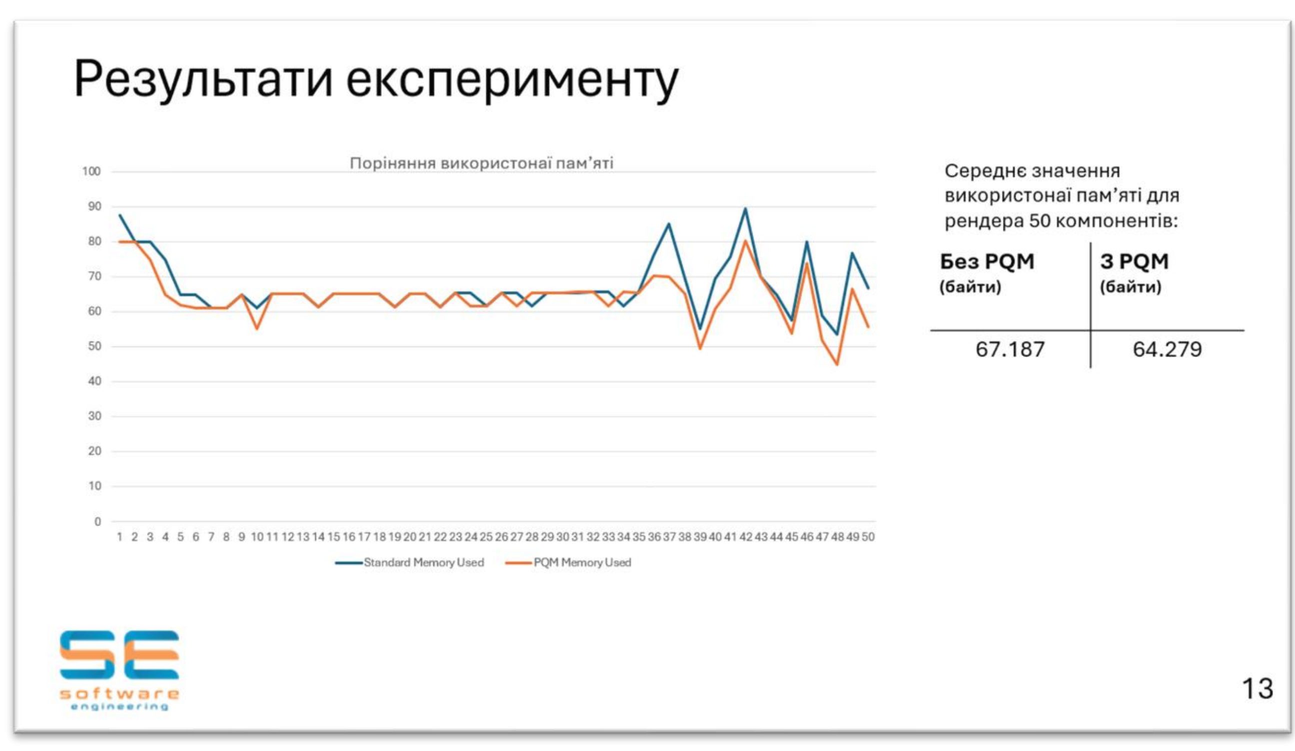
Архітектура система для проведення експериментального дослідження



Опис програмного забезпечення, що було використано у дослідженні

- Було створено тестовий додаток на 50 компонентів
- Компоненти:
  - ❑ Картинки
  - ❑ Текст
  - ❑ SVG іконки
- Відповідне навантаження на додаток під час перемальовування компонентів


Результати експерименту
Порівняння Результатів Часу Рендерингу
Standard Render Time
PQM Render Time
Середнє значення часу для рендера 50 компонентів:
Без PQM (мс) | З PQM (мс)
9.004 | 8.571
software engineering
12


Результати експерименту
Порівняння використонаї пам'яті
Standard Memory Used
PQM Memory Used
Середнє значення використонаї пам'яті для рендера 50 компонентів:
Без PQM (байти) | З PQM (байти)
67.187 | 64.279
software engineering
13

Аналіз отриманих результатів

Співставлення з цілями дослідження:
- Метою було зменшення непотрібних перерисовок у React-додатках
- PQM успішно оптимізує рендеринг, враховуючи пріоритети та видимість компонентів

Висновки з отриманих даних:
- PQM зменшив час рендерингу на 5% (Мал. 4.2 порівняно з Мал. 4.1)
- Використання пам'яті знизилось на 4.5%
- Більш зрозуміле використання без великої частини коду

Інтерпретація результатів:
- Пріоритизація рендерингу зменшує навантаження на основний потік
- Використання `IntersectionObserver` ефективно відкладає рендеринг невидимих компонентів
- PQM забезпечує баланс між продуктивністю та плавністю інтерфейсу

Вплив результатів на існуючі теорії та практики:
- Покращує стандартні методи оптимізації
- Відкриває можливості для інтеграції з DevTools для моніторингу продуктивності

14

Публікація результатів

Публікації:

- Стаття опублікована у журналі "АСУ та прилади автоматики" № 184 (2025)
- Опубліковані тези на конференції " Інформаційні системи та технології – MIT@AISm 2025 "

15

Підсумки

На виході маємо:

- Практична цінність
- Потенціал для використання в комерційних проектах (планується тестування на реальному додатку)
- Спрощення розробки
- Підвищення ефективності
- Вирішення проблеми "boilerplate" коду
- Зменшення часу рендерингу на 5% і пам'яті компютера на 4.5% під час рендерингу

Потенціал:

- Інтегрування з React Developer Tools
- Створення Google Chrome Extension
- Створення Visual Studio Code Extension
- Інтегрування рішення в бібліотеку

16

ДОДАТОК В

Стаття у збірнику "АСУ та прилади автоматики"

УДК 004.4’236

А.Л. Єрохін, Д.В. Каменєв

Оптимізація повторного рендерингу у вебзастосунках: аналіз проблемИ та рішення на основі React

Розглянуто основні особливості повторного рендерингу в сучасних Javascript-фреймворках під час оновлення дерева Document Object Model після зміни стану вебзастосунку. Встановлено, що вирішення питань контролю повторного рендерингу є критичним для забезпечення продуктивності React-застосунків, оскільки саме контроль повторного рендерингу допомагає уникнути ситуацій, коли незначні зміни стану викликають каскадне оновлення всього дерева компонентів. Запропоновано використовувати модель оптимізації процесу рендерингу в вебзастосунках на основі визначення пріоритету рендерингу компонентів, що забезпечує мінімізацію повторних рендерів і покращить продуктивність вебзастосунків.

1. Вступ

У сучасному світі веброзробки проблема ефективності та продуктивності застосунків стає все критичнішою. З постійним зростанням складності вебзастосунків та збільшенням вимог користувачів до швидкодії питання оптимізації повторного рендерингу в JavaScript-фреймворках набуває особливої актуальності. React, Vue, Angular та подібні провідні JavaScript-фреймворки використовують різні методології для оновлення дерева Document Object Model (DOM) після зміни стану [1], [2].

Без стратегічної оптимізації ці платформи можуть ініціювати неефективні каскади візуалізації, що значно збільшує накладні витрати на обробку, прискорює розрядження акумулятора на мобільних пристроях і створює відчутну затримку, що безпосередньо ставить під загрозу взаємодію з клієнтами.

На сучасному цифровому ринку, де очікування щодо продуктивності постійно підвищуються, ефективність рендерингу є не лише технічним імперативом, але й критично важливою бізнес-відмінюваністю, яка значно впливає на показники

задоволеності клієнтів, показники утримання користувачів і, зрештою, потенціал отримання прибутку.

Середовище Javascript-розробників є супердинамічним: щороку з'являється новий фреймворк, новий бандлер, новий метафреймворк, новий спосіб керування станом. У [3] наведено комплексний порівняльний аналіз інтерфейсних фреймворків Angular, React та Vue.js у контексті розробки односторінкових вебзастосунків (Single page application, SPA). Досліджено контрольні показники продуктивності, підтримку екосистеми та криву навчання, що дозволяє розробникам і організаціям обґрунтовано обирати найприйнятнішу структуру для конкретних задач, а також встановлено, що фреймворки мають відмінності в архітектурі, можливостях продуктивності та підтримці спільноти. Проте є проблема, яку потрібно вирішувати для усіх фреймворків, – це проблема повторного рендерингу дерева компонентів.

На рис. 1 наведено статистику завантажень різних фреймворків за даними GitHub Stats.

Серед усіх фреймворків і бібліотек можна виділити три основні, а саме: Angular, React, Vue. Для порівняння у табл. 1 наведено характеристики цих трьох фреймворків, оскільки вони є основними технологіями, яким розробники надають перевагу при виборі стеку технологій.

Рис. 1. Статистика завантажень різних фреймворків за даними GitHub Stats.

Таблиця 1

Окремі характеристики фреймворків Angular, React, та Vue

Фреймворк	Angular	React	Vue
Рік появи	2010	2013	2014
Розробник	Google	Facebook	Evan You
Стан ринку України	212 вакансій	1025 вакансій	158 вакансій
Стан світового ринку	91 000 вакансій	165 000 вакансій	5 000 вакансій
Рекомендоване застосування	Великі масштабовані програми в реальному часі	Кросплатформні програми	Легкі та інтуїтивно зрозумілі програми
Основні проблеми та недоліки	Складнощі зі створенням кросплатформних додатків; непридатність для SEO; документація рідко оновлюється; потрібне знання JSX	Документація не на належному рівні; потрібне глибоке знання JSX; часті оновлення призводять до того, що багато інструментів застарівають	Відсутність спільноти; відсутність великомасштабних проєктів, на яких можна вчитися; проблеми із застосуванням в iOS і Safari

Проблема повторного рендерингу є однією з ключових у сучасних вебзастосунках, оскільки надмірні повторні рендери (re-renders) компонентів можуть призводити до значного зростання витрат обчислювальних ресурсів, прискореної розрядки батареї на мобільних пристроях та появи помітних затримок у відображенні інтерфейсу. Це негативно впливає на користувацький досвід, особливо в умовах високих очікувань щодо продуктивності.

Така проблема є актуальною для більшості сучасних JavaScript-фреймворків, таких як React, Vue та Angular, оскільки вони використовують динамічні оновлення DOM після зміни стану.

2. Аналіз стану проблеми, літературних джерел і постановка проблеми дослідження

Невирішеною частиною проблеми повторного рендерингу залишається відсутність універсальних моделей, які б дозволяли більш ефективно управляти пріоритетами повторних рендерів компонентів залежно від їхньої видимості, важливості та вкладеності в структурі застосунку. Це особливо актуально для масштабних вебзастосунків, де надмірні повторні рендери можуть значно погіршити продуктивність. Очікувані вигоди від вирішення цієї проблеми включають підвищення швидкості відгуку інтерфейсу, зменшення витрат ресурсів та покращення користувацького досвіду, що може бути корисним для розробників вебзастосунків, компаній у сфері електронної комерції, соціальних мереж та потокового контенту.

Дослідженням працездатності фреймворків під різними навантаженнями присвячена низка робіт [4]-[7]. Зокрема, у роботі [5], присвяченій також оптимізації рендерингу, досліджується продуктивність рендерингу популярних вебфреймворків з точки зору ефективності оновлення інтерфейсу користувача в реальному часі. Автори порівнюють різні стратегії рендерингу та оптимізації, щоб застосувати ті, які є найефективнішими для обробки великих обсягів даних та складних інтерфейсів.

Оскільки React на сьогодні є інструментом з високим ступенем технічної ефективності, дослідження та вирішення проблеми зайвого повторного рендерингу компонет проводилися саме на його прикладі.

Детальніший розгляд основи рендерингу і проблеми повторного рендерингу бібліотеки React показав, що ключова особливість рендерингу React полягає у використанні віртуального DOM та процесу примирення (reconciliation), які дозволяють оптимізувати оновлення реального DOM, зменшуючи ресурсомісткі операції. Віртуальний DOM – це JavaScript-об’єкт, який є полегшеною копією реального DOM. Реальний DOM, який використовується браузерами для відображення вебсторінок, є складною структурою, і його оновлення може бути повільним через необхідність повторного рендеру та перерахунку простору.

React вирішує цю проблему, створюючи за допомогою JSX віртуальний DOM, який перетворюється на виклики React.createElement. Коли в компонентів змінюється стан (state) або пропси (props), React генерує новий віртуальний DOM, що відображає поточний стан застосунку. Цей об’єкт оновлюється швидко, оскільки не потрібна

взаємодія з браузером. Наприклад, якщо користувач натискає кнопку, що змінює текст, React створює нову версію віртуального DOM, яка відображає оновлений текст.

Примирення – це процес, який дозволяє React ефективно оновлювати реальний DOM, застосовуючи лише необхідні зміни. Він складається з таких етапів:

– створення віртуального DOM: React генерує об'єктну структуру на основі JSX, яка відображає поточний стан компонентів;

– порівняння версій: при кожній зміні стану або пропсів створюється новий віртуальний DOM, який порівнюється з попередньою версією;

– визначення змін: процедура дифінгу аналізує відмінності між двома віртуальними DOM, щоб визначити, які компоненти потрібно оновити;

– оновлення реального DOM: React застосовує лише мінімальні зміни до реального DOM, що зменшує кількість ресурсомістких операцій.

Процес примирення забезпечує швидке оновлення інтерфейсу, навіть якщо в застосунку відбувається багато змін.

Процедура дифінгу є ключовою складовою процесу примирення, оскільки вона визначає, які саме зміни потрібно внести до реального DOM. Вона працює за такими принципами:

– якщо кореневі елементи двох віртуальних DOM мають різні типи (наприклад, <div> змінюється на <span>), то React видаляє старе дерево і створює нове з нуля;

– якщо елементи мають однаковий тип (наприклад, два <div>), то React порівнює їхні атрибути і оновлює лише ті, що змінилися, зберігаючи той самий DOM-елемент.

– для компонентів одного типу React зберігає стан і пропси, оновлюючи лише необхідні частини, що дозволяє уникнути створення нових елементів;

– при порівнянні дочірніх елементів React аналізує їх послідовно. Якщо порядок елементів змінюється (наприклад, додається новий елемент на початок списку), без унікальних ключів (key) React може перебудувати весь список. Ключі дозволяють алгоритму ефективно визначати, які елементи залишилися незмінними.

Наприклад, якщо на початок списку <ul> додається новий елемент <li>, без ключів React може перебудувати всі елементи списку. З ключами React порівнює ці елементи і оновлює лише новий елемент, що значно підвищує ефективність.

React використовує два основні алгоритми примирення:

– стековий алгоритм примирення (до React 16) обробляє оновлення послідовно, що могло призводити до затримок, особливо при багатьох змінах. Наприклад, якщо відбувалося кілька змін стану, проміжні стани могли відображатися на екрані, що знижувало плавність інтерфейсу;

– Fiber-алгоритм примирення (з React 16) – це сучасніший підхід, який дозволяє обробляти оновлення асинхронно. Він розбиває процес оновлення на менші частини (файбери), що дає змогу React пріоритизувати критичні оновлення, обробляти їх паралельно та підтримувати відгук інтерфейсу. Наприклад, Fiber може призупинити оновлення для обробки взаємодії користувача, що забезпечує плавність анімацій і швидку реакцію на дії.

Детальніший розгляд проблеми повторного рендерингу React показав, що не кожний повторний рендеринг React є необхідним, швидше необхідна повторна візуалізація, коли потрібно повторно відтворити компонент, який є джерелом змін у програмі. Повторне відтворення також корисне для тих частин, які безпосередньо використовують будь-яку нову інформацію. Наприклад, ту частину програми, яка керує станом процесу введення користувачем тексту, потрібно оновлювати або повторно відтворювати при кожному натисканні клавіші.

Непотрібні рендеринги поширюються через кілька механізмів повторного рендерингу. Це може спричинятись як помилками кодування React, так і неефективною архітектурою програми. Наприклад, якщо замість частини повторно відображається вся сторінка програми під час кожного натискання клавіші, коли користувач вводить текст у полі введення, то повторна візуалізація сторінки непотрібна.

Проблема надмірного повторного рендерингу не є унікальною для React. У інших фреймворках, наприклад, у Vue, надмірні рендери можуть виникати через реактивну систему, яка автоматично оновлює компоненти при зміні даних, якщо не використовуються оптимізаційні техніки, наприклад, computed properties. У Angular проблема може проявлятися через механізм зон (NgZone) і часте спрацьовування детектора змін, якщо не застосовувати стратегії OnPush. Дослідження [5] показують, що проблема надмірного рендерингу є актуальною не лише для React, але й для інших фреймворків, таких як Vue та Angular, для яких різні автори пропонують використовувати різні стратегії оптимізації. Наприклад, у [8] наголошується на ефективності використання хуків у React, але зазначається, що їх неправильне застосування може призвести до додаткових витрат продуктивності. У [9] розглядається повторний рендеринг дочірньої компоненти без рендерингу сторінки у Vue.js, а в [10] розглянуто стратегії OnPush у Angular для зменшення частоти спрацьовування детектора змін.

Таким чином, хоча в даному дослідженні основний акцент було зроблено на React, проблема є загальною для багатьох фреймворків, і підходи до її вирішення можуть бути адаптовані для інших технологій. Непотрібні повторні рендери не є проблемою, оскільки React досить швидкий, і немає помітної різниці в продуктивності, якщо деякі компоненти повторно рендеряться без потреби. Однак занадто часто непотрібне повторне відтворення вагомих компонентів може призвести до погіршення взаємодії з користувачем. У React є певні рішення для таких непотрібних повторних рендерингів, які вирішують проблему доволі ефективно (UseMemo, UseCallback, React.Memo) та призначені для оптимізації, проте їх використання не завжди приносить користь і може навіть погіршити продуктивність у певних сценаріях. Основні причини цих недоліків такі:

– накладні витрати на перевірку залежностей: наприклад, useMemo і useCallback порівнюють масиви залежностей, щоб визначити, чи потрібно оновити кешоване значення або функцію. Якщо обчислення чи функція недостатньо ефективні, то витрати можуть перевищити користь. Тобто, якщо використовувати useMemo для простого додавання чисел, то витрати на мемоізацію можуть бути більшими, ніж користь від уникнення повторного виконання;

– часті зміни залежностей: якщо залежності хуків змінюються на кожному рендері (наприклад, через стан, який оновлюється щоразу), то мемоізація втрачає сенс. У такому випадку useMemo виконуватиме обчислення щоразу, а useCallback створюватиме нову функцію, що робить їх використання марним.

– надмірна мемоізація: якщо застосовувати ці хуки до всіх компонентів або обчислень без розбору, це може призвести до зайвої складності коду. Наприклад, обгортання кожного компонента у React.memo без потреби може зробити код важким для розуміння та підтримки, особливо якщо компоненти не отримують пропсів або їх рендеринг недорогий;

– проблеми з читабельністю та підтримкою: надмірне використання мемоізації може ускладнити відстеження потоку даних і стану в застосунку;

– витрати на пам’ять: кешування значень (useMemo) або функцій (useCallback) вимагає додаткової пам’яті, що може бути проблемою для великих застосунків, якщо мемоізація застосовується надмірно.

Проведений аналіз дозволив сформулювати висновок про те, що дослідження та розроблення моделей та методів підвищення ефективності рендерингу вебсторінок за останні роки набуває актуальності як з теоретичної, так і з прикладної точки зору.

Загальні аспекти вирішення питання контролю повторного рендерингу є критичними для забезпечення продуктивності React-застосунків і залишаються майже недослідженими. Саме контроль повторного рендерингу допомагає уникнути ситуацій, коли незначні зміни стану викликають каскадне оновлення всього дерева компонентів.

3. Мета і задачі дослідження

Метою даної роботи є розробка ефективної моделі оптимізації процесу рендерингу в сучасних вебзастосунках на основі визначення пріоритету рендерингу компонентів, що забезпечить мінімізацію повторних рендерів і покращить продуктивність вебзастосунків.

Для досягнення мети дослідження було визначено такі задачі:

– розробити модель оптимізації повторного рендерингу компонентів на основі їх видимості, важливості та вкладеності;

– оцінити ефективність запропонованої моделі шляхом порівняння часу рендерингу з оптимізацією та без неї.

4. Матеріали і методи дослідження

Об’єктом даного дослідження є процеси повторного рендерингу у вебзастосунках.

Основною гіпотезою даного дослідження запропоновано вважати можливість вирішення задачі розробки ефективної моделі оптимізації процесу рендерингу шляхом пріоритизації компонентів, які підлягають рендерингу.

Для перевірки цієї гіпотези під час дослідження було вирішено розробити модель оптимізації рендерингу компонентів у вебзастосунку, що дозволить мінімізувати повторні рендери, та розглянути її застосування на реальних компонентах, які є одними із розповсюдженних на сьогоднійшній день у веброзробці. За основу цієї моделі було запропоновано взяти один з існуючих методів пріоритизації, який дозволив би впорядкувати розташування окремих компонентів у черзі рендерингу. Пріоритизація дозволяє організувати оновлення так, щоб рендеринг відбувався для найважливіших і видимих компонентів першочергово, що, в свою чергу, зменшує навантаження на систему.

Існує кілька способів пріоритизації, які можуть бути застосовані для управління рендерингом, наприклад, модель Кано [11], метод числового оцінювання пріоритетів тощо. У контексті веброзробки часто використовуються методи, засновані на оцінці видимості компонентів (Intersection Observer API) або їхньої важливості для користувача. У цьому дослідженні було обрано комплексний підхід до оптимізації

рендерингу Selective Component Tree Hydration (SCTH), який враховує множинні фактори при прийнятті рішень про оновлення компонентів.

Для проведення експериментальної перевірки наукових результатів дослідження було запропоновано рішення для оптимізації повторного рендерингу в React-додатках, а саме, менеджер пріоритетної черги (Priority Queue Manager, PQM). Цей інструмент створений з метою зменшення кількості непотрібних повторних рендерів компонентів та підвищення продуктивності додатків шляхом інтелектуального управління рендерингом на основі пріоритетів та видимості компонентів у в'юпорті. Приорітизація дозволяє організувати оновлення так, щоб рендеринг відбувався для найважливіших і видимих компонентів першочергово, що в свою чергу зменшує навантаження на систему.

5. Вирішення задачі оптимізації рендерингу компонентів у вебзастосунках

Виходячи з положень комплексного підходу SCTH, було запропоновано використати для приорітизації комбіновану оцінку трьох факторів – видимості, важливості та вкладеності. Така оцінка дозволяє врахувати як технічні, так і користувацькі аспекти рендерингу.

Базуючись на цій пропозиції, було розроблено модель пріоритизації у вигляді:

$$P(c) = \alpha \cdot V(c) + \beta + I(c) + \gamma \cdot D(c) \quad (1)$$

де $V(c)$ – функція видимості; $I(c)$ – індекс важливості; $D(c)$ – глибина в дереві, визначається як $D(c) = 1 - (depth / maxDepth)$; α, β, γ – вагові коефіцієнти.

Функція видимості $V(c)$ відображає, чи знаходиться компонент у межах видимої частини екрану користувача (viewport). Чим більше компонент є видимим, тим більший його вплив на загальний пріоритет рендерингу. Якщо компонент є повністю видимим ($V(c)$ = 1.0), це означає, що він має високий пріоритет і має бути оновлений негайно. Якщо ж компонент частково видимий або зовсім невидимий ($V(c)$ = 0.5 або $V(c)$ =0.0), це означає, що його оновлення можна відкласти, оскільки він не є критичним для поточного етапу взаємодії користувача з інтерфейсом.

Індекс важливості $I(c)$ визначає, наскільки критичним є компонент для загального функціонування застосунку. Наприклад, для форм входу чи кошика покупок цей індекс буде наближатися до 1.0, оскільки ці компоненти є ключовими для користувацького досвіду та функціональності. Для важливих компонентів, таких як навігація, основний контент, індекс дорівнюватиме 0.7. Для менш важливих компонентів, таких як коментарі або додаткова інформація, індекс буде знижений до 0.4, оскільки вони не критичні для основної взаємодії користувача. Для неважливих

компонентів, таких як футер, реклама, індекс дорівнюватиме 0.1. Важливість компонентів прямо впливає на її пріоритет у рендерингу, тобто чим важливіший компонент, тим раніше він буде оновлений.

Глибина в дереві $D(c)$ вказує на рівень вкладеності компонента в ієрархії застосунку. Кореневі компоненти, що знаходяться на найвищому рівні ($D(c)$= 1.0), мають найбільший вплив на рендеринг, оскільки вони, як правило, керують основними частинами інтерфейсу. $D(c)$ може також набувати таких значень: 0.8 (перший рівень вкладеності), 0.6 (другий рівень), 0.4 (третій рівень), 0.2 (четвертий рівень і глибше). Чим глибше компонент знаходиться в дереві, тим меншим буде його вплив на загальний пріоритет рендерингу. Це дозволяє зменшити навантаження на процес оновлення, оскільки менш важливі компоненти, які знаходяться глибше в дереві, оновлюються лише тоді, коли це дійсно необхідно.

Вагові коефіцієнти α, β, γ дозволяють встановити важливість кожної з цих функцій у загальному розрахунку пріоритету.

Запропонована модель дає змогу визначати пріоритет рендерингу компонентів, базуючись на їхній видимості, важливості та позиції в дереві. Це дозволяє підвищувати ефективність рендерингу в умовах обмежених ресурсів, мінімізуючи повторні рендери, і таким чином мінімізувати затримки в оновленні інтерфейсу для користувача.

Для проведення експериментальної частини дослідження було розроблено систему, архітектура якої включає в себе чітко визначені компоненти, такі як Priority Queue Manager, Viewport Observer, та State Diff Calculator. Важливим аспектом розробленої системи є її інтеграція з існуючими інструментами розробника, зокрема з React DevTools, що дозволяє ефективно відстежувати та аналізувати процес рендерингу.

Перед використанням цієї системи було становлено такі значення вагових коефіцієнтів: α = 0.5 (видимість); β = 0.3 (важливість); γ = 0.2 (глибина).

Видимість компонента має найбільший вплив (α = 0.5), оскільки компоненти, які знаходяться у полі зору користувача, є найкритичнішими для оновлення. Індекс важливості має середній коефіцієнт (β = 0.3), оскільки важливі компоненти повинні бути оновлені швидше, але вони не завжди знаходяться у полі зору. Глибина компонента в дереві має найменший коефіцієнт (γ = 0.2), оскільки компоненти на глибших рівнях дерева, хоча й важливі, можуть бути оновлені пізніше.

За результатами експериментів із застосуванням розробленої моделі (1) система продемонструвала значне покращення продуктивності, зокрема, зменшення кількості

непотрібних повторних рендерів на 15-25 % та покращення FPS на 25-35 % для складних застосунків. В табл. 2 наведено результати порівняння часу рендерингу без оптимізації та з використанням оптимізації на основі моделі визначення пріоритету рендерингу компонентів.

Таблиця 2

Результати порівняння часу рендерингу без оптимізації та з використанням оптимізації на основі моделі визначення пріоритету рендерингу компонентів

	Використання пам’яті (середнє значення, МБ)	Час рендерингу (середнє значення, мс)	Кількість рендерів (середнє значення, мс)
З оптимізацією	61	7.17	100
Без оптимізації	64	7.5	100

6. Обговорення результатів дослідження

Дослідження свідчить, що застосування моделі визначення пріоритету рендерингу компонентів зменшує використання пам’яті, особливо пікові значення, і також оптимізує середній час рендерингу. Це вказує на можливість покращення балансу між цими метриками, що є важливим для розробників React-додатків, перед якими стоїть завдання підвищити продуктивність.

Особливу увагу було приділено розробці моделі пріоритизації компонентів та створення ефективних алгоритмів обробки черги оновлень та практичній реалізації системи. Запропоновано використати комплексний підхід до оптимізації рендерингу SCTH, який враховує множинні фактори при прийнятті рішень про оновлення компонентів. Основою системи є модель пріоритизації, яка враховує видимість компонентів, їхню важливість та положення в дереві компонентів.

Отримані результати дослідження пояснюються тим, що пріоритизація оновлень компонентів на основі їхньої видимості та важливості дозволяє уникнути непотрібних повторних рендерів.

Переваги запропонованих результатів порівняно з існуючими методами, такими як React.memo чи useMemo, полягають у комплексному підході до пріоритизації, який враховує не лише залежності, а й контекст використання компонентів. Наприклад, на відміну від React.memo, яке лише запобігає рендерингу при незмінних пропсах, розроблена модель (1) дозволяє визначати пріоритети оновлення залежно від видимості компонентів, що є гнучкішим рішенням.

Запропоноване рішення частково вирішує проблему надмірного рендерингу, зменшуючи кількість повторних рендерів та оптимізуючи використання ресурсів. Проте його ефективність залежить від правильного налаштування вагових коефіцієнтів (α, β, γ), що потребує додаткових експериментів. Іншими обмеженнями дослідження є відсутність врахування історії змін стану компонентів, що могло б покращити динамічність моделі, а також обмежена кількість експериментальних даних.

У наступних дослідженнях планується адаптувати модель для інших фреймворків (Vue.js, Angular) та інтегрувати її з технологіями, такими як WebAssembly, для подальшої оптимізації продуктивності. Як додатковий напрям подальших досліджень планується розглянути інтеграцію аналізу часових рядів для динамічної пріоритизації.

7. Висновки

У ході дослідження було проведено детальний аналіз проблеми повторного рендерингу у сучасних React-застосунках. Було запропоновано вирішити задачу розробки ефективної моделі оптимізації процесу рендерингу шляхом пріоритизації компонентів, які підлягають рендерингу. Для цього було вирішено використати комплексний підхід до оптимізації рендерингу SCTH, який враховує множинні фактори при прийнятті рішень про оновлення компонентів.

З використанням комплексного підходу SCTH було розроблено лінійну модель пріоритизації (1). Розроблена модель дозволила визначати пріоритет рендерингу компонентів, базуючись на їхній видимості, важливості та позиції в дереві. Застосування моделі (1) дозволило підвищити ефективність рендерингу в умовах обмежених ресурсів, мінімізуючи повторні рендери, і таким чином мінімізувати затримки в оновленні інтерфейсу для користувача.

Для апробації розробленої моделі (1) було проведено експериментальні дослідження. Результати цих досліджень, наведені в табл. 2, показують, що застосування моделі (1) дозволяє зменшити середній час рендерингу з 7,5 мс до 7,37 мс та знизити пікове використання пам’яті з 64 МБ до 61 МБ.

ДОДАТОК Г

Тези на концференції "MIT@AISM-2025"

OPTIMIZING RE-RENDERING IN WEB APPLICATIONS: PROBLEM ANALYSIS AND REACT-BASED SOLUTIONS

A.L. YEROKHIN, D.V. KAMENEV

[1] *andriy.yerokhin@nure.ua*
[2] *dmytro.kameniev@nure.ua*
Keywords: re-rendering ; user experience; virtual DOM; flexible prioritization; React.

BACKGROUND

Modern web applications built on JavaScript frameworks such as React, Vue, and Angular suffer from excessive re-rendering, which slows down performance and degrades the user experience. React is particularly vulnerable to this problem due to the frequent updating of the Document Object Model (DOM) tree when state changes, which can cause cascaded updates to components. For example, a small state change such as typing text can cause the entire page to be re-rendered, which increases compute overhead and accelerates battery drain on mobile devices. Existing optimization methods such as React.memo, useMemo, and useCallback are not always effective because they do not take into account the context of components, such as their visibility or importance to the user. As web applications become more complex and performance requirements increase, optimizing rendering becomes not only a technical but also a business-critical task that impacts user satisfaction and commercial success. Research shows that inefficient rendering management leads to interface lag, reduced FPS (frames per second), and increased memory consumption, which is unacceptable in a competitive digital environment.

OBJECTIVE

The goal of the study is to develop an effective model for optimizing re-rendering in React applications by prioritizing components based on their visibility, importance, and nesting in the component tree. The model aims to minimize unnecessary renders, reduce computational load, and increase interface responsiveness, which improves performance and user experience.

METHODS

The study is based on the analysis of the virtual DOM and reconciliation mechanisms in React, in particular the Fiber algorithm, which allows for asynchronous processing of updates. A component prioritization model was developed using the formula $Pc = \alpha \times V(c) + \beta \times I(c) + \gamma \times D(c)$, where $V(c)$ is the visibility function, $I(c)$ is the importance index, $D(c)$ is the depth in the tree, and $\alpha=0.5$, $\beta=0.3$, $\gamma=0.2$ are the weights. Visibility is assessed using the Intersection Observer API, which determines whether a component is in the viewport. The importance index depends on the functional role of the component (e.g., 1.0 for login forms, 0.4 for comments). The depth in the tree reflects the level of nesting, where root components have the highest priority ($D(c)=1.0$). A system with Priority Queue Manager (PQM), Viewport Observer, and State Diff Calculator components integrated with React DevTools was created for the experiments. Testing was conducted on complex React applications, comparing rendering time, number of renders, and memory usage with and without optimization.

RESULTS

The proposed model demonstrated significant performance improvements. The average rendering time decreased from 7.50 ms to 7.37 ms, peak memory usage from 64 MB to 61 MB, and the number of unnecessary renders decreased by 15-25%. In complex applications, FPS increased by 25-35%, which provided a smoother interface. Compared to traditional methods such as React.memo, the model is more flexible because it takes into account the visibility and context of components, and not just the immutability of props. For example, components outside the viewport receive a lower priority, which reduces the load on the system. Integration with React DevTools allowed for accurate rendering tracking, confirming the effectiveness of the approach. The limitation is the need for fine-tuning of weighting factors, which depends on the specifics of the application.

CONCLUSION

The developed prioritization model based on Selective Component Tree Hydration (SCTH) effectively solves the problem of excessive rendering in React applications, minimizing unnecessary updates and optimizing resources. Its advantage over standard methods such as React.memo or useMemo is that it comprehensively takes into account the visibility, importance, and nesting of components, which ensures adaptability to different scenarios. Experimental results confirm the reduction of rendering time and memory usage, which improves the user experience. In the future, it is planned to adapt the model for other frameworks such as Vue.js and Angular, as well as integrate with WebAssembly to improve performance. Additional research may include time series analysis for dynamic prioritization, which will make the model even more effective in large-scale applications.

LIST OF SOURCES:

1.Ollila R., Makitalo N., Mikkonen T. Modern Web Frameworks: A Comparison of Rendering Performance. Journal of Web Engineering. 2022. Vol. 23, Iss. 3. P. 789–814. https://doi.org/10.13052/jwe1540-9589.21311.
2.Shah H. Server-Side Rendering vs Static Site Generation: Choosing the Right Approach for Your Next.js Project. DEV Community. URL: https://dev.to/shahharsh/server-side-rendering-vs-static-site-generation-choosing-the-right-approach-for-your-nextjs-project-39op (дата звернення: 12.03.2025).
3.Singh P., Srivastava M., Kansal M., Singh A.P., Chauhan A., Gaur A. A Comparative Analysis of Modern Frontend Frameworks for Building Large-Scale Web Applications. 2023 International Conference on Disruptive Technologies (ICDT). Greater Noida, India, 2023. P. 531-535. https://doi.org/10.1109/ICDT57929.2023.10150911.