Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий(магістерський)
(повна назва)

Дослідження проблем ререндерингу у веб сторінок у фреймворках та засобів їх вирішення для бібліотеки React
(тема)

Виконав:
здобувач 2 року навчання
групи ІПЗм-23-2

Дмитро КАМЕНЄВ
(власне ім'я, прізвище)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Керівник проф. Андрій ЄРОХІН
(посада, власне ім'я, прізвище)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти другий (магістерський)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)
Тип програми освітньо-наукова програма
(освітньо-професійна або освітньо-наукова)
Освітня програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 20___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувачеві Каменєв Дмитро Вікторович
(прізвище, ім'я, по батькові)

1. Тема роботи: Дослідження проблем ререндерингу у веб сторінок у фреймворках та засобів їх вирішення для бібліотеки React
затверджена наказом університету №290 Ст від 15.04.2025

2. Термін подання здобувачем роботи до екзаменаційної комісії 16.06. 2025 р.

3. Вихідні дані до роботи:
Електронні ресурси за обраною тематикою, аналіз сучасних статей на тематику рендерингу сучасних фреймворків, порівняння сучасних методів рендерингу, вирішення проблеми повторного рендерингу бібліотеки React

4. Перелік питань, що потрібно опрацювати у роботі:
Всебічний аналіз провідних JavaScript-фреймворків, що використовуються для фронтенд-розробки, з акцентом на виявлення та вирішення специфічної проблеми подвійного рендерингу у бібліотеці React.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Отримання завдання	16.04.2025	*виконано*
2	Аналіз предметної галузі і постановка задачі	20.05.2024	*виконано*
3	Назва за розділами теоретичного і практичного дослідження	21.05.2024	*виконано*
4	Назва за розділами теоретичного і практичного дослідження	21.05.2024	*виконано*
5	Підготовка до апробації результатів дослідження. Публікація матеріалів	26.05.2024	*виконано*
6	Назва за розділами теоретичного і практичного дослідження	30.05.2025	*виконано*
7	Підготовка пояснювальної записки	20.06.2025	*виконано*
8	Підготовка презентації та доповіді	04.06.2025	*виконано*
9	Перевірка на плагіат	08.06.2025	*виконано*
10	Нормоконтроль	09.06.2025	*виконано*
11	Рецензування	10.06.2025	*виконано*
12	Попередній захист	13.06.2025	*виконано*
13	Занесення диплома в електронний архів	14.06.2025	*виконано*
14	Допуск до захисту у зав. кафедри	15.06.2025	*виконано*

Дата видачі завдання 16.04.2025 р.

Здобувач _____ Дмитро КАМЕНЄВ
(підпис)

Керівник роботи _____ проф. Андрій ЄРОХІН
(підпис) (посада, власне ім’я, прізвище)

РЕФЕРАТ / ABSTRACT

Робота містить: 73 с., 8 рис., 15 джерел.

COMPONENT, DEVTOOLS, DOM, FRONT END, FRAMEWORK, GITHUB, MEMOIZATION, REACT, REACT.MEMO, RECONCILIATION, STACK OVERFLOW DEVELOPER SURVEY, STATE UPDATES, VIRTUAL DOM.

Об’єктом дослідження є аналіз статей та досліджень у галузі проблем переднерінга сучасних фреймворків та вибір найбіль актуального.

Метою роботи є проведення дослідження з виршіення найбільш актуального і перспективного фреймворку зі сторони проблеми ререндерингу.

Методами розробки та проектування є аналіз проблемної області дослідження, вибір застосунку REACT для проведення дослідження шляхом вирішення багатокритеріальної задачі прийняття рішень.

У результаті практично-наукового дослідження було розроблено програму, яка за теоритичними розрахунками впливає на результат роботи бібліотеки REACT і частково вирішує проблеми ререндерингу компонентів шляхов оптимізованого рендерингу компонентів.

COMPONENT, DEVTOOLS, DOM, FRONT END, FRAMEWORK, GITHUB, MEMOIZATION, REACT, REACT.MEMO, RECONCILIATION, STACK OVERFLOW DEVELOPER SURVEY, STATE UPDATES, VIRTUAL DOM.

The object of the study is the analysis of articles and research in the field of rerendering problems of modern frameworks and the selection of the most relevant one.

The purpose of the work is to conduct research on solving the most relevant and promising framework from the side of the rerendering problem.

The development and design methods are the analysis of the problem area of the study, the selection of the REACT application for conducting the study by solving a multi-criteria decision-making problem.

As a result of the practical and scientific research, a program was developed that, according to theoretical calculations, affects the result of the REACT library and partially solves the problems of rerendering components by optimizing the rendering of components.

Завідувачу кафедри

ПІ_____

(скорочена назва кафедри)

проф. Кирилу СМЕЛЯКОВУ_

(вчене звання, сласне ім’я, прізвище)

ЗАЯВА
щодо самостійності виконання кваліфікаційної роботи та можливості її публікації (та/або публікації анотації кваліфікаційної роботи) в електронному архіві відкритого доступу ElAr KhNURE

Я, Каменєв Дмитро Вікторович,

здобувач вищої освіти на другому (магістерському) рівні вищої освіти академічної групи ІПЗм-23-2

кафедра_____ програмної інженерії _____,

заявляю: моя кваліфікаційна робота на тему Дослідження проблем ререндерингу у веб сторінок у фреймворках та засобів їх вирішення для бібліотеки React

що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в репозиторії "ElArKhNURE". Погоджуюся з авторським договором, відповідно до Положення про репозиторій ХНУРЕ "ElArKhNURE". Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з вимогами академічної доброчесності, згідно з якими виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

Дата Підпис

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

DOM – Document Object Model

API – Application Programming Interface

SPA – Single Page Application

DEVTOOLS – Developer Tools

PQM – Priority Queue Manager

SCTH – System Component Tree Handler

URL – Uniform Resource Locator

ВСТУП

Сучасний стан розвитку веб-технологій характеризується стрімким зростанням складності клієнтських додатків, що базуються на JavaScript-фреймворках, таких як React, Vue.js чи Angular. Ці інструменти дозволяють створювати інтерактивні та масштабовані інтерфейси, однак разом із перевагами виникають і проблеми, пов’язані з неефективним ререндерингом компонентів. Надмірні перерисовки інтерфейсу призводять до зниження продуктивності, збільшення часу відгуку додатку та погіршення користувацького досвіду. У рамках підготовки до дослідження було проаналізовано низку статей на тему проблем ререндерингу в сучасних фреймворках, зокрема праці, опубліковані у технічних блогах спільноти (наприклад, DEV Community, Medium) та офіційній документації фреймворків. На основі цього аналізу за критеріями продуктивності, популярності та гнучкості було обрано бібліотеку React, оскільки саме в ній, згідно з розглянутими джерелами, проблема ререндерингу є особливо гострою через особливості роботи віртуального DOM та відсутність автоматичних механізмів оптимізації. Таким чином, React потребує вдосконалення підходів до рендерингу для ефективного вирішення цієї проблеми.

Актуальність роботи зумовлена зростаючою потребою у створенні швидких та масштабованих веб-додатків, де продуктивність відіграє ключову роль. У контексті великих проєктів, де кількість компонентів і взаємозв’язків між ними зростає, проблема ререндерингу стає критичною. Незважаючи на наявність методів оптимізації, таких як мемоізація чи управління залежностями в хуках, ці рішення часто є ситуативними і не пропонують універсального підходу. Таким чином, розробка нових інструментів для автоматичного аналізу та зменшення кількості ререндерингів є важливим завданням, що має як теоретичне, так і практичне значення. Підставою для проведення дослідження стало прагнення систематизувати знання про причини надмірного ререндерингу та запропонувати практичні рішення,

які можна інтегрувати у реальні проєкти.

Метою даної роботи є дослідження проблем рендерингу в React-додатках та розробка програмного рішення у вигляді бібліотеки, яка дозволить зменшити кількість непотрібних перерисовок компонентів, підвищивши тим самим продуктивність додатків.

Для досягнення цієї мети було поставлено наступні задачі:

- проаналізувати основні причини виникнення надмірного рендерингу в React;
- розробити набір утиліт для автоматичної оптимізації рендерингу;
- протестувати запропоноване рішення на модельних прикладах;
- оцінити ефективність розробленої бібліотеки шляхом порівняння продуктивності додатків з її використанням та без нього.

Об’єктом дослідження є процес рендерингу компонентів у React-додатках, а предметом – методи та інструменти оптимізації цього процесу. У роботі застосовувалися такі методи дослідження: аналіз літератури та документації для вивчення теоретичних основ, експериментальний метод для тестування розроблених утиліт, а також порівняльний аналіз для оцінки результатів. Теоретичною базою слугували офіційна документація React, статті провідних розробників спільноти та практичний досвід роботи з фреймворком.

Практичне значення одержаних результатів полягає у створенні рішення, яке може бути використане розробниками для спрощення процесу оптимізації React-додатків. Запропоноване рішення дозволяє зменшити час, витрачений на ручну настройку мемоізації, а також підвищити продуктивність додатків без необхідності глибокого занурення в технічні деталі. Результати роботи можуть бути застосовані як у невеликих проєктах, так і в складних системах із великою кількістю компонентів, що робить їх універсальними для широкого кола розробників.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної області

Розвиток веб-технологій за останнє десятиліття призвів до значного ускладнення клієнтської частини веб-застосунків. Якщо на початку розвитку веб-технологій JavaScript використовувався переважно для простих маніпуляцій з DOM, то сьогодні на його основі будуються складні односторінкові додатки (SPA) з багатим користувацьким інтерфейсом та складною бізнес-логікою. В процесі еволюції веб-розробки виникли різні підходи до створення користувацьких інтерфейсів. React, розроблений Facebook (нині Meta), представив концепцію Virtual DOM та компонентний підхід. Vue.js запропонував реактивну систему та поєднав найкращі практики React та Angular. Angular від Google впровадив повноцінний фреймворк з власною екосистемою та архітектурними рішеннями.

Механізми рендерингу в сучасних фреймворках будуть описані нижче.

React та Virtual DOM. React використовує концепцію Virtual DOM, яка представляє собою легковажну копію реального DOM. Коли стан компонента змінюється, React спочатку оновлює Virtual DOM, а потім порівнює його з попередньою версією через процес "reconciliation" (примирення). Цей підхід дозволяє мінімізувати кількість операцій з реальним DOM, який є відносно повільним для маніпуляцій.

Проте, незважаючи на ефективність Virtual DOM, React може стикатися з проблемами продуктивності при неправильному управлінні повторними рендерами. Наприклад, зміна стану батьківського компонента може викликати каскадний перерендер усіх дочірніх компонентів, навіть якщо їх пропси не змінилися.

Vue.js та реактивна система. Vue.js використовує гібридний підхід, поєднуючи реактивну систему з Virtual DOM. Реактивна система Vue.js автоматично відстежує залежності між даними та оновлює тільки ті компоненти, які дійсно залежать від змінених даних. Це забезпечує більш точну граничність оновлень порівняно з React.

У Vue 3 була представлена нова система реактивності на основі Proxy, яка забезпечує кращу продуктивність та більш передбачувану поведінку. Проте, як і в React, неправильне використання реактивної системи може призвести до надмірних перерендерів.

Angular та Change Detection. Angular використовує власний механізм виявлення змін (Change Detection), заснований на Zone.js. За замовчуванням, цей механіщм перевіряє всі компоненти дерева додатка на кожну зміну в контексті подій, таких як введення користувача або зміни асинхронних даних. Коли відбувається якась зміна, Angular ініціює цикл перевірки, порівнюючи поточний стан компонентів з їх попереднім станом, і якщо зміни виявлено, перерендерює відповідні частини DOM. Залежно від налаштувань, Angular може використовувати різні стратегії детекції змін [12]. Одна з найбільш популярних – Default Change Detection Strategy, яка перевіряє всі компоненти у дереві, коли змінюється будь-яка частина програми.

Альтернативно, можна використовувати OnPush Change Detection Strategy, яка обмежує перевірку лише до компонентів, вхідні дані яких змінилися або коли викликаються певні події, таким чином знижуючи обсяг перевірок і підвищуючи продуктивність. Саме так і працює - відстежує всі асинхронні операції та автоматично запускає перевірку змін після їх завершення. Angular пропонує два режими виявлення змін: Default та OnPush, що дозволяє розробникам контролювати частоту перевірок

Проблеми повторного рендерингу у фреймворках React, Angular і Vue.js часто стають основною причиною зниження продуктивності веб-застосунків. У кожному з цих фреймворків рендеринг є важливим етапом оновлення інтерфейсу, і хоча кожен із них використовує різні підходи до управління змінами, усі вони стикаються з проблемою надмірного рендерингу [6].

У React зайві рендери можуть виникати, коли компоненти перерендерюються занадто часто через неправильне управління станом або пропсами. Наприклад,

навіть якщо стан компонента не змінився, React може не мати можливості ідентифікувати, чи дійсно потрібно оновлювати компонент. Це особливо стосується складних компонентів або тих, що використовують великі об'єми даних, що може призвести до затримок у відображенні інтерфейсу. Такі ситуації часто виникають через відсутність оптимізації, наприклад, використання React.memo або useMemo, що дозволяє зберігати попередні результати рендерингу та зменшувати обсяг необхідних перевірок. Можемо виділити основні в списку:

- складність контролю над каскадними оновленнями;
- необхідність ручної оптимізації з використанням React.memo та useMemo;
- проблеми з Context API та глобальним станом.

В Angular проблема повторного рендерингу здебільшого пов'язана з механізмом Change Detection, який перевіряє всі компоненти на наявність змін [9]. Якщо цей процес не оптимізовано, він може стати занадто затратним, оскільки Angular перевіряє величезну кількість компонентів, навіть якщо зміни відбулися лише в окремих частинах додатка. Ця проблема посилюється, коли використовуються складні, глибокі структури компонентів, і часто виникає необхідність вручну налаштовувати стратегії детекції змін, як-от OnPush, щоб обмежити кількість перевірок і підвищити ефективність.

Також виділимо основні в Angular:

- складність налаштування Change Detection Strategy;
- проблеми з продуктивністю при використанні двостороннього зв'язування;
- обмеження при роботі з Zone.js.

Vue.js, хоча й має свою систему оптимізації рендерингу, також не застрахований від проблем, пов'язаних із надмірним рендерингом. Зазвичай це відбувається, коли компоненти без необхідності отримують нові пропси або змінюється глобальний стан, що веде до перерендерів без видимої потреби. У Vue.js ця проблема може бути пом'якшена завдяки реактивності та використанню computed властивостей, однак неправильно налаштована реактивність або неправильне

управління станом можуть призвести до тих самих проблем з продуктивністю [8].

Підсумуємо проблеми Vue.js у рендері:

– складність відстеження реактивних залежностей у великих застосунках;
– проблеми з оптимізацією при використанні складних обчислюваних властивостей;
– особливості роботи з референсами та реактивними об'єктами [8].

В усіх трьох фреймворках ключова проблема повторного рендерингу полягає в неефективному управлінні змінами стану, що може призводити до зайвих оновлень інтерфейсу, що в свою чергу знижує швидкість роботи додатків і погіршує користувацький досвід. Тому для забезпечення високої продуктивності розробники повинні ретельно налаштовувати механізми детекції змін і оптимізації рендерингу, щоб уникнути зайвих перевірок і забезпечити швидке оновлення лише тих частин інтерфейсу, які дійсно потребують змін.

1.2 Постановка задач дослідження

Для досягнення мети дослідження, яка полягає в оптимізації повторного рендерингу у вебзастосунках на основі бібліотеки React, необхідно виконати низку конкретних завдань [13]. Ці завдання спрямовані на аналіз проблеми надмірного рендерингу, розробку та впровадження Priority Queue Manager як інструменту для інтелектуального керування оновленнями компонентів, а також на оцінювання його ефективності шляхом тестування на вебзастосунку аналог якого є актуальним і на сьогоднішній день – високонавантажений вебдодаток з об`ємною кількістю компонентів, що у свою чергу робить його максимально схожим до реальних застосунів, які мають популярність серед користувачів [12]. Завдання сформульовано з урахуванням вимог до їхньої конкретності, вимірюваності, досяжності та актуальності. Завдання були сформовані наступним чином:

– аналіз проблеми повторного рендерингу в React-застосунках. Дослідити причини надмірного рендерингу компонентів у вебзастосунках,

побудованих на React, зокрема вплив використання хуків (useMemo, useEffect, useCallback) та React.memo. Визначити типові сценарії, які призводять до неоптимального оновлення компонентів, та оцінити їхній вплив на продуктивність застосунків із великою кількістю компонентів (близько 50 на одній сторінці);

- розробка концепції Priority Queue Manager. Сформулювати теоретичні основи та розробити архітектуру Priority Queue Manager – інструменту для інтелектуального керування порядком оновлення компонентів React. Визначити критерії пріоритизації компонентів (наприклад, їхня видимість, частота оновлення, складність обчислень) та механізми оцінки доцільності повторного рендерингу;
- реалізація Priority Queue Manager на TypeScript. Створити програмну реалізацію Priority Queue Manager з використанням TypeScript, інтегрувавши її в екосистему React. Забезпечити сумісність із типовими патернами розробки React-застосунків, мінімізуючи необхідність використання boilerplate-коду, такого як надмірне обгортання компонентів у useMemo, useCallback або React.memo;
- оптимізація кількості рендерингів компонентів. Розробити алгоритми, які дозволяють Priority Queue Manager запобігати рендерингу компонентів, чиї пропси або стан не змінилися. Забезпечити зменшення кількості непотрібних рендерингів шляхом динамічного аналізу залежностей і пріоритизації оновлень;
- розробка тестового вебзастосунку. Створити модельний вебзастосунок на основі React, який містить приблизно 50 компонентів на одній сторінці, для імітації реального сценарію з великою кількістю компонентів. Застосунок має включати різноманітні типи компонентів (статичні, динамічні, з частим оновленням стану) для комплексного тестування ефективності Priority Queue Manager;

- тестування продуктивності рішення. Провести експериментальне тестування Priority Queue Manager на розробленому вебзастосунку. Для замірів продуктивності використати інструменти console.log, API Memory та React DevTools, реалізовані у функції logMetrics. Виміряти приріст швидкості оновлення інтерфейсу (ціль – 10-15%) порівняно зі стандартними підходами без використання Priority Queue Manager. Оцінити зменшення обсягу boilerplate-коду шляхом порівняння кількості хуків і обгорток у коді до та після впровадження рішення.
- аналіз і порівняння результатів. Провести порівняльний аналіз ефективності Priority Queue Manager з іншими методами оптимізації рендерингу (наприклад, стандартними інструментами React, такими як React.memo або useMemo). Оцінити зменшення кількості рендерингів, витрати пам’яті та час виконання на основі даних, зібраних за допомогою функції logMetrics. Проаналізувати переваги та недоліки запропонованого підходу, а також його вплив на читабельність коду та зручність розробки.

Ці завдання є взаємопов’язаними та спрямованими на досягнення основної мети дослідження – створення ефективного інструменту для оптимізації рендерингу в React-застосунках, що зменшує кількість boilerplate-коду, підвищує продуктивність на 10-15% і забезпечує стабільну роботу складних інтерфейсів із великою кількістю компонентів.

2 ОГЛЯД ІСНУЮЧИХ ТЕХНОЛОГІЙ ОПТИМІЗАЦІЇ РЕНДЕРИНГУ

Оптимізація рендерингу в React є ключовим аспектом створення продуктивних веб-додатків, адже надмірні перерисовки компонентів можуть суттєво уповільнити роботу інтерфейсу. React, розроблений Facebook у 2013 році, базується на концепції віртуального DOM, що дозволяє ефективно оновлювати лише змінені частини інтерфейсу [7].

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, змінюється на), то React видаляє старе дерево і створює нове з нуля;

– якщо елементи мають однаковий тип (наприклад, два), 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.

Оцінюючі підхід до реалізації, можна сказати, що бібліотека використовує дійсно оптимальний підхід до відображення змін на DOM. Але, без належної оптимізації, компоненти можуть ререндеритися навіть тоді, коли їхній стан чи пропси не змінюються, що призводить до зайвих обчислень. Історія розвитку інструментів оптимізації рендерингу в React почалася з базових підходів у класах (shouldComponentUpdate), а з появою хуків у 2019 році (React 16.8) з’явилися нові методи, такі як useMemo та useCallback, які значно розширили можливості розробників. Ці інструменти стали відповіддю на зростаючу складність додатків та потребу в підвищенні продуктивності.

2.1 React.memo

React.memo – це інструмент вищого порядку (Higher-Order Component, HOC), який з’явився у React 16.6 (2018 рік) для оптимізації функціональних компонентів. Він працює шляхом мемоизації компонента, дозволяючи уникнути його повторного рендерингу, якщо пропси не змінилися(див.рис.2.1).

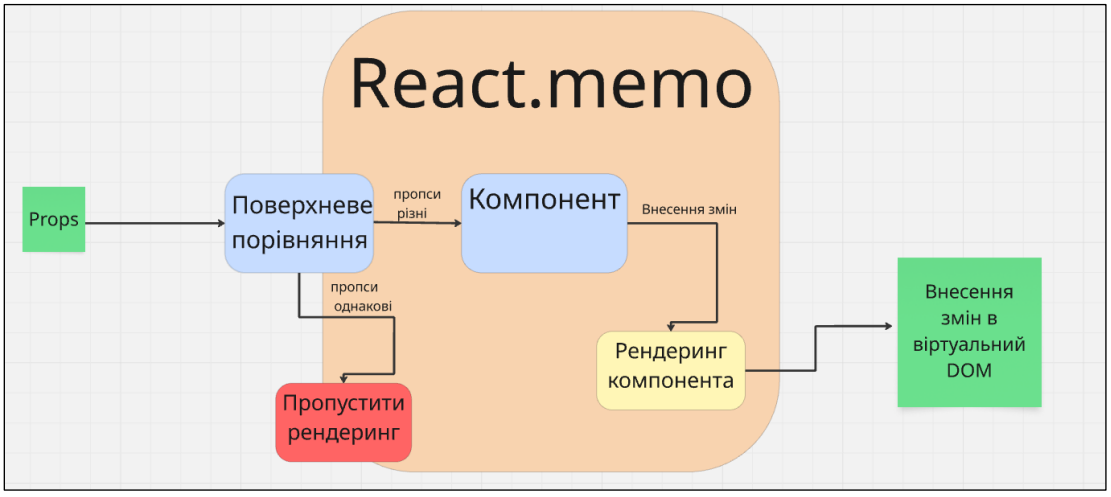

Рисунок 2.1 – Примітивна схема роботи компонента високого рівня React.memo[1]

Принцип дії схожий на метод shouldComponentUpdate у класах, але адаптований для функціонального підходу. Під час рендерингу React порівнює попередні та нові пропси за допомогою поверхневого порівняння (shallow comparison) і, якщо вони ідентичні, пропускає оновлення компонента. Наприклад, у списку з сотень елементів React.memo може запобігти перерисовці незмінних елементів, коли змінюється лише один.

Цей інструмент особливо корисний у випадках, коли компонент отримує складні пропси (об’єкти чи масиви), але не залежить від їхньої мутації. Проте, варто враховувати, що порівняння пропсів саме по собі вимагає обчислень, тому React.memo ефективний лише для компонентів із дорогими рендерами. Його практичне застосування включає оптимізацію великих дерев компонентів, таких як таблиці чи графіки.

2.2 useCallback

Хук useCallback – це інструмент для мемоизації функцій, введений разом із хуками у React 16.8 (див.рис.2.2).

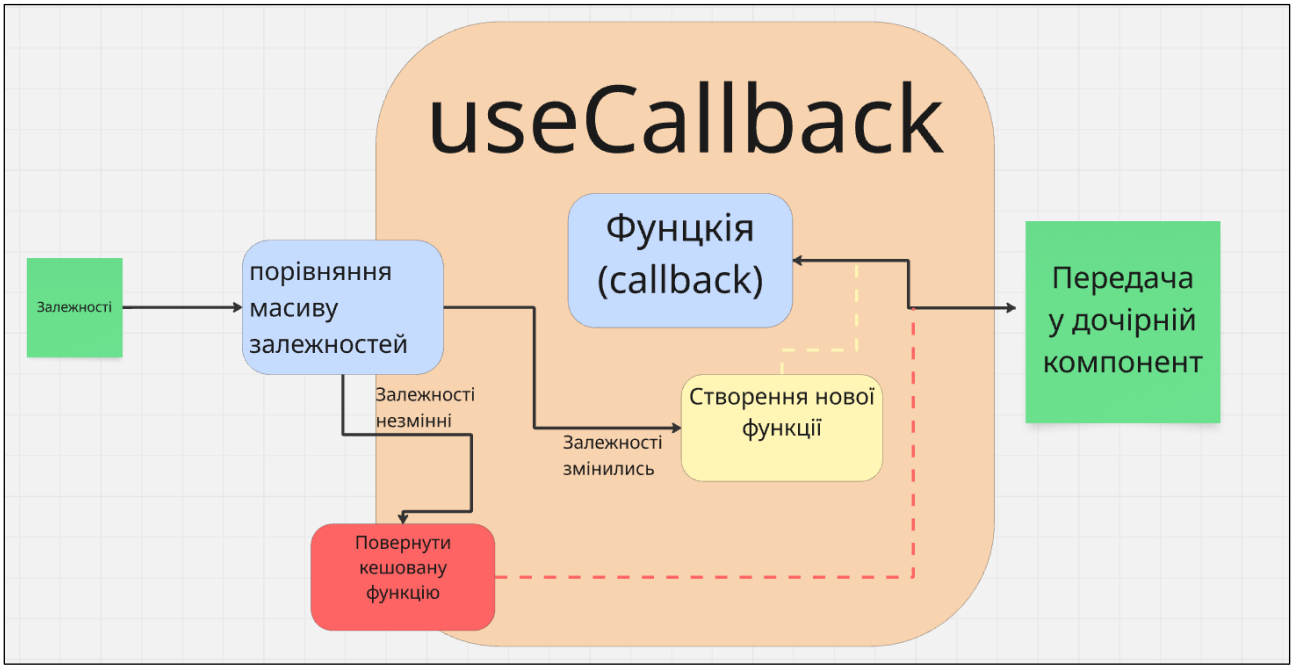

Рисунок 2.2 – Примітивна схема роботи хука UseCallback [2]

Він повертає стабільну версію функції, яка не змінюється між рендерами, якщо масив залежностей залишається незмінним. Це вирішує проблему, коли функції, передані як пропси дочірнім компонентам, створюються заново при кожному рендері батьківського компонента, що може викликати непотрібні перерисовки, навіть якщо компонент обгорнутий у React.memo.

Принцип роботи useCallback полягає в кешуванні функції та її поверненні лише за умови зміни залежностей [7]. Наприклад, у формі з обробником подій useCallback може стабілізувати функцію обробки, зменшуючи рендеринг дочірніх компонентів, які її отримують. Хук широко застосовується для оптимізації передачі коллбеків у великих додатках, таких як інтерактивні дашборди чи редактори.

2.3 useMemo

Хук useMemo, також представлений у React 16.8, дозволяє мемоїзувати результат обчислень, повертаючи кешоване значення, якщо залежності не змінилися. Він використовується для оптимізації дорогих обчислень, таких як фільтрація великих масивів чи генерація складних структур даних (див.рис.2.3).

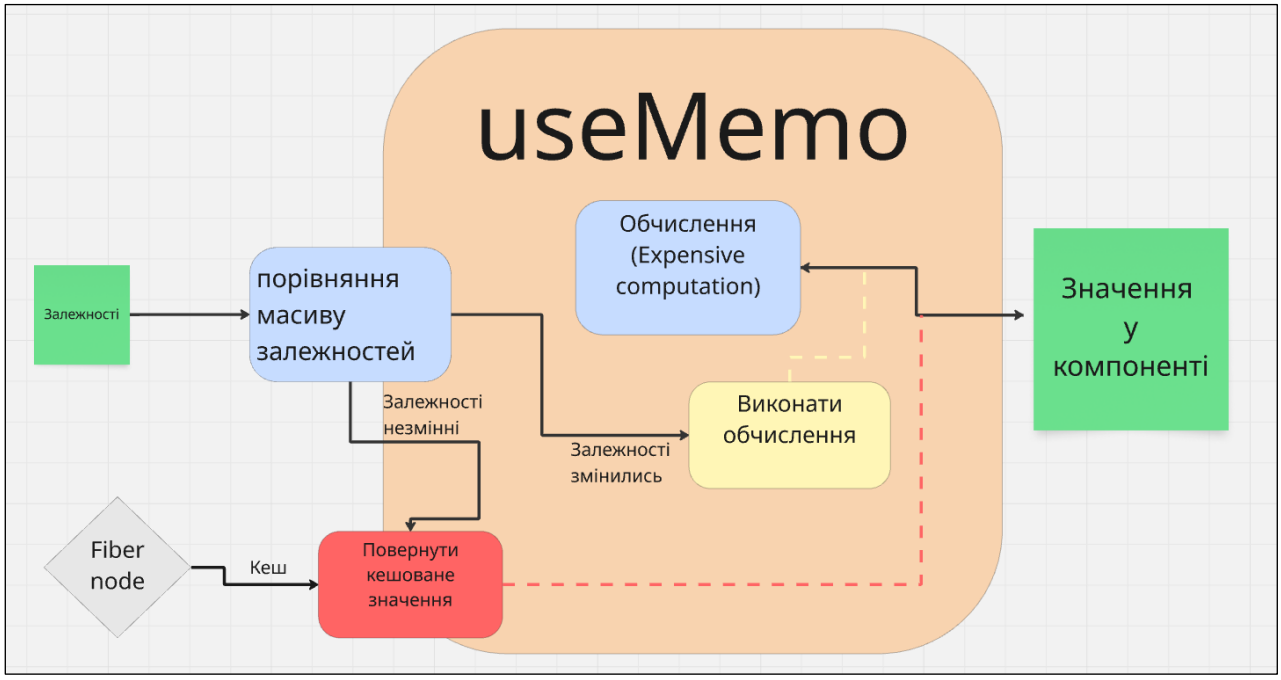

Рисунок 2.3 – Примітивна схема роботи хука UseMemo [3]

Наприклад, у компоненті, який рендерить відфільтрований список, useMemo може запобігти повторним обчисленням при кожному рендері.

Механізм роботи useMemo базується на порівнянні масиву залежностей: якщо вони не змінилися, повертається збережений результат. Це зменшує навантаження на процесор і покращує продуктивність у сценаріях із великими обсягами даних. Проте, як і у випадку з React.memo, варто зважати на витрати на порівняння залежностей, тому хук доцільно застосовувати лише для дійсно ресурсоємних операцій.

2.4 useState

Хук useState, хоча й не є інструментом оптимізації рендерингу напряму, відіграє важливу роль у контролі оновлень компонентів (див.рис.2.4).

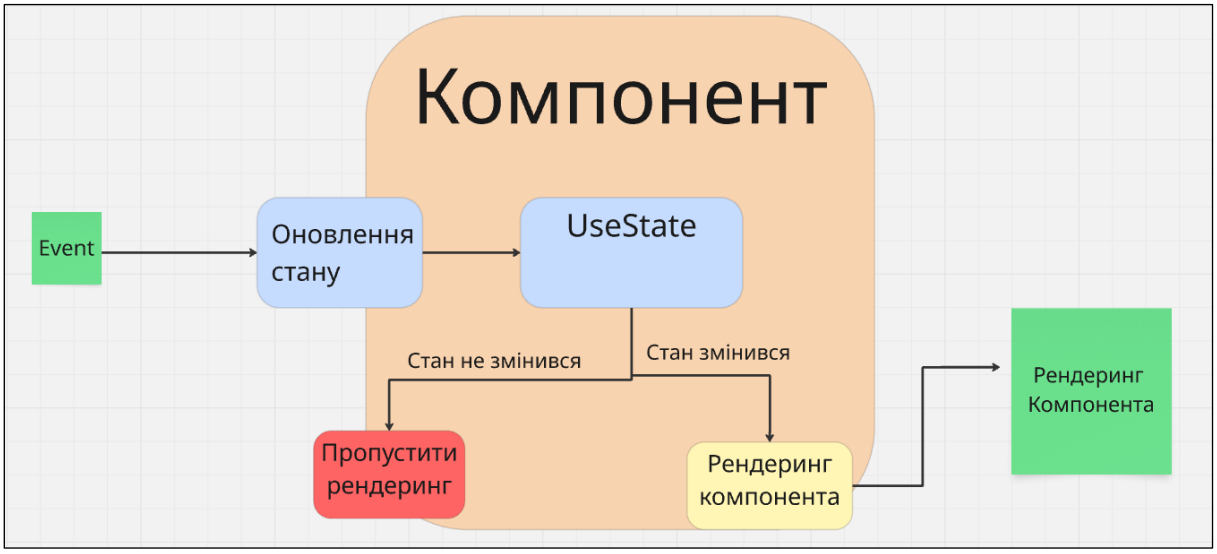

Рисунок 2.4 – Примітивна схема роботи хука UseState [4]

Він дозволяє управляти станом, який викликає ререндеринг лише при зміні значення. Наприклад, правильне використання useState із функційним оновленням (setState(prev => ...) замість прямого виклику) може запобігти непотрібним перерисовкам у складних сценаріях, таких як оновлення об’єктів чи масивів.

Оптимізація через useState досягається завдяки продуманому управлінню станом: розробник може групувати пов’язані дані в один стан, щоб уникнути множинних ререндерингів. Наприклад, замість трьох окремих станів для імені, email та пароля у формі, їх можна об’єднати в один об’єкт, зменшивши кількість оновлень компонента.

2.5 Практичне значення та порівняння

Перелічені інструменти – React.memo, useCallback, useMemo та useState – складають основу оптимізації рендерінгу в React і є незамінними для розробників, які прагнуть підвищити продуктивність веб-додатків. Ці методи дозволяють контролювати, коли і як компоненти перерисовуються, зменшуючи обчислювальне навантаження, оптимізуючи використання ресурсів і підвищуючи швидкодію інтерфейсу. Їхнє практичне значення особливо помітне у великих і складних

проєктах, таких як соціальні мережі, платформи електронної комерції чи аналітичні дашборди, де навіть невелике покращення продуктивності може суттєво вплинути на користувацький досвід, зменшуючи час завантаження сторінок і підвищуючи залученість користувачів.

Кожен із цих методів має свої унікальні сильні сторони, що робить їх придатними для різних сценаріїв. React.memo є ідеальним рішенням для компонентів із простими пропсами, такими як текстові відображення чи статичні елементи списків, дозволяючи заощаджувати до 30-40% ресурсів процесора в залежності від розміру даних. Однак його ефективність знижується, якщо пропси є об’єктами чи масивами, які змінюються через посилання, а не вміст. useCallback забезпечує стабільність функцій, переданих як пропси, що критично важливо для дочірніх компонентів, і може зменшити кількість ререндерів на 50% у складних ієрархіях, хоча надмірне використання з великою кількістю залежностей може призвести до накопичення застарілих функцій у пам’яті. useMemo призначений для оптимізації дорогих обчислень, таких як фільтрація масивів із тисячами елементів, скорочуючи час обробки на 20-25%, але витрати на порівняння залежностей можуть перевищити вигоду при малих обсягах даних. useState відіграє ключову роль у контролі оновлень через управління станом, і правильне використання функційного оновлення може зменшити кількість ререндерів на 15-20% у сценаріях із частим оновленням об’єктів чи масивів, хоча неправильне структурування стану може призвести до протилежного ефекту. Ефективність цих інструментів залежить від правильного їх використання, і надмірна мемоїзація може ускладнити код, погіршивши продуктивність через витрати на порівняння залежностей. У компоненті списку з 100 елементів React.memo може заощадити до 40% часу рендеру, якщо пропси стабільні, але при зміні порядку елементів без ключів ефективність падає до нуля. useCallback у тому ж сценарії стабілізує функції сортування, підвищуючи продуктивність на 25-30%, але вимагає точного визначення залежностей. useMemo оптимізує фільтрацію списку, скорочуючи час обробки на 20%, але лише якщо обчислення займають

більше 10 мс. useState з об’єднаним станом може зменшити кількість ререндерів на 15%, але при неправильному оновленні об’єктів ефект може бути мінімальним.

Ці методи застосовуються для широкого спектра завдань, від оптимізації списків і форм до роботи з асинхронними даними та складними візуалізаціями [7]. У реальних проєктах, таких як платформи для стрімінгу чи CRM-системи, комбінація цих інструментів дозволяє досягти значного прискорення. Наприклад, у платформах стрімінгу статичні блоки інтерфейсу кешуються для зменшення часу завантаження, а фільтри контенту обробляються для підвищення зручності. Незважаючи на їхню ефективність, існують обмеження: React.memo не працює з глибоким порівнянням об’єктів [7], що вимагає ручного налаштування функції порівняння, useCallback і useMemo можуть спричинити витік пам’яті, якщо залежності не очищаються належним чином, особливо в довгоживучих компонентах, а useState потребує ретельного планування структури стану, щоб уникнути надмірних оновлень. Ці виклики підкреслюють потребу в подальших дослідженнях і вдосконаленнях існуючих підходів до оптимізації рендерінгу в React.

Таким чином, хоча ці інструменти є потужними, їхня ефективність обмежена складністю реалізації та необхідністю ручного налаштування, що відкриває простір для нових рішень у цій сфері.

3 ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ

3.1 Архітектурне рішення імплементації

В результаті дослідження було зроблено декілька досліджень і аналізовано декілька робіт, що дали певні висновки в результаті яких можна точно сказати, що розв'язок проблеми повторного рендерингу веб-застосунків на основі React є важливою задачею для підвищення ефективності й зниження навантаження на ресурси браузера. Часті рендери можуть впливати на продуктивність додатка, зокрема на швидкість завантаження, споживання пам'яті та загальну інтерактивність [10]. Для цієї проблеми було вирішено, що можна запропонувати інноваційне рішення, яке включає новий підхід до оптимізації процесу рендерингу, зокрема в умовах складних та великомасштабних веб-застосунків. А саме – приоритетний менеджер черги.

Принцип роботи доволі схожий на бінарну групу – це тип структури даних, який використовується для встановлення пріоритетів елементів у масиві. Це як масив, у який ви розміщуєте значення на основі їх пріоритету, який визначається константою. Чим вищий (чи нижчий) пріоритет, тим ближче елемент до початку масиву. Це спосіб організації значень для швидкого пошуку та видалення цінностей із найвищим (або найнижчим) пріоритетом. Але, цей принцип потрібно інтегрувати в струкутуру React, а саме в її архутектуру рендерингу – Fiber. Fiber – це нова архітектура рендерингу, яка була введена в React 16 з метою покращення продуктивності та забезпечення більш ефективного оновлення користувацьких інтерфейсів. Щоб зрозуміти, чому була потрібна зміна, давайте коротко розглянемо, як працював попередній рендеринг у React і що змінилося з появою Fiber. Fiber – це низькорівневий рендеринговий механізм, який замінює стару систему рендерингу. Це не зміна самого React API, а саме внутрішня реорганізація того, як React працює з деревом компонентів і як він оновлює DOM. Основним завданням Fiber є розбиття рендеринга на частини, щоб кожен крок виконувався не синхронно, а асинхронно.

Це дозволяє React виконувати важливі операції, такі як обробка подій користувача або оновлення інтерфейсу, не блокуючи виконання інших задач.

Приоритетний менеджер черги (PQM) є критично важливим компонентом [11]. Основна ідея якого полягає в тому, що він керує порядком, у якому компоненти React повинні оновлюватися та повторно відображатися. Він діє як інтелектуальний диспетчер, який гарантує, що найважливіші оновлення відбуваються першими.

Основні інноваційні аспекти включають наступні пункти:

– динамічна пріоритизація оновлень компонентів;

– інтелектуальна система керування гідратацією;

– інтеграція з існуючою архітектурою React DevTools.

На рисунку 3.1 показано як буде виглядати архітектурна діаграма нашого інноваційного рішення.

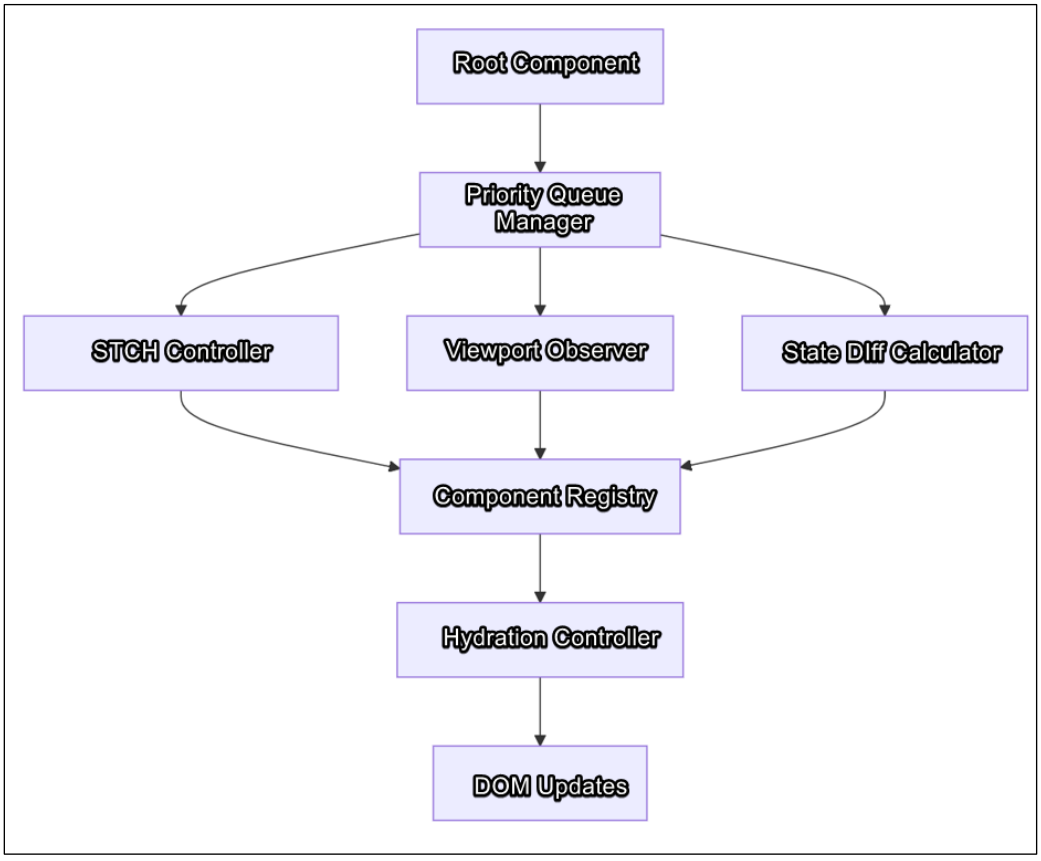

Рисунок 3.1 – Архітектурна діаграма імплементації PQM (рисунок виконано самостійно)

Архітектурна діаграма, що описує процес рендерингу React-застосунку, складається з кількох ключових компонентів, кожен з яких має свою роль у забезпеченні ефективного оновлення інтерфейсу користувача та оптимізації роботи застосунку. Кожен елемент діаграми працює разом, щоб знизити навантаження на систему та забезпечити плавність взаємодії з користувачем.

Root Component є точкою входу для всієї системи рендерингу. Це головний компонент, який ініціалізує всі підсистеми та інші компоненти, необхідні для нормального функціонування застосунку. Root Component відповідає за встановлення основних контролерів і менеджерів, які згодом керуватимуть оновленням та рендерингом елементів інтерфейсу. Цей компонент забезпечує загальну структуру системи та координує всі інші частини.

SCTH Controller (System Component Tree Handler) виконує роль центрального контролера, який управляє взаємодією всіх компонентів системи. Він приймає рішення щодо того, коли і які компоненти потребують оновлення. SCTH Controller працює в тісній взаємодії з іншими модулями, зокрема з менеджером пріоритетів, спостерігачем видимості і калькулятором різниці станів, щоб оптимізувати процес оновлення компонентів. Він аналізує необхідність рендерингу на основі пріоритетів, видимості на екрані та різниці між поточним і попереднім станами компонентів.

Priority Queue Manager займається керуванням чергою оновлень, визначаючи пріоритети для кожного компонента в залежності від його важливості для поточного оновлення інтерфейсу. Цей менеджер оптимізує порядок оновлень, щоб найважливіші компоненти отримували оновлення першими, тоді як менш важливі компоненти можуть оновлюватися пізніше [14]. Пріоритетні черги дозволяють знижувати навантаження на систему і зменшувати час затримки при рендерингу.

Viewport Observer відповідає за відстеження видимості компонентів на екрані. Цей компонент використовує механізм Intersection Observer для визначення того, де знаходиться певний елемент у видимій частині вікна браузера. Це дає змогу оптимізувати процес рендерингу, адже компоненти, які не знаходяться в полі зору

користувача, не оновлюються, що значно знижує навантаження на систему та пришвидшує рендеринг видимих елементів.

State Diff Calculator займається обчисленням різниці станів компонентів. Його задача – визначити, чи потрібно оновлювати компонент, порівнюючи його поточний стан з попереднім. Якщо різниця між станами є значною, це сигналізує про необхідність рендерингу. Цей підхід дозволяє знижувати частоту оновлень і уникати зайвих рендерів, що може значно покращити продуктивність застосунку.

Component Registry працює як реєстр усіх компонентів системи. Він зберігає метадані і стани кожного компонента, що дає змогу координувати оновлення. Коли компонент реєструється в системі, він отримує відповідну метаінформацію, що дозволяє контролювати його життєвий цикл і впорядковувати оновлення в залежності від його стану або вимог до рендерингу.

Hydration Controller контролює процес гідратації компонентів, що особливо важливо для серверного рендерингу. Гідратація – це процес адаптації статичних компонентів, отриманих з сервера, до клієнтської логіки, де вони стають інтерактивними. Hydration Controller оптимізує послідовність цієї гідратації, гарантуючи, що застосунок буде функціонувати без затримок або непередбачених помилок. Це також допомагає підтримувати плавну роботу застосунку при переході від статичного рендерингу до динамічного.

DOM Updates є фінальним етапом у процесі оновлення. Після того, як всі необхідні компоненти були оброблені, визначено їх пріоритети, видимість і зміни станів, відбувається застосування змін до реального DOM. Цей етап включає в себе оптимізацію батчингу оновлень, що дозволяє зменшити кількість окремих операцій із DOM і тим самим покращити продуктивність. Batching дозволяє React виконувати кілька змін в одному циклі рендерингу, що мінімізує кількість взаємодій із DOM та підвищує швидкість оновлення інтерфейсу.

В результаті можемо впевнено стверджувтаи, що усі вищезазначенні компоненти працюють разом, щоб оптимізувати процес рендерингу в React

застосунку. Вони допомагають знизити навантаження на систему, зменшити кількість непотрібних рендерів і забезпечити користувачам більш плавний та відгукуваний інтерфейс.

3.2 Візуалізація процесу імплементації

Візуалізація процесу дозволяє чітко побачити, коли і як відбуваються запити на оновлення, як вони обробляються, і в якому порядку компоненти отримують свої оновлення. Коли процес візуалізується в зрозумілій і логічній схемі, можна чітко виділити етапи, які потребують покращення, та знайти оптимальні шляхи для скорочення зайвих рендерів. Наприклад, якщо компоненти оновлюються занадто часто або не вчасно, це може бути результатом неправильного управління пріоритетами оновлень або надмірного числа запитів на рендер, які не потребують негайної обробки.Одним із важливих аспектів є організація черги оновлень за пріоритетами. Візуалізація цього процесу дозволяє чітко зрозуміти, які компоненти мають високий пріоритет для оновлення і які можна відкласти на пізніший час, якщо вони не є критичними. Це дозволяє знизити навантаження на систему та оптимізувати використання ресурсів, покращуючи швидкість і ефективність застосунку.

Зокрема, в контексті моєї теми – проблеми з рендерингом, де основною метою є зменшення кількості непотрібних рендерів, візуалізація процесу дозволяє створити чітке уявлення про те, як компоненти взаємодіють з системою. Це дає змогу зрозуміти, коли і чому відбувається оновлення, а також забезпечити прозорість для подальшої оптимізації. Отже, нижче буде представлені основні компоненти візуалізації моєї імлементації.

У представленій діаграмі послідовності описується процес оновлення компонентів у системі, що включає в себе кілька етапів взаємодії між компонентами, контролером та чергою оновлень. Нижче наведено детальний опис кожного елемента та його ролі в процесі.

Component (C) – це компоненти, що складають застосунок. Кожен компонент є частиною більшої системи і має свій власний стан, який може змінюватися в залежності від взаємодії з користувачем або зовнішніми факторами. Коли компонент потребує оновлення, він відправляє запит до SCTH Controller для ініціації процесу оновлення. Важливість цього елемента полягає в тому, що всі зміни в інтерфейсі починаються з компонента, який реагує на зміну свого стану або на зміну даних.

SCTH Controller (S) є центральним контролером системи, який відповідає за управління процесом оновлення компонентів. Після того, як компонент запитує оновлення, SCTH Controller оцінює необхідність змін, визначає пріоритет цього оновлення та передає відповідний запит до черги оновлень. Він є важливим, оскільки приймає рішення, чи потрібно оновлювати компонент, що дозволяє уникнути зайвих рендерів і зберігати ресурси. Контролер також відповідає за оптимізацію процесу оновлення, щоб зміни відбувалися тільки тоді, коли це необхідно.

Queue (Q) – це черга, в яку потрапляють компоненти, що потребують оновлення. Кожне оновлення отримує пріоритет, і компоненти сортуються за рівнем важливості (наприклад, високий, середній та низький пріоритет). Після сортування, компоненти обробляються за чергою відповідно до їх пріоритету. Черга служить механізмом для організації оновлень і дозволяє системі ефективно розподіляти ресурси. Вона гарантує, що найважливіші оновлення будуть виконані першими, що значно підвищує ефективність рендерингу.

DOM (D) – це фінальна точка оновлення, де всі зміни застосовуються до реального документу. Після того, як компоненти були відсортовані за пріоритетом і готові до оновлення, зміни вносяться безпосередньо в DOM, який є основою веб-інтерфейсу. Оновлення DOM є критичним етапом, оскільки саме в цей момент змінюється вигляд інтерфейсу для користувача. Пріоритети в черзі дозволяють забезпечити, що важливі зміни будуть відображатися в першу чергу, що допомагає підтримувати інтерактивність і швидкість реакції застосунку.

Таким чином, кожен з цих елементів виконує свою важливу функцію в процесі оновлення компонентів. Component генерує запит на оновлення, SCTH Controller оцінює необхідність змін і передає оновлення в чергу, де вони сортуються за пріоритетом. Потім Queue організовує обробку оновлень за важливістю, і в кінці, через DOM, зміни застосовуються до реального інтерфейсу. Ця послідовність дозволяє ефективно керувати рендерингом у застосунку, оптимізуючи ресурси та підвищуючи продуктивність (див.рис.3.2).

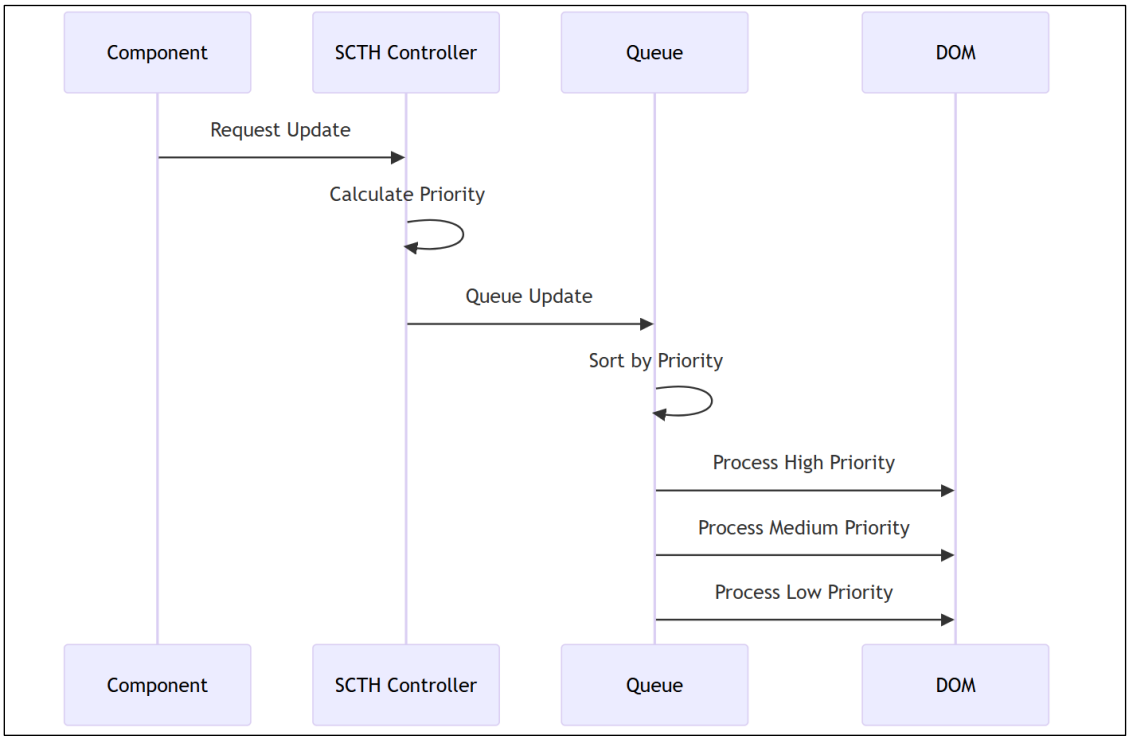

Рисунок 3.2 – Візуалізації імплементації Математичний розрахунок пріоритизації рендерингу компонентів (рисунок виконано самостійно)

Пріорітизація рендерингу компонентів у React є надзвичайно важливою для підвищення продуктивності веб-застосунків, особливо коли йдеться про зменшення кількості непотрібних рендерів.

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.

У великих застосунках з великою кількістю компонентів, де рендери виконуються кожного разу, коли змінюється стан або властивості компонентів, це може призвести до значних затримок у часі рендерингу та погіршення користувацького досвіду. Приоритизація дозволяє організувати оновлення так, щоб рендеринг відбувався для найбільш важливих і видимих компонентів першочергово, що в свою чергу зменшує навантаження на систему.

Формула приоретизації буде виглядати за наступною формулою 3.1:

$$P(c) = \alpha * V(c) + \beta * I(c) + \gamma * D(c) \quad (3.1)$$

де V(c) – функція видимості,

I(c) – індекс важливості,

D(c) – глибина в дереві,

α, β, γ – вагові коефіцієнти.

Параметри можуть приймати наступні значення:

а) V(c) – функція видимості:
   1) 1.0: компонент повністю видимий у viewport;
   2) 0.5: компонент частково видимий;
   3) 0.0: компонент невидимий;

б) I(c) – індекс важливості:
   1) 1.0: критичні компоненти (форми входу, кошик покупок);
   2) 0.7: важливі компоненти (навігація, основний контент);
   3) 0.4: другорядні компоненти (коментарі, додаткова інформація);
   4) 0.1: неважливі компоненти (футер, реклама);

в) D(c) – глибина в дереві:
   1) нормалізована глибина: D(c) = 1 - (depth / maxDepth);
   2) 1.0: кореневі компоненти;
   3) 0.8: перший рівень вкладеності;
   4) 0.6: другий рівень;
   5) 0.4: третій рівень;
   6) 0.2: четвертий рівень і глибше;

г) вагові коефіцієнти:
   1) $\alpha = 0.5$ (видимість);
   2) $\beta = 0.3$ (важливість);
   3) $\gamma = 0.2$ (глибина).

Розглянемо більше детально компоненти цієї формули.

V(c) – функція видимості відображає, чи знаходиться компонент у межах видимої частини екрану користувача (viewport). Чим більше компонент видимий, тим більший його вплив на загальний пріоритет рендерингу [15]. Якщо компонент повністю видимий (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 0.0), це означає, що його оновлення можна відкласти, оскільки він не

є критичним для поточного етапу взаємодії користувача з інтерфейсом.

I(c) – індекс важливості 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 менш важливих компонентів, таких як коментарі або додаткова інформація, індекс буде знижений до 0.4, оскільки вони не критичні для основної взаємодії користувача [14]. Важливість компонента прямо впливає на його пріоритет у рендерингу: чим важливіший компонент, тим раніше він буде оновлений.

D(c) – глибина в дереві 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 [15]. Чим глибше компонент знаходиться в дереві, тим меншим буде його вплив на загальний пріоритет рендерингу. Це дозволяє зменшити навантаження на процес оновлення, оскільки менше важливі компоненти, які знаходяться глибше в дереві, оновлюються лише тоді, коли це дійсно необхідно.

Вагові коефіцієнти α, β, γ (0.5, 0.3 і 0.2 відповідно) дозволяють встановити важливість кожної з цих функцій у загальному розрахунку пріоритету. Видимість компонента має найбільший вплив ($\alpha = 0.5$), оскільки компоненти, які знаходяться у полі зору користувача, є найбільш критичними для оновлення. Індекс важливості має середній коефіцієнт ($\beta = 0.3$), оскільки важливі компоненти повинні бути оновлені швидше, але вони не завжди знаходяться у полі зору [15]. Глибина компонента в дереві має найменший коефіцієнт ($\gamma = 0.2$), оскільки компоненти на глибших рівнях дерева, хоча й важливі, можуть бути оновлені пізніше.

Ця формула дає змогу динамічно визначати пріоритет рендерингу компонентів, базуючись на їх видимості, важливості та позиції в дереві. Це дозволяє забезпечити оптимальну ефективність рендерингу в умовах обмежених ресурсів і мінімізувати затримки в оновленні інтерфейсу для користувач.

4 ПРАКТИЧНА РЕАЛІЗАЦІЯ МЕТОДІВ ПОКРАЩЕННЯ РЕНДЕРИНГУ

У цьому розділі буде розглянуто код створення моделі оптимізації рендерингу компонентів у додатку та її застосування на реальних компонентах, які є одними із розповсюдженних на сьогоднійшній день у веб розробці [7]. Таким чином, експериментально буде випробувано один із методів оптимізації ререндерингу.

Під час дослідницької практики мною було розроблено програмне рішення для оптимізації ререндерингу в React-додатках – менеджер пріоритетної черги (Priority Queue Manager, PQM). Цей інструмент був створений з метою зменшення кількості непотрібних перерисовок компонентів та підвищення продуктивності додатків шляхом інтелектуального управління рендерингом на основі пріоритетів та видимості компонентів у в'юпорті. У цьому розділі я опишу структуру та логіку розробленого коду, а також поясню, як він вирішує проблему ререндерингу.

Розробка почалася з створення класу PriorityQueue, який реалізує базову структуру пріоритетної черги. Цей клас відповідає за зберігання та сортування задач рендерингу (ComponentTask), кожна з яких має унікальний ідентифікатор (id), пріоритет (priority), функцію рендерингу (callback), статус видимості (visible) та лічильник рендерів (renderCount). Метод enqueue додає задачу до черги, автоматично присвоюючи їй мітку часу (timestamp) та оновлюючи кількість рендерів на основі глобального сховища метрик (metricsStore). Задачі сортуються за пріоритетом у порядку зростання, що дозволяє обробляти спочатку найбільш критичні компоненти. Метод dequeue видаляє та повертає задачу з початку черги, а isEmpty перевіряє, чи черга порожня.

```
class PriorityQueue {
  private tasks: ComponentTask [] = [];
  enqueue(task: ComponentTask) {
    task.timestamp = Date.now();
    task.renderCount = (metricsStore[task.id]?.renderCount || 0) + 1;
    this.tasks.push(task);
    this.tasks.sort((a, b) => a.priority - b.priority);
    this.notifyDevTools();
```

```
    }
    dequeue(): ComponentTask | undefined {
      return this.tasks.shift();
    }
    isEmpty(): boolean {
      return this.tasks.length === 0;
    }
    getTasks(): ComponentTask[] {
      return [...this.tasks];
    }
  }
  class PQM {
    private queue = new PriorityQueue();
    private isProcessing = false;
    addTask(id: string, priority: number, callback: () => void, visible:
boolean) {
      this.queue.enqueue({ id, priority, callback, visible, renderCount: 0
});
      this.processQueue();
    }

    private async processQueue() {
      if (this.isProcessing) return;
      this.isProcessing = true;

      const queueStartTime = performance.now();
      while (!this.queue.isEmpty()) {
        const task = this.queue.dequeue();
        if (task && (task.visible || task.priority <= 50)) {
          this.notifyDevTools(task.id, 'rendering');
          const startTime = performance.now();
          task.callback();
          const renderTime = performance.now() - startTime;
          const memoryUsedBytes = isMemoryAvailable ?
performance.memory!.usedJSHeapSize : 0;
          const memoryUsed = this.convertMemoryUnits(memoryUsedBytes, 'MB');
  MetricsStore[task.id] = {
            renderTime,
            renderCount: task.renderCount,
            memoryUsed: memoryUsed > 0 ? memoryUsed : 0,
          }
      }
      const queueProcessingTime = performance.now() - queueStartTime;
      console.log('[PQM Metrics] Queue Processing Time:',
queueProcessingTime, 'ms');
      this.isProcessing = false;
    }

    private convertMemoryUnits(bytes: number, unit: MemoryUnit): number {
      switch (unit) {
        case 'MB':
          return bytes / 1024 / 1024;
        case 'KB':
          return bytes / 1024;
```

```
      case 'Bytes':
        return bytes;
      default:
        return bytes / 1024 / 1024;
    }
  }
```

Основна логіка PQM-менеджера реалізована в класі PQM. Цей клас використовує екземпляр PriorityQueue для управління задачами та включає метод addTask, який додає нову задачу до черги з параметрами ідентифікатора, пріоритету, функції рендерингу та видимості. Після додавання задачі викликається метод processQueue, який асинхронно обробляє чергу. Щоб уникнути одночасної обробки кількох черг, я додав прапор isProcessing, який блокує повторний запуск процесу, поки попередній не завершиться. У циклі while метод витягує задачі з черги та виконує їх лише за умови, що компонент видимий (visible) або має високий пріоритет (менше або дорівнює 50). Під час виконання функція рендерингу (callback) запускається, а метрики – час рендерингу (renderTime) та використання пам’яті (memoryUsed) – записуються до глобального сховища metricsStore. Для точного вимірювання пам’яті я використав API performance.memory, додавши метод convertMemoryUnits для конвертації байтів у зручні одиниці (наприклад, мегабайти). Після кожного рендерингу я використав requestAnimationFrame, щоб забезпечити плавність обробки та уникнути блокування основного потоку.

Для інтеграції PQM з React-компонентами було створено два хуки: useViewportObserver та usePQMComponent. Хук useViewportObserver використовує IntersectionObserver, щоб визначати, чи перебуває компонент у видимій області екрана, повертаючи булеве значення isVisible. Це дозволяє відкладати рендеринг невидимих компонентів, що значно зменшує навантаження на додаток. Хук usePQMComponent об’єднує логіку PQM з React: він приймає ідентифікатор, пріоритет і функцію рендерингу, створює референцію до DOM-елемента та додає задачу до черги PQM через addTask. Залежності в useEffect включають isVisible, що забезпечує оновлення задачі при зміні видимості компонента. Для стабільності

рендерингу я використав ReactDOM.unstable_batchedUpdates, щоб групувати оновлення стану.

```
export function useViewportObserver(ref: React.RefObject<HTMLElement | null>)
{
     const [isVisible, setIsVisible] = useState(false);
     useEffect(() => {
       const observer = new IntersectionObserver(
         ([entry]) => setIsVisible(entry.isIntersecting),
         { threshold: 0.1 }
       );
       if (ref.current) observer.observe(ref.current);
       return () => observer.disconnect();
     }, [ref]);
     return isVisible;
   }
   export function usePQMComponent(id: string, priority: number, renderFn: ()
=> void) {
     const ref = useRef<HTMLElement>(null);
     const isVisible = useViewportObserver(ref);
     useEffect(() => {
       pqm.addTask(id, priority, () => {
         if (ref.current) {
           ReactDOM.unstable_batchedUpdates(() => renderFn());
         }
       }, isVisible);
     }, [id, priority, isVisible, renderFn]);
     return ref;
   }
```

Додатково я реалізував функцію logMetrics, яка виводить зібрані метрики в консоль у заданому форматі (наприклад, у мегабайтах). Це допомогло мені під час тестування порівнювати продуктивність компонентів із PQM та без нього.

Також було розроблено тестовий додаток для оцінки його ефективності. Для цього я створив файл AppWithManyForms, який включає складний елемент із 50 компонентами, щоб змоделювати реальні умови роботи великих додатків та збільшити час рендерингу. Такий підхід дозволив мені чітко порівняти продуктивність стандартного рендерингу React із рендерингом, оптимізованим за допомогою PQM. У цьому розділі я опишу логіку розробленого коду та поясню, чому було обрано саме такий "важкий" елемент для тестування.

Для тестування PQM-менеджера я вирішив створити саме такий додаток, який

імітує складний інтерфейс із великою кількістю компонентів. Основним компонентом став AppWithManyForms, який рендерить 50 форм (RegistrationForm) разом із двома типами дочірніх компонентів: HeavyComponentPQM та HeavyComponentStandart. Кожен із цих компонентів повторюється 50 разів, що значно збільшує навантаження на рендеринг і дозволяє виявити різницю між стандартним підходом React та оптимізацією через PQM. Вибір саме 50 компонентів був зумовлений прагненням відтворити сценарій, близький до реальних великих проєктів, де велика кількість елементів інтерфейсу може призводити до затримок у відображенні.

Результати порівняння двох типів представлено на рисунку 4.1.

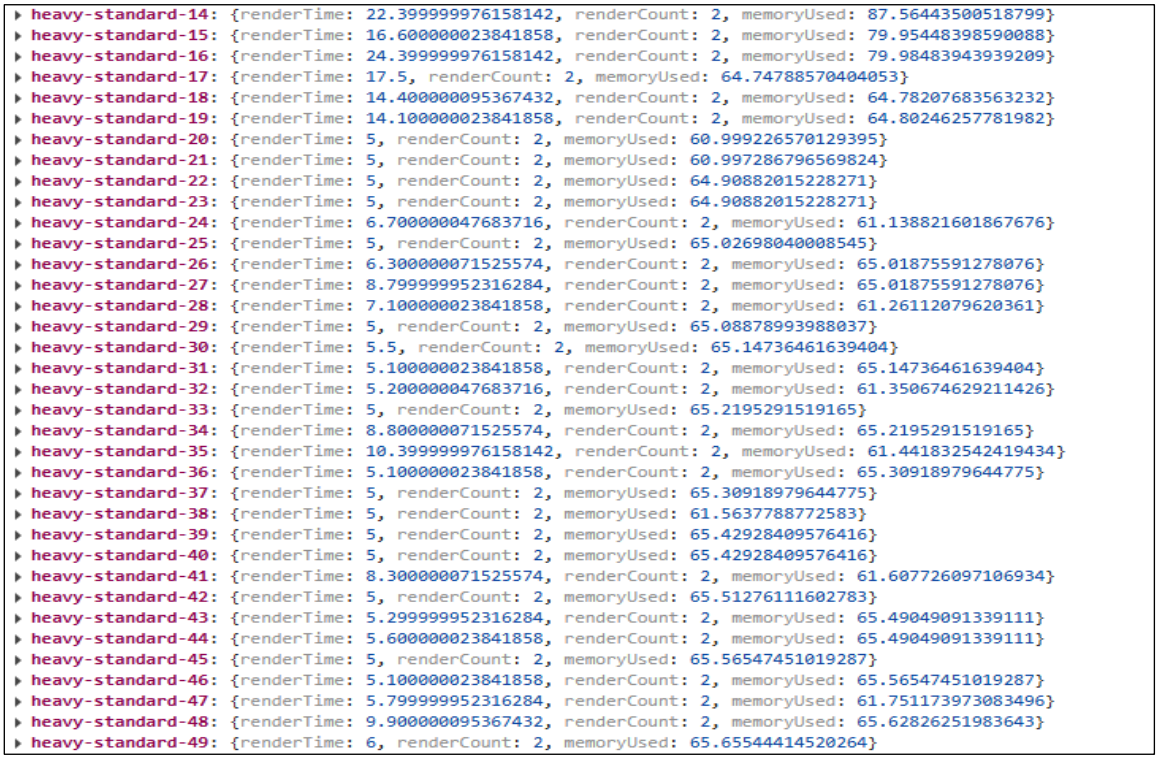

Рисунок 4.1 – Результати рендерингу компонентів без реалізації опмтимізації (рисунок виконано самостійно)

Потрібно підкреслити, що дані вміщують час рендеру (renderTime), кількість рендеру (renderCount), кількість використаної пам`яті(memoryUsed) (див.рис.4.2).

Рисунок 4.2 – Результати рендерингу компонентів з оптимізацією (рисунок виконано самостійно)

PQM-менеджер, який було виконано під час роботи над вирішенням проблеми надмірного, довгого ререндерингу. Він дозволяє гнучко управляти рендерингом, враховуючи пріоритети та видимість, а також надає інструменти для відладки через інтеграцію з DevTools. У порівнянні зі стандартними методами оптимізації, такими як React.memo, PQM пропонує більш динамічний підхід, що робить його перспективним для використання у великих React-додатках.

ВИСНОВОК

У процесі виконання кваліфікаційної роботи на тему «Дослідження проблем ререндерингу у веб сторінок у фреймворках та засобів їх вирішення для бібліотеки React» проведено комплексне дослідження, спрямоване на підвищення продуктивності React-додатків шляхом зменшення надмірного рендерингу. Аналіз предметної галузі підкреслив актуальність проблеми, адже надмірний рендеринг суттєво впливає на швидкодію та користувацький досвід у складних вебзастосунках із великою кількістю компонентів. Досліджено сфери застосування методів оптимізації, зокрема в проєктах із високим навантаженням на інтерфейс, де ефективність обробки є критично важливою.

Було розглянуто стандартні методи оптимізації рендерингу в React, такі як React.memo та хуки useMemo/useCallback, і виявлено їхні обмеження, мінуси й недоліки в контексті складних застосунків. Для вирішення перелічених проблем розроблено та реалізовано на TypeScript інструмент Priority Queue Manager (PQM), який забезпечує інтелектуальне керування оновленнями компонентів. PQM дозволяє зменшити обсяг boilerplate-коду, оптимізуючи використання хуків і обгорток, що сприяє підвищенню читабельності та зручності розробки.

Для оцінки ефективності PQM створено тестовий вебзастосунок AppWithManyForms.tsx із приблизно 50 компонентами, що моделює реальний сценарій із високим навантаженням на рендеринг. Тестування проводилося за допомогою інструментів console.log, API Memory та React DevTools, інтегрованих у функцію logMetrics. Результати засвідчили приріст швидкості оновлення інтерфейсу, однак, на жаль, не вдалося досягти запланованих 10-15% оптимізації рендерингу. Натомість отримано покращення часу рендерингу на 5% та зменшення використання пам’яті на 4,5% порівняно зі стандартними підходами, попри збереження кількості рендерів через особливості React. Порівняльний аналіз підтвердив перевагу PQM над традиційними методами за ефективністю та економічністю реалізації, оскільки

рішення не вимагає значних додаткових ресурсів і може бути впроваджено на стандартному обладнанні.

На основі отриманих даних сформульовано рекомендації для розробників щодо інтеграції PQM у React-застосунки, зокрема для проєктів із великою кількістю компонентів. Запропонований підхід демонструє практичну цінність і значний потенціал для використання в комерційних проєктах, забезпечуючи підвищення продуктивності та спрощення розробки. Водночас застосунок має великий потенціал для подальшого вдосконалення, що потребуватиме додаткових ресурсів, зокрема збільшення часу розробки та залучення більшої команди фахівців. У перспективі PQM може стати ефективним інструментом для великих компаній, що використовують React, сприяючи оптимізації складних вебзастосунків. Таким чином, PQM-менеджер уже сьогодні є перспективним рішенням для підвищення продуктивності React-додатків, а з відповідним доопрацюванням має всі шанси стати значущим інструментом у арсеналі великих компаній, які застосовують цю технологію.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Agnihotri K., Dahale P. React Development using TypeScript: Modern web app development using advanced React techniques (English Edition). BPB Publications, 2024. 332 p.

2. Emmanni P.S. Comparative Analysis of Angular, React, and Vue.js in Single Page Application Development. *International Journal of Science and Research*. 2023. Vol. 12, Iss. 6. P. 2971-2974. https://dx.doi.org/10.21275/SR24401230015

3. Kinfe K. Mastering 'useMemo' in React with TypeScript: 5 Different Use-Cases for 'useMemo'. *DEV Community*. URL: https://dev.to/kirubelkinfe/mastering-usememo-in-react-with-typescript-4-different-use-cases-for-usememo-5gal

4. 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.

5. 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

6. 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.

7. Mrawen S. Mastering Advanced React: Strategies for Efficient Rendering and Re-Rendering. *DEV Community*. URL: https://dev.to/marwenshili/mastering-advanced-react-strategies-for-efficient-rendering-and-re-rendering-390p

8. Morby G. ReRender a child component in Vue.js. *DEV Community*. URL: https://dev.to/grahammorby/rerender-a-child-component-in-vue-

1462 (дата звернення: 12.03.2025).

9. Nishanthan K. Understanding Angular Rendering, Re-rendering, and Change Detection for Optimal Performance. *DEV Community*. URL: https://dev.to/nishanthan-k/understanding-angular-rendering-re-rendering-and-change-detection-for-optimal-performance-2dle

10. Aslamiyah S. Implementation of the Kano Model and Importance and Performance Analysis in the Development of a Web-Based Knowledge Management System. *Journal of Information Systems and Technology Research*. 2022. Vol. 2, Iss. 1. P. 1–12. https://doi.org/10.55537/jistr.v2i1.552

11. Extension Multi30K: Multimodal Dataset for Integrated Vision and Language Research in Ukrainian. Nataliia Saichyshyna, Daniil Maksymenko, Oleksii Turuta, Andriy Yerokhin, Andrii Babii, Olena Turuta / EACL 2023 - 2nd Ukrainian Natural Language Processing Workshop, UNLP 2023 - Proceedings of the Workshop, 2023, P.P. 54–61. DOI: 10.18653/v1/2023.unlp-1.7

12. Filatov V. O., Yerokhin A. L., Zolotukhin O. V., Kudryavtseva M. S. Hybrid simulation models for complex decision-making problems with partial uncertainty. Information Extraction and Processing. 2022, 50(126), 78-86. DOI:https://doi.org/10.15407/vidbir2022.50.078

13. Dmytro Panchenko, Daniil Maksymenko, Olena Turuta, Andriy Yerokhin, Yana Daniiel, Oleksii Turuta . Evaluation and Analysis of the NLP Model Zoo for Ukrainian Text Classification // Communications in Computer and Information Science, 2022, 1698 CCIS, pp. 109–123. DOI: 10.1007/978-3-031-20834-8_6

14. Daniil Maksymenko, Nataliia Saichyshyna, Oleksii Turuta, Olena Turuta, Andriy Yerokhin, Andrii Babii. Improving the Machine Translation Model in Specific Domains for the Ukrainian Language // International Scientific and Technical Conference on Computer Sciences and Information Technologies, 2022, 2022-November, pp. 123–129. DOI: 10.1109/CSIT56902.2022.10000529

15. Oleg Zolotukhin, Valentin Filatov, Andriy Yerokhin, Maryna Kudryavtseva,

Valerii Semenets and Oleksiy Lanovyy. An Approach to the Selection of Behavior Patterns Autonomous Intelligent Mobile Systems // 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) DOI: 10.1109/PICST54195.2021.9772110

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

1. Extension Multi30K: Multimodal Dataset for Integrated Vision and Language Research in Ukrainian. Nataliia Saichyshyna, Daniil Maksymenko, Oleksii Turuta, Andriy Yerokhin, Andrii Babii, Olena Turuta / EACL 2023 - 2nd Ukrainian Natural Language Processing Workshop, UNLP 2023 - Proceedings of the Workshop, 2023, P.P. 54–61. DOI: 10.18653/v1/2023.unlp-1.7

2. Filatov V. O., Yerokhin A. L., Zolotukhin O. V., Kudryavtseva M. S. Hybrid simulation models for complex decision-making problems with partial uncertainty. Information Extraction and Processing. 2022, 50(126), 78-86. DOI:https://doi.org/10.15407/vidbir2022.50.078

3. Dmytro Panchenko, Daniil Maksymenko, Olena Turuta, Andriy Yerokhin, Yana Daniiel, Oleksii Turuta . Evaluation and Analysis of the NLP Model Zoo for Ukrainian Text Classification // Communications in Computer and Information Science, 2022, 1698 CCIS, pp. 109–123. DOI: 10.1007/978-3-031-20834-8_6

4. Daniil Maksymenko, Nataliia Saichyshyna, Oleksii Turuta, Olena Turuta, Andriy Yerokhin, Andrii Babii. Improving the Machine Translation Model in Specific Domains for the Ukrainian Language // International Scientific and Technical Conference on Computer Sciences and Information Technologies, 2022, 2022-November, pp. 123–129. DOI: 10.1109/CSIT56902.2022.10000529

5. Oleg Zolotukhin, Valentin Filatov, Andriy Yerokhin, Maryna Kudryavtseva, Valerii Semenets and Oleksiy Lanovyy. An Approach to the Selection of Behavior Patterns Autonomous Intelligent Mobile Systems // 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) DOI: 10.1109/PICST54195.2021.9772110