

6. Матвієнко О., Закутній С. Нечітка логіка в задачах визначення економічних параметрів виконання проєктів. *Сучасний стан наукових досліджень та технологій в промисловості*. 2024. № 1(27). С. 96-108. URL: <https://doi.org/10.30837/ITSSI.2024.27.096>
7. Mazluma M., Günericpm A.F. PERT and Project Management With Fuzzy Logic Technique an implementation on a business. *Procedia - Social and Behavioral Sciences*. 2015. № 210. P. 348–357. URL: <https://doi.org/10.1016/j.sbspro.2015.11.378>
8. Dudnyk O., Sokolovska Z. Application of Fuzzy Expert Systems in IT Project Management. *Project Management - New Trends and Applications. IntechOpen*. 2023. URL: <http://dx.doi.org/10.5772/intechopen.102439>
9. Васильків Н., Дубчак Л., Турченко, І., Мінчук В. Визначення пріоритетності завдань ІТ-проєкту на основі нечіткої логіки. *Herald of Khmelnytskyi National University. Technical Sciences*, 2024. № 335 (1). С. 41-46. URL: <https://doi.org/10.31891/2307-5732-2024-335-3-6>
10. Kalibatienė D., Miliauskaitė J., Slotkienė A. Ontology and Fuzzy Theory Application in Information Systems: A Bibliometric Analysis. *Informatica*. 2024. Vol. 35, no. 3. P. 557-576. URL: <https://doi.org/10.15388/24-INFOR557>
11. Білова Т. Г., Дьоміна В.М., Побіженко І.А., Остапенко О.О. Метод міркувань на прецедентах для підтримки прийняття рішень в гуманітарному реагуванні. *АСУ та прилади автоматизації*. 2024. № 180. С. 36–44. URL: <https://doi.org/10.30837/0135-1710.2024.180.036>

Надійшла до редколегії 28.03.2025 р.

Білова Тетяна Георгіївна, кандидат технічних наук, доцент, доцент кафедр ІУС та СТ ХНУРЕ, м. Харків, Україна, e-mail: tetiana.bilova@nure.ua, ORCID: <https://orcid.org/0000-0002-1085-7361> (науковий керівник здобувача вищої освіти Остапенко О.О.)

Побіженко Ірина Олександрівна, кандидат технічних наук, доцент, доцент кафедри ПІ ХНУРЕ, доцент кафедри цифрових комунікацій та інформаційних технологій ХДАК, м. Харків, Україна, e-mail: irina_pob@ukr.net, ORCID: <https://orcid.org/0000-0002-0723-1878>.

Остапенко Олена Олексіївна, здобувач вищої освіти, група УПГІТм-23-1, факультет комп'ютерних наук, ХНУРЕ, м. Харків, Україна, e-mail: olena.ostapenko@nure.ua, ORCID: <https://orcid.org/0009-0004-4146-4669>.

УДК 004.4'236

DOI: 10.30837/0135-1710.2025.184.090

А.Л. СРОХІН, 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-відмінністю, 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.



Рис. 1. Статистика 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.

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.

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 ефективніше 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 компонентів 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 розповсюджених сьогодні день у веброботі. 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 веброботки 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 пріоритизації 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:

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

де $V(c)$ – функція видимості; $I(c)$ – індекс важливості; $D(c)$ – глибина в дереві, визначається як $D(c) = 1 - (\text{depth} / \text{maxDepth})$; 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: $\alpha = 0.5$ (видимість); $\beta = 0.3$ (важливість); $\gamma = 0.2$ (глибина).

Видимість компонента має найбільший вплив ($\alpha = 0.5$), оскільки компоненти, які знаходяться у полі зору користувача, є найкритичнішими 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. Глибина компонента в дереві має найменший коефіцієнт ($\gamma = 0.2$), 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	7.37	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.

Перелік посилань:

1. Rippon C. Learn React with TypeScript. Packt Publishing, 2023. 474 p.
2. Agnihotri K., Dahale P. React Development using TypeScript: Modern web app development using advanced React techniques (English Edition). BPB Publications, 2024. 332 p.
3. 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. URL: <https://dx.doi.org/10.21275/SR24401230015>
4. Kinfе 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> (дата звернення: 12.03.2025).
5. 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. URL: <https://doi.org/10.13052/jwe1540-9589.21311>.
6. 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).

7. 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. URL: <https://doi.org/10.1109/ICDT57929.2023.10150911>.

8. 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> (дата звернення: 12.03.2025).

9. Morby G. ReRender a child component in Vue.js. *DEV Community*. URL: <https://dev.to/grahammorby/re-render-a-child-component-in-vue-1462> (дата звернення: 12.03.2025).

10. 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> (дата звернення: 12.03.2025).

11. 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. URL: <https://doi.org/10.55537/jistr.v2i1.552>

Надійшла до редколегії 25.03.2025 р.

Єрохін Андрій Леонідович, доктор технічних наук, професор, декан факультету комп'ютерних наук ХНУРЕ, м. Харків, Україна, e-mail: andriy.yerokhin@nure.ua, ORCID: <https://orcid.org/0000-0002-8867-993X> (науковий керівник здобувача вищої освіти Каменєва Д.В.)

Каменєв Дмитро Вікторович, здобувач вищої освіти, група ІПЗм-23-2, факультет комп'ютерних наук, ХНУРЕ, м. Харків, Україна, e-mail: dmytro.kameniev@nure.ua