Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Використання Web-workers для роботи із багатопоточністю у браузері

Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки»

Керівник курсової роботи
Глибовець А. М.

_____
(підпис)
"___" _____ 2022 р.

Виконала студентка
Барабуха М. М.
"___" _____ 2022 р.

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри інформатики,

Малашонок Г. І.

_____

(підпис)

“___” _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентці Барабусі Марії Максимівні факультету інформатики 3 курсу

ТЕМА Використання Web-workers для роботи із багатопоточністю у браузері

Індивідуальне завдання

Вступ

1 Огляд теоретичних відомостей

2 Розробка застосунку

3 Дослідження та порівняння обраних методів

Висновки

Список літератури

Додатки (за необхідністю)

Дата видачі “___” _____ 2022 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: Використання Web-workers для роботи із багатопоточністю у браузері

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	11.10.2021	
2.	Пошук тематичної літератури	11.11.2021	
3.	Ознайомлення з літературою	11.12.2021	
4.	Створення плану роботи	30.12.2021	
5.	Написання теоретичної частини роботи	20.01.2022	
6.	Подання першої версії записки науковому керівнику	25.01.2022	
7.	Розробка застосунку для дослідження	15.03.2022	
8.	Описання практичної частини роботи	15.04.2022	
9.	Проведення дослідження	01.05.2022	
10.	Аналіз дослідження та висновки	15.05.2022	
11.	Перегляд змісту роботи керівником	25.05.2022	
12.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	01.06.2022	

Студентка Барабуха М.М.

Керівник Глибовець А.М.

“____” _____

Зміст

Анотація

В роботі розглянуто два варіанти реалізації важких обчислень: через псевдо-багатопоточність та вебворкер. Дослідження були проведені на файлах формату CSV з різною кількістю рядків, для яких було визначено, який алгоритм відпрацює краще. Ефективність обраних алгоритмів визначається швидкістю обробки та загальним рівнем частоти оновленням кадрів для файлу певного розміру. Також в роботі розглядаються особливості роботи браузера та структура фреймів.

Вступ

Всім нам знайома ситуація, коли сайт довго завантажується. Інколи проблема полягає в швидкості підключення, інколи ж в навантаженні на браузер. На жаль, ми як розробники не можемо вплинути на інтернет-з'єднання, проте ми маємо можливість покращити продуктивність. Яким чином і на скільки ефективно це можна зробити, ми розглянемо в цій роботі.

На щастя, сьогодні ми маємо доволі широкий спектр технологій, що дозволяють нам впливати на роботу браузера. Наприклад, якщо в головному потоці відбувається багато подій, то можна частину з них винести в інший. Але тут потрібно знати і про обмеження другого потоку, і про те, наскільки це буде раціонально, і скільки таких потоків буде доцільно використати, і так далі. Проте, також є варіант, де ми можемо не створювати нічого нового і оптимізувати роботу всередині одного потоку за допомогою засобів, які вже вбудовані в браузер. Так, наприклад, ми можемо виконувати код в той час, коли браузер ще (або вже) не зайнятий промальовкою сторінки.

Як вирішити, коли і яку технологію варто використати? Чи не призведе це до ще більшого навантаження? Всі ці питання спонукають нас до більш детального вивчення можливостей браузера.

Метою нашої роботи є дослідження використання вебворкерів на прикладі завантаження файлів різного об'єму та встановлення меж, коли їх застосування є дійсно необхідним та не призводить до зворотного ефекту – зниженню частоти оновлення кадрів чи задовгого виконання; а також порівняємо його з іншими способами вирішення даної проблеми. Так ми зможемо оцінити ефективність існуючих методів на конкретних приладах, визначити, в яких ситуаціях який буде кращим.

1 Аналіз предметної області. Опис проблематики

1.1 Проблеми браузера. UX та FPS

Браузер має кілька потоків (основний, GUI-rendering і потік, який провокує події), але скрипт може виконуватися лише в головному, а тому однією з проблем браузера називають однопоточність [1], маючи на увазі саме виконання коду. В сучасних сайтах багато інтерактивності, анімацій тощо. Всі ці процеси потребують як часових затрат, так і ресурсних. Тож недивно, якщо через неоптимізований код, анімація на сайті може підвисати, бо браузер не встигатиме все обробити вчасно.

Якщо сайт довго вантажиться, то у користувачів з'являється відчуття, наче вони щось неправильно ввели чи натиснули, або сайт не працює. Доволі часто ж трапляються ситуації, коли без певної затримки обійтися просто неможливо – завантажуються якісь дані, повільне з'єднання тощо. Тому є певні рекомендації, як зробити це максимально зручним: для коротких затримок (до 10 секунд) використовують комбінування іконки очікування та маленького віконця з відсотком виконаної роботи, для довших – індикатор виконання чи інший зворотній зв'язок, який дає зрозуміти користувачу, що його запит «в процесі».

Для зовсім маленьких затримок (0.1-1.0 секунд) ніякого зворотного зв'язку надавати не потрібно, хоч користувач і помітить, що система відповідає з затримкою. В ідеалі ж ми хочемо, щоб дії оброблялися миттєво, тобто менше ніж за 0.1 секунду. В такому випадку, у користувача буде відчуття, що він безпосередньо маніпулює елементами інтерфейсу [2].

Також важливим показником для оцінки UX (зручності користування) є частота оновлення кадрів. Мабуть, найпоширенішою на сьогодні є 60Гц. Проте звідки взялося це число?

Історично склалося, що завдяки частоті електричних мереж було розроблено аналогове телебачення з частотою кадрів 50Гц (у більшості країн світу) та 60Гц (США, Канада, Південна Корея, Японія). Так як частота була доволі стабільною, було б логічно використовувати її для синхронізації.

Тому, якщо апаратне забезпечення допускає максимум 60 fps, то незалежно від того, скільки кадрів в секунду ми відтворюємо, лише 60 з них відобразяться на екрані протягом секунди. Але не в усіх дисплеях частота оновлення становить 60Гц. Наприклад, наразі доволі поширеними також є дисплеї з частотою 120 Гц. І так як браузер завжди намагається оновлюватися з тією ж частотою як і дисплей, то якщо на нашому пристрої частота кадрів становитиме 60 fps, то й в браузері максимум буде 60fps, в той час, як на іншому це може бути 90 fps, 120 fps чи навіть і більше [3].

В нашому дослідженні ми візьмемо за стандарт 60 Гц, так як цього цілком достатньо для забезпечення цілісності анімації. Тоді ми можемо порахувати, що 1 сек / 60 fps = 16.7 сек – тривалість одного фрейму; за цей час має виконатися як скрипт, так і промальовка кадру. Якщо ж не вкластися в дані часові обмеження, то від цього постраждає UX.

Тому, щоб картинка була плавною, розробник повинен розуміти, як це працює, та забезпечувати продуктивність у всьому: від прокрутки до швидкої реакції на натискання на екран. Основною ідеєю є те, що ми хочемо зменшити час рендерінгу (уникнути непотрібних паремальовок тощо) та залишити більше часу на обробку вводу користувача. Для цього потрібно навчитися розвантажувати основний потік браузера там, де це можливо і допустимо. Тому для початку розберемося, як працює сам рендерінг та головний потік браузера.

1.2 Анатомія фрейму

Побудова фрейму складається з багатьох процесів. Проте, у нашому дослідженні нас цікавить та частина, що відбувається в головному потоці, так як ми можемо безпосередньо втручатися у його роботу. Тож розберемося, що в ній відбувається.

Рисунок 1.1 - Головний потік браузера

Опишемо, за що відповідає кожен крок:

- *Обробка вхідних подій.* Вхідні дані передаються обробникам подій у головному потоці.
- *requestAnimationFrame*. Це ідеальне місце для візуальних оновлень на екрані. Тут ми збираємо всі потрібні зміни і обробляє їх пізніше.
- Розбір *HTML*. Аналізуємо весь новий HTML та створюємо нові DOM-елементи.
- *Перерахунок стилів*. Тут ми переобраховуємо всі нові чи змінені стилі. Це може бути як ціле дерево, так і частина.
- *Макет*. Обчислення геометричної інформації (позиція та розмір кожного елементу). Зазвичай це робиться для всього документа, і часто обчислювальні затрати є пропорційними розміру DOM.
- *Оновлення дерева макету*. Процес створення дерева макету та сортування елементів за глибиною.
- *Розмалювання.* Розфарбовує нові або оновленні елементи.

- *Компонування.* Обраховується вся інформація та передається в композиційний потік.
- *requestIdleCallback.* Якщо в головному потоці ще залишається певний час, то можна викликати *requestIdleCallback*, в якому виконується, наприклад, передача аналітичних даних [4].

Інколи немає необхідності оновлювати макет або перерозфарбовувати елементи, тож в деяких ситуаціях деякі кроки пропускаються.

Як ми можемо помітити, головний потік виконує багато роботи. У випадках, коли ми перевантажуємо його, і фрейм не встигає намалюватися вчасно, то наступний фрейм випадає (інколи навіть не один).

Картинка на екрані перестає бути плавною та цілісною, що візуально виглядає непривабливо для користувача. Чому так трапляється? Коли якийсь з кроків займає багато часу, то браузер не може перейти до виконання наступних, і процес «заморожується» в очікуванні завершення попереднього завдання. Таке явище називається блокуванням. Що саме може викликати такий ефект? Зазвичай, це може статися, коли в обробці вхідних подій (*input event handler*) виконується певні складні обчислення. Так як цей крок є одним з перших, то його затримка призводить до гальмування всього процесу і, в кінці кінців, до того, що на промальовку просто не вистачить часу. Це відбувається, коли ми даємо обробнику задачі, які він виконувати не мав би. Наприклад, аналізувати певні дані, щось завантажувати тощо – ті речі, які варто було б передати на виконання іншому процесу.

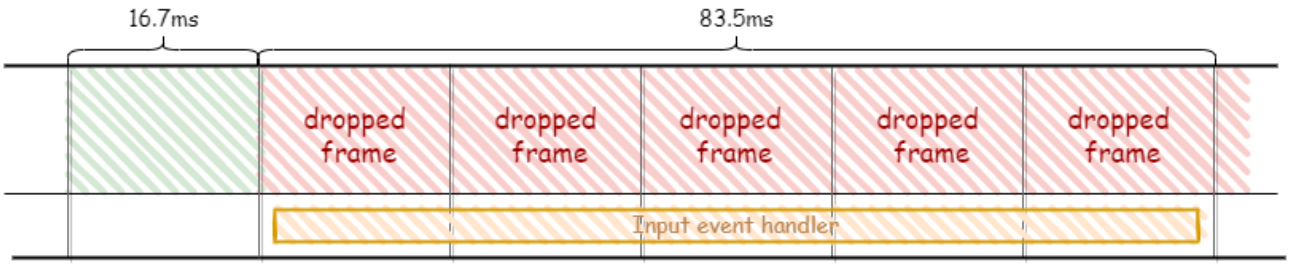


Рисунок 1.2 - Пропущений фрейм

З усього, що було зазначено вище, ми можемо зробити висновок, що складні обчислення у браузері можуть викликати низку проблем, які пов’язані з тим, що браузер має лише один потік. Які ж є для цього рішення?

Перший варіант – розбити великі задачі на менші (і мова йде не тільки про декомпозицію): наприклад, ми маємо цикл на мільйон ітерацій, розуміємо, що такий код буде виконуватися довго, і в браузера не буде можливості оброблять інші задачі, тож, як рішення, можна за одне виконання скрипту проходитися не по всім повторам, а лише по певній кількості, яка не буде затримувати виконання інших кроків створення фрейму. Це буде реалізовано так, що за один кадр відбудеться частина ітерацій, а решта виконається за наступні; таким чином, ми уникнемо видимих затримок.

Згадаємо про схему наведену вище та розглянемо, які з процесів ми можемо використати. Для нас корисними стануть *requestAnimationFrame()* і *requestIdleCallback(),* адже вони зможуть забезпечити нам таке виконання «частинками». Саме ці функції ми будемо використовувати для реалізації псевдо-багатопоточності.

Другий варіант напрошується сам собою: створити другий потік. JavaScript надає нам можливість запускати файли з кодом (з певним обмеження щодо функціоналу) в іншому потоці використовуючи вебворкери. Таким чином ми розвантажимо основний потік, даючи йому змогу виконувати його основні функції, а складні обчислення доручити вебворкеру.

2 Методи вирішення

2.1 Псевдо-багатопоточність

Чому ми говоримо не про багатопоточність, а саме про псевдо-багатопоточність? Як було зазначено вище, код може виконуватися лише в основному потоці. Якщо ж ми захочемо використати, наприклад, асинхронні методи, то вони будуть також виконуватися в цьому ж потоці, але після виконання синхронного коду [5]; тобто браузер не створить окремий потік, а побудує чергу викликів. Ми ж будемо досягати псевдо-багатопоточності, використовуючи 2 методи: requestAnimationFrame() і requestIdleCallback(), які хоч і не є асинхронними, про те мають схожий принцип дії. Далі – детальніше.

2.1.1 requestAnimationFrame()

Цей метод повідомляє браузеру, що ми хочемо виконати анімацію і запрошує запланувати перемальовування в наступному кадрі. Як параметр метод отримує функцію, яка буде викликана перед перемальовуванням. В методі, що буде здійснений до зміни кадру, є один параметр – часова мітка, яка буде передана під час виклику задачі з черги. Якщо таких задач за один фрейм буде кілька, то вони матимуть однакову часову мітку, хоч і будуть фізично виконуватися в різний час. Це пов’язано з тим, що час вказано з точністю в 1 мс [6].

requestAnimationFrame() - це чудове місце для виконання коду, що не стосується обробки користувацького вводу, перед безпосередньою перемальовкою. Проте, є певні обрахунки, яких варто уникати в методі, так як це може призвести до примусової синхронізації макету - явища, коли зміна макету трапляється раніше, ніж переобрахунок стилів.

Чому так відбувається? В *requestAnimationFrame()* ми працюємо з тими значеннями стилів, які були на момент завершення попереднього кадру. Проте, якщо ми змінимо стиль у виклику перед тим, як звернутися до нього, то браузеру потрібно буде спочатку застосувати цю зміну, що й призведе до оновлення макету, а потім дізнатися її значення.

Як приклад, хочемо обрахувати функцію, яка задає переміщення об’єктів на екрані. Об’єктів може бути багато, а функції можуть бути складними тощо. В такому випадку, щоб уникнути затримок в анімації, функцію, яка це обчислює, можна задати в *requestAnimationFrame().* Також перевагою у використанні є те, що зміна кількох стилів не призведе до кількох переобрахунків – всі вони будуть виконані пізніше й одночасно.

Приклад використання *requestAnimationFrame()*:

```
var step = 0;
requestAnimationFrame(move);
function move() {
  step += 100;
  element.offsetWidth = step + 'px';
  element2.offsetWidth = 2*step + 'px';
  requestAnimationFrame(move);
}
```

2.1.2 requestIdleCallback()

Цей метод викликається, якщо по завершенню перемалювання фрейма залишається вільний час. Тут можна обраховувати дані, що не відносяться до пікселів на екрані (передача аналітичних даних, читання файлу тощо). Має дві сигнатури: *requestIdleCallback(callback)* та *requestIdleCallback(callback, options),*

де *callback* – функція, яку ми передаємо на виконання, та *options* – додаткові параметри конфігурації (наразі є лише *timeout*: якщо час, представлений цим параметром, минув, то функція *callback* буде поставлена в чергу в циклі подій та виконається після наступного фрейма) [7].

Основна ідея використання – максимально ефективно використати час та ресурси браузера; таким чином, ми надамо перевагу обробці подій, рендерінгу тощо, а скрипт, який ми передаємо на виконання в *requestIdleCallback(),* буде виконуватися за можливості та мати менший пріоритет, ніж інші процеси.

Як приклад застосування, ми хочемо, щоб під час виконання анімації браузер також обробляв певний файл та виконував його аналіз. Скажімо, треба зробити багато складних операцій з доволі великим об’ємом даних; якщо запустити таку обробку разом з кодом анімації, то ми помітимо, що знизилася частота оновлення кадрів та з’явилися пропущені фрейми, що погано для UX.

Приклад коду:

```
var array = getContent(); //array of items

var index = 0; //counter
window.requestIdleCallback(compute);
function compute(deadline) {
 //process data until there is time left
 while (deadline.timeRemaining() > 0) {
  array[index++].process();
 }
}
```

Тут *array* – певні дані, які ми будемо обробляти та які, в даному прикладі, реалізовують функцію *process()*; *deadline* – час, що залишилися до кінця фрейму, за який виконається максимально можлива кількість ітерацій.

Так, в теорії, без застосування псевдо-багатопоточності завдання з останнього прикладу завантажували б роботу основного потоку та блокувало користувацький ввід: прокрутку, натискання на кнопки, введення даних тощо.

Чому ж такий підхід називається «псевдо-багатопоточністю»? Якщо схематично зобразити порядок викликів, то ми помітимо, що браузер ніби перемикається між основною задачею та *requestAnimationFrame* чи *requestIdleCallback,* та виконує їх у «вільний» час, а все, що не встигне, відтерміновує до наступного фрейму. Але, як і зображено на рисунку, браузер не створює новий потік, і весь скрипт (як головний, так і другорядний), виконується в одному головному потоці.

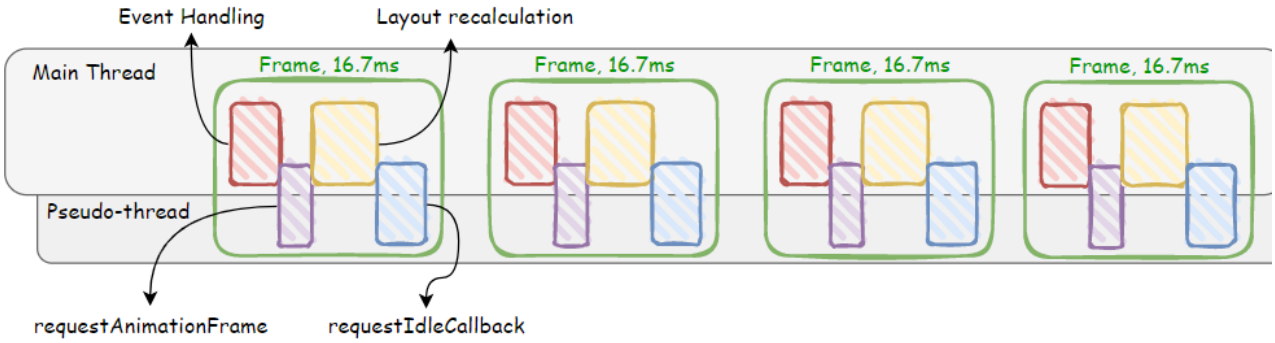


Рисунок 2.1 - Схематичне зображення псевдо-багатопоточності

2.2 Вебворкери

Вебворкери – це засіб, що дозволяє запускати скрипт в фоновому потоці. Цей потік може виконувати завдання, які не втручаються в інтерфейс користувача. Після створення воркери можуть вільно комунікувати з JavaScript-кодом, надсилаючи повідомлення. Також вони можуть здійснювати ввід та вивід, використовуючи *XMLHttpRequest* (API, що надає функціонал для обміну даними між клієнтом і сервером) [9].

Воркер запускає код, який знаходиться у файлі, назву якого ми передаємо параметром у конструкторі. Весь такий скрипт виконується в іншому глобальному контексті, відмінного від поточного *window*. Тому, якщо ми будемо намагатися доступитися з контексту воркера до *window*, то отримаємо помилку.

Натомість, ми можемо звертатися до контексту воркера через *self* - властивість *WorkerGlobalScope,* що доступна тільки для читання та яка повертає відсилку до області видимості воркера; зазвичай це більш специфічний простір: *DedicatedWorkerGlobalScope, SharedWorkerGlobalScope*, чи *ServiceWorkerGlobalScope*. Всі ці три простори відрізняються рівнем доступності до конкретного воркера з різних скриптів та тривалістю життєвого циклу. Так, якщо ми створили виділений (dedicated) воркер, то він матиме простір *DedicatedWorkerGlobalScope* та буде доступний лише з того скрипта, що його породив [10]. Проте, якщо ми хочемо, щоб доступ до воркера здійснювався з багатьох скриптів, то ми створимо спільний воркер, у якого буде простір *SharedWorkerGlobalScope* [11]. До *ServiceWorkerGlobalScope* також можна доступитися з багатьох скриптів, проте головна його відмінність – тривалість життєвого циклу – сервіс-воркери зберігаються за межами існуваннями даної веб-сторінки [12].

Воркери можуть виконувати будь-який код всередині потоку, але з певними винятками. Зокрема, ми не можемо маніпулювати DOM-вузлами з потоку воркера, а також використовувати функції, притаманні простору *window*.

Для комунікації між потоками використовують метод *postMessage()* – для надсилання повідомлень та обробник подій *onmessage* – для відповіді на них. *onmessage* має параметр, що містить в собі подію; отримати значення можна через атрибут *data*.

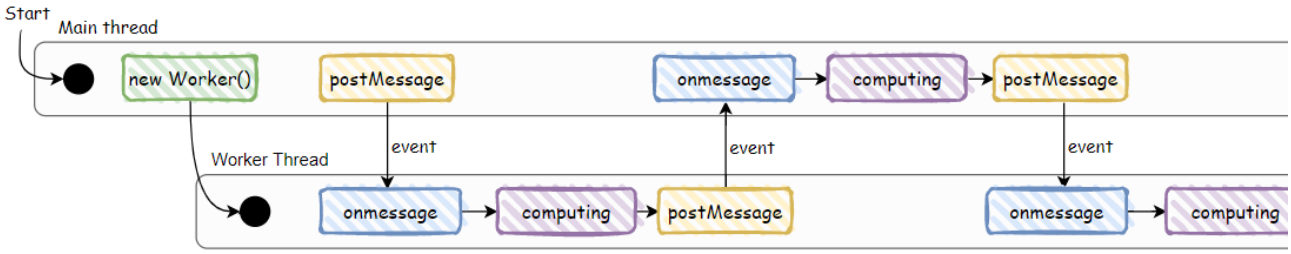


Рисунок 2.2 - Комунікація між потоками

Приклад передачі даних між основним скриптом та воркером:

```
// main.js
let person = {
  name: 'John',
  surname: 'Doe'
};
worker.postMessage(person);

// worker.js
self.onmessage = (event) =>{
  let person = event.data.person;
  console.log(person.name, person.surname);
}
```

Також воркери можуть породжувати нові воркери, якщо ті знаходяться у тому ж джерелі, що й батьківська сторінка.

Щоб завершити роботу воркера можна використати методи *terminate()* або *close()*. Перший одразу припиняє подальше виконання коду незалежно від того, виконується певне завдання чи ні. Другий завершує роботу воркера в його ж області видимості (scope).

3 Розробка практичної частини

3.1 Постановка задачі

Головною метою роботи є дослідження ефективності використання псевдо-багатопоточності і вебворкерів у випадку обробки великих об’ємів даних. Джерелом таких даних будуть CSV-файли різного розміру.

Критеріями якості будуть середній час виконання аналізу та частота оновлення кадрів (зокрема відсутність пропущених фреймів).

3.2 Реалізація застосунку

Так як обрана тема прямо стосується браузера та клієнтської сторони, то для створення застосунку ми використовуємо JavaScript для обрахунків та HTML для розмітки сторінки. Стилі сторінки задаються за допомогою CSS, а також за допомогою бібліотеки “*Bootstrap*”.

Для того, аби на результати виконання не впливали процеси, в які ми не можемо безпосередньо втрутитися, ми відмовимося від використання фреймворків.

3.2.1 Вміст застосунку

На сторінці матимемо форму для завантаження файлу та таблицю, в яку записуватимуться результати (знімок екрану надано в **додатку А**).

Користувач може обрати будь-який файл формату CSV та провести його аналіз. Для зручності праворуч відображається розмір поточного файлу. Після

успішного підтвердження почнеться читання та обробка. На екрані з’явиться відповідний статус процесу – “*Analyzing…*” (знімок екрану надано в **додатку Б**).

Також для демонстрації того, що довготривалий аналіз файлу не погіршує продуктивність браузера та UX, додано індикатор виконання та показується співвідношення вже оброблених рядків до загальної їх кількості.

Після виконання статус зміниться на “*Done!*” та додасться запис в таблицю з часом, який було затрачено на аналіз файлу як за допомогою псевдо-багатопоточності, так і за допомогою вебворкера (знімок екрану надано в **додатку В**).

3.2.2 Неоптимізований аналізатор

Цю реалізацію ми не будемо розглядати надалі, адже вона апріорі гірша за наступні. Тут ми не використовуватимемо жодної оптимізації, а просто пройдемося циклом по всім рядкам. В кінці кожної ітерації ми б мали змінювати наш індикатор, але на практиці ми помітимо, що цього не відбудеться, адже через те, що код виконуватиметься неперервно, момент перемальовки кадру не наступить – отримаємо пропущені фрейми.

```
for (let i = 1; i < lines.length; i++) {
  let obj = {};
  let props = lines[i].split(",");
  for (let j = 0; j < keys.length; j++) {
    obj[keys[j]] = props[j];
  }

  pr.innerHTML = `${i}/${lines.length - 1}`;
  progressBar.style = `width:${Math.round((i * 100) / (lines.length - 1))}%`;
}
```

3.2.3 Аналізатор через псевдо-багатопоточність

Головна задача алгоритму: в кінці фрейму, після рендерінгу, якщо залишається ще час, то потрібно витратити його на обробку рядків файлу. Для цього, передамо метод-обробник (назвемо його *_analyze()*) як параметр в *requestIdleCallback()*.

Тепер необхідно вирішити, чи потрібно задавати *timeout* другим параметром. На що він впливає? Так бувають випадки, коли браузер дуже завантажений, і ми хвилюємося, що метод, переданий в *requestIdleCallback()* може ніколи не викликатися. В таких випадках нам необхідно вказувати цей параметр, бо так ми явно повідомимо браузеру, що за задану кількість мілісекунд потрібно викликати цей метод. Проте це може знизити продуктивність та спричинити затримки при прокрутці, натисканні на кнопки та іншій взаємодії користувача зі сторінкою.

Так як в нашому застосунку не буде багато інтерактивного функціоналу та нашою ціллю є досягнення максимальної продуктивності, ми дозволимо браузеру вирішувати, коли саме викликати функцію.

Тож задамо сам метод-обробник та його виклик в *requestIdleCallback()*:

```
//main method
export function analyzePseudoMT(text, onComplete) {

  /* not essential code */
  const _analyze = (deadline) => {
    while (
      (deadline.timeRemaining() > 3 || deadline.didTimeout) &&
      index < lines.length
    ) {
      if (index === lines.length - 1) {
```

```
        break;
      }
       /* analyzing */
      index++;
    }
  if (!handler) {
    handler = window.requestIdleCallback(_analyze);
  }
}
```

Якщо ж за один виклик методу браузер не встиг виконати всю заплановану роботу, то потрібно знову запланувати виклик *requestIdleCallback()*.

```
// processor method
function _analyze(deadline) {
  while (
    (deadline.timeRemaining() > 0 || deadline.didTimeout) &&
    index < lines.length
  ) {
    ...
  }
  if (handledLines != lines.length-1) {
    handler = requestIdleCallback(_analyze);
  }else{
    ...
  }
}
```

Наступний крок – задати зміну індикатора виконання та кількості вже оброблених рядків. Для цього ми використаємо *requestAnimationFrame()*. Після

завершення циклу, коли час на виконання методу *requestIdleCallback()* вже минув, ми заплановуємо перемалювання.

```
// create an instance of the class that is responsible for displaying updates
let progressBar = new ProgressBar("pmt", lines.length);
const _analize = (deadline) => {
  while (
    (deadline.timeRemaining() > 0 || deadline.didTimeout) &&
    index < lines.length
  ) {
    ...
  }
  // redrawing
  progressBar.update(index);
  if (handledLines != lines.length-1) {
    handler = requestIdleCallback(_analyze);
  }else{
    ...
  }
}
```

Відповідальним за перевалювання є клас *ProgressBar* (код класу наведено в **додатку Г**). При створенні ми передаємо йому ім’я та загальну кількість рядків, яка є в файлі. Клас містить в собі головний метод *update(index)*, де *index* – поточний номер рядка, в якому ми задаємо, які зміни мають відбутися. Для досягнення максимальної ефективності, ми намагаємося не застосовувати всі зміни, що передаються в *requestAnimationFrame()*, а лише останні. Для цього ми запам’ятовуємо індекс останнього звернення до *requestAnimationFrame()* та перевіряємо, чи є в змінній *handlerIdx* якесь значення. Якщо є, то використовуємо *cancelAnimationFrame()* для того, аби відмінити промалювання кадру та

переписуємо значення змінної. Таким чином, коли на етапі створення фрейму викличеться *requestAnimationFrame()*, то в черзі буде лише одне завдання.

3.2.4 Вебворкер

Основою цього підходу є створення окремого потоку для обробки файлу за допомогою вебворкера. Тож першим кроком буде створення самого воркера.

```
// create new worker

let worker = new Worker("./DataAnalizers/Worker.js");
```

Далі необхідно передати дані на оброку. Цей процес займає доволі багато часу, тож це може призвести до видимих затримок. Для передачі даних воркеру використаємо *postMessage()*:

```
  // post a message to the worker

  worker.postMessage({ lines, keys });
```

Воркер має обробити повідомлення, надіслане з основного скрипту через *onmessage*:

```
/* worker.js
 Process the message */
self.onmessage = (event) => {
  const data = event.data;
  analyze(data.lines, data.keys);
  self.close();
};
```

Тут ми отримуємо дані з повідомлення та передаємо їх на обробку. В кінці аналізу ми зупиняємо обробку воркера. Основний функціонал виконує функція *analyze(lines, keys)*:

```
function analyze(lines, keys) {
```

```
  for (let i = 1; i < lines.length; i++) {
    let obj = {};
    let props = lines[i].split(",");
    for (let j = 0; j < keys.length; j++) {
      obj[keys[j]] = props[j];
    }
    if (i % 1500 === 0 || i === lines.length - 1) {
      postMessage({ index: i, isDone: i == lines.length - 1 });
    }
  }
};
```

Ми надсилаємо повідомлення лише раз на 1500 рядків або ж в кінці обробки, для того, щоб зменшити навантаження на браузер та уникнути надсилання зайвих повідомлень. Головному скрипту ми надсилаємо поточний індекс та значення *isDone*, яка приймає значення *true*, якщо всі рядки було оброблено, та *false* в іншому випадку.

Далі основний скрипт має обробити дані, надіслані воркером.

```
worker.onmessage = (event) => {
  let data = event.data;
  progressBar.update(data.index);
  if (data.isDone) onComplete(new Date().getTime() - start);
};
```

4 Дослідження результатів

Користувач може одразу побачити порівняння часу, затраченого на обробку файлу двома способами. Проте, щоб встановити межі, коли і який із наведених варіантів реалізації буде доцільнішим, цього може бути недостатньо.

Тож оцінимо час виконання алгоритму як для псевдо-багатопоточності, так і для вебворкерів на файлах, що відрізнятимуться за розміром.

Для зручності, поділимо файли за розміром на категорії: маленькі (до 1 Мб), середні (до 100 Мб) та великі (понад 100 Мб).

Розпочнемо з маленьких файлів, виміряємо час, за який вони обробляться та оцінимо частоту оновлення кадрів.

Розмір файлу, кб	**Кількість рядків**	**Час, мс**	
		Псевдо-багатопоточність	Вебворкер
0.71	3	10	22
14.09	189	10	20
16.08	210	10	26
47.81	614	15	17
136.24	242	7	20
515.07	12001	24	33
868.13	2334	35	38
967.60	2588	42	43
1012.35	103664	98	89

Таблиця 4.1 - Результати для маленьких файлів

Зобразимо результати у вигляді графіка:

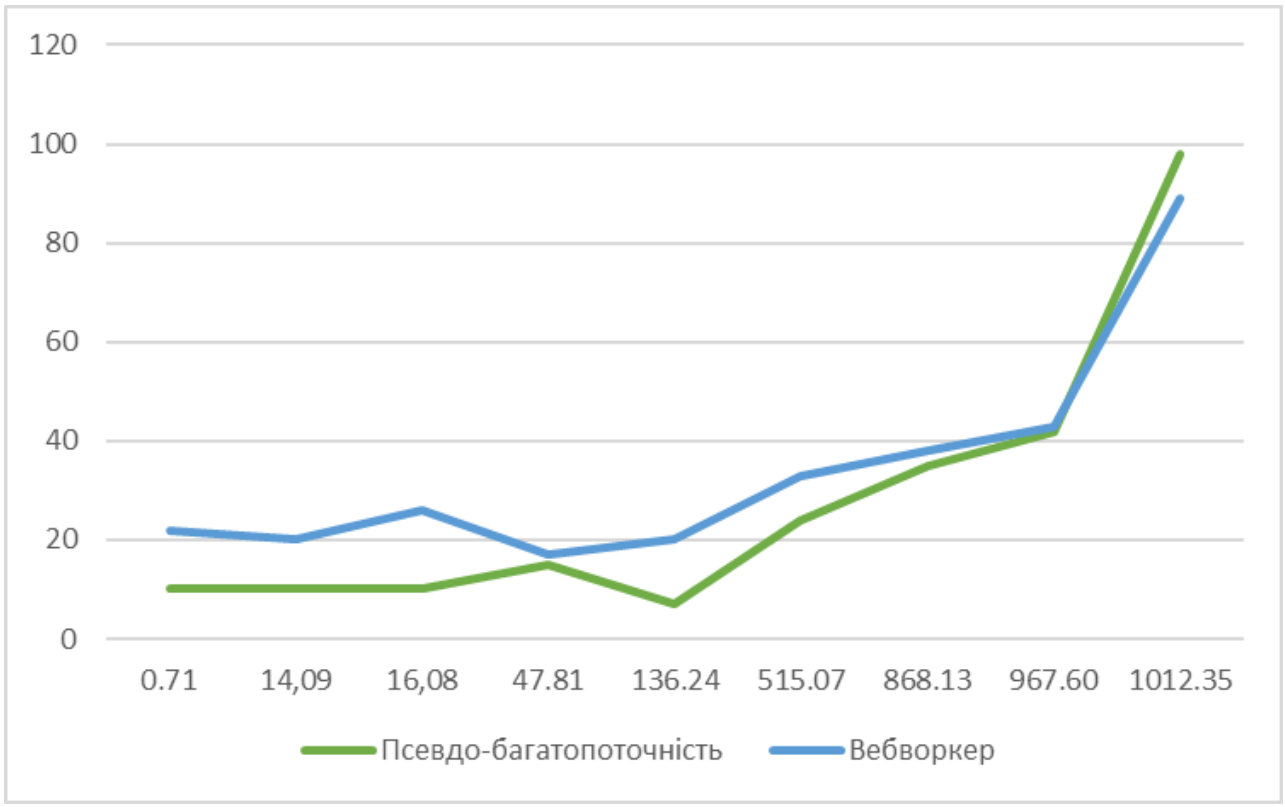


Рисунок 4.1 - Графік порівняння часу виконання на маленьких файлах

На цих даних ми бачимо, що псевдо-багатопоточність затрачає менше часу на обробку файлів допоки розмір файлу менший за 967 кб. Далі ми можемо помітити, що швидше файл аналізує вебворкер.

Для того, аби оцінити частоту оновлення кадрів, скористуємося інструментом браузера *Google Chrome – DevTools, Performance*, який наглядно демонструє продуктивність. Для найбільшого файлу в цій групі (1012.35 кб):

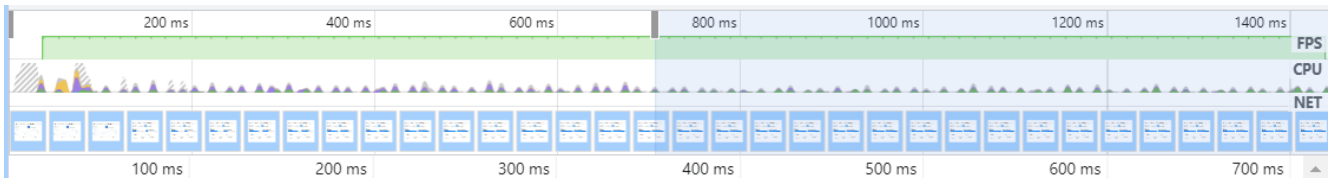


Рисунок 4.2 - Рівень частоти оновлення кадрів для маленьких файлів

Бачимо, що для файлів невеликої розмірності, рівень частоти оновлення кадрів завжди залишається високим.

Проаналізуємо алгоритми на файлах середнього розміру:

Розмір файлу, кб	**Кількість рядків**	**Час, мс**	
		Псевдо-багатопоточність	Вебворкер
1692.44	4637	66	50
3227.74	49069	100	84
7145.48	731696	782	522
19747.18	260494	526	410
41359.65	545124	1153	799
57387.11	754441	1577	1091
79675.91	1048576	2082	1632
80235.91	522278	1316	1109

Таблиця 4.2 - Результати для середніх файлів

Також візуалізуємо результати:

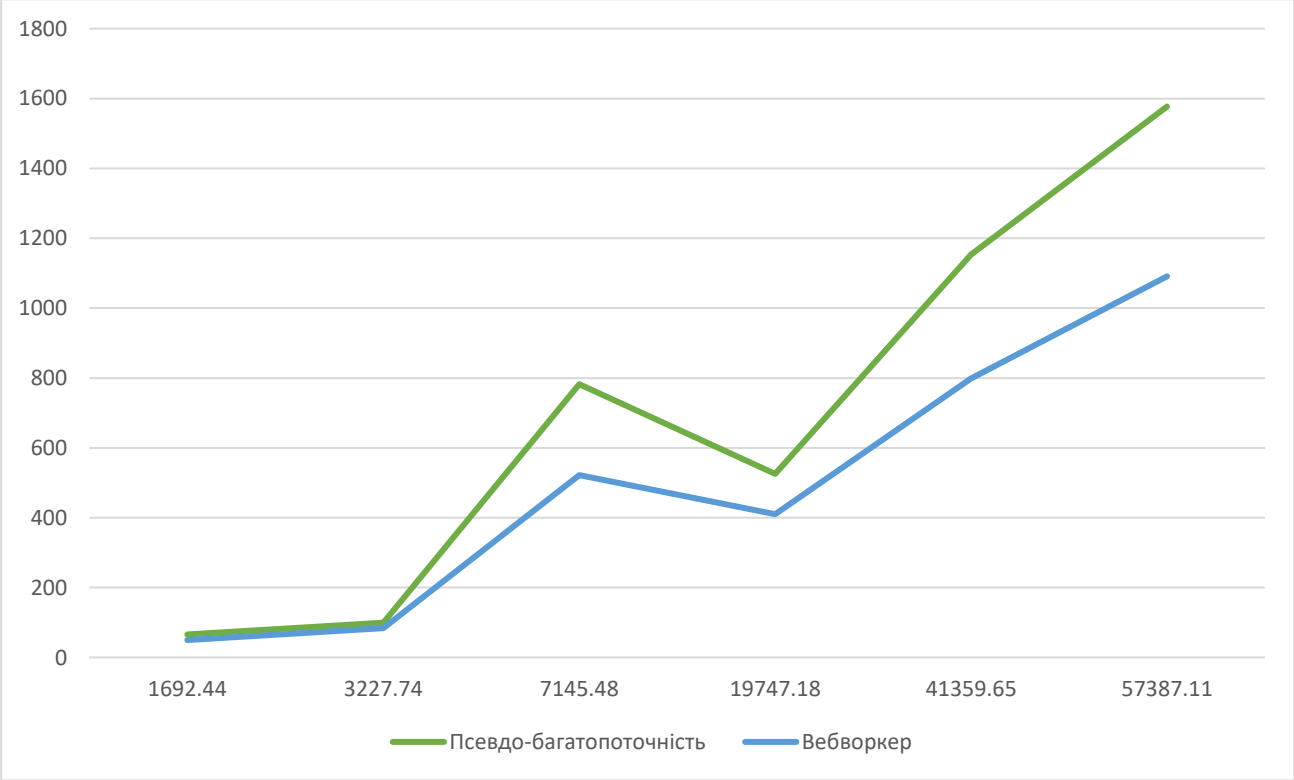


Рисунок 4.3 - Графік порівняння часу виконання на середніх файлах

З графіку, що наведено вище, можна зробити висновок, що вебворкер показує кращий результат на всіх файлах.

Аналогічно до попередньої групи файлів, оцінимо частоту оновлення кадрів:

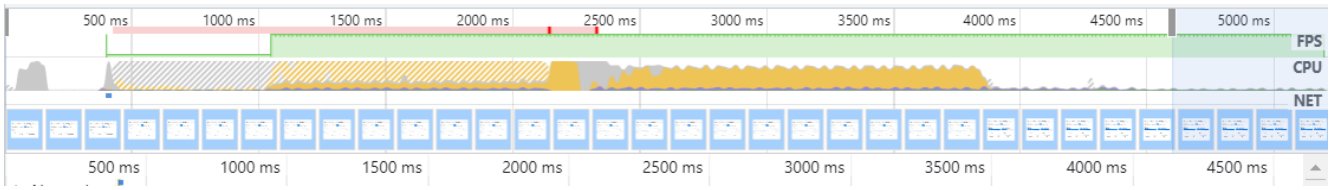


Рисунок 4.4 - Рівень частоти оновлення файлів для середніх файлів

На цих файлах вже видно, що на першій половині графіку відбувається погіршення рівня частоти оновлення кадрів (аналіз файлу вебворкером), проте для другої половини (псевдо-багатопоточність) FPS залишається високим. Чому так відбувається? Справа в тому, що надсилання повідомлення з основного потоку в потік воркера відбувається з великими затратами часу, що й призводить до затримок в анімації тощо. Переконаємося в цьому, вимірявши час від надсилання повідомлення до його отримання:

```
// WebWorkerAnalyzer.js

let timeNow = new Date().getTime();
worker.postMessage({ lines, keys, timeNow });


// Worker.js

onmessage = (event) => {
console.log("The time for which the message was received",  new
Date().getTime() - event.data.time);

 ...
};
```

Протестуємо на файлі, розмір якого становить 80 Мб. Отримаємо вивід в консоль:

Рисунок 3. Час, затрачений на надсилання повідомлення воркеру

204 мс – це приблизно 12 кадрів, що доволі багато; тому недивно, що на початку обробки ми отримуємо таку затримку.

Тепер проведемо дослідження на великих файлах:

Розмір файлу, кб	**Кількість рядків**	**Час, мс**	
		Псевдо-багатопоточність	Вебворкер
160156.27	1000001	909	1223
196389.96	1466084	2847	3248
294016.37	35888	154	726
304801.96	523618	7415	5260
318775.43	886931	12031	9132
481967.56	6362621	12303	11465

Таблиця 4.3 - Результати для великих файлів

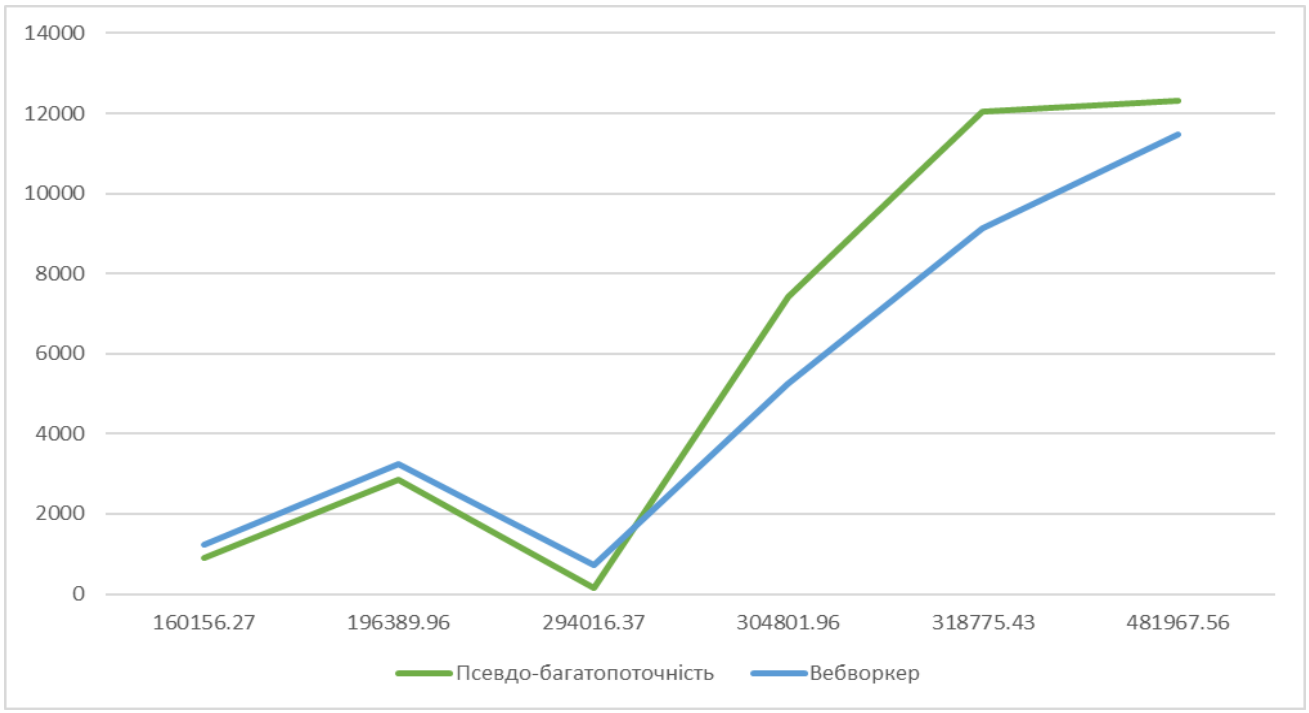


Рисунок 4.5 - Графік порівняння часу виконання на великих файлах

Псевдо-багатопоточніть показує кращий результат на файлах до приблизно 300 Мб, а на більших швидше відпрацював вебворкер.

Щодо оцінки FPS маємо аналогічну ситуацію, як і в попередній групі файлів:

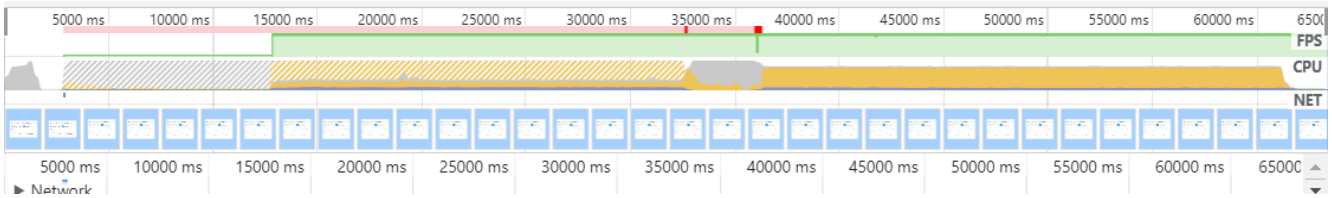


Рисунок 4.6 - Рівень частоти оновлення файлів для великих файлів

Висновки

З дослідження, описаного вище, можна зробити наступні висновки.

По-перше, ми можемо встановити приблизні межі, коли варто використовувати один чи інший спосіб аналізу. Врахувавши результати, для псевдо-багатопоточності – це файли розміром до 1000 кб або від 156 Мб до приблизно 287 Мб. Для вебворкерів – решта файлів (від 1000 кб до 156 Мб та більших, ніж 287 Мб).

По-друге, на файлах середніх та великих розмірів прослідковується зниження частоти оновлення кадрів, якщо обробляти його за допомогою вебворкерів.

Підсумовуючі, можна зробити висновок, що загалом вебворкери мають кращу часову оцінку, проте можуть показувати гіршу продуктивність, так як надсилання повідомлень з основного потоку до вебворкера займає доволі великий проміжок часу. Натомість псевдо-багатопоточність забезпечує високий показний частоти оновлення кадрів, хоч і може показувати гірший часовий результат, особливо, якщо під час обробки файлу браузер також виконує й інші задачі.

Список використаної літератури

1. Populating the page: how browsers work – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:

https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work

2. Response Times: The 3 Important Limits – 1993. – [Електронний ресурс] – Режим доступу до ресурсу:
Response Time Limits: Article by Jakob Nielsen (nngroup.com)

3. Rendering and the WebXR frame animation callback – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/WebXR_Device_API/Rendering

4. The Anatomy of a Frame – 2016. – [Електронний ресурс] – Режим доступу до ресурсу:
https://aerotwist.com/blog/the-anatomy-of-a-frame/

5. Avoid large, complex layouts and layout thrashing – 2018. – [Електронний ресурс] – Режим доступу до ресурсу:
https://web.dev/avoid-large-complex-layouts-and-layout-thrashing/#avoid-layout-thrashing

6. window.requestAnimationFrame() – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame

7. window.requestIdleCallback() – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback

8. javascript: multithreading and pseudo-threading – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://blog.actorsfit.com/a?ID=00950-baa2340e-6e63-499e-864f-02f2000779d6

9. Using Web Workers – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

10. DedicatedWorkerGlobalScope – 2022. – [Електронний ресурс] – Режим доступу до ресурсу:
https://developer.mozilla.org/en-US/docs/Web/API/DedicatedWorkerGlobalScope

11. SharedWorkerGlobalScope– 2022. – [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/SharedWorkerGlobalScope

12. ServiceWorker– 2022. – [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker

Додаток А

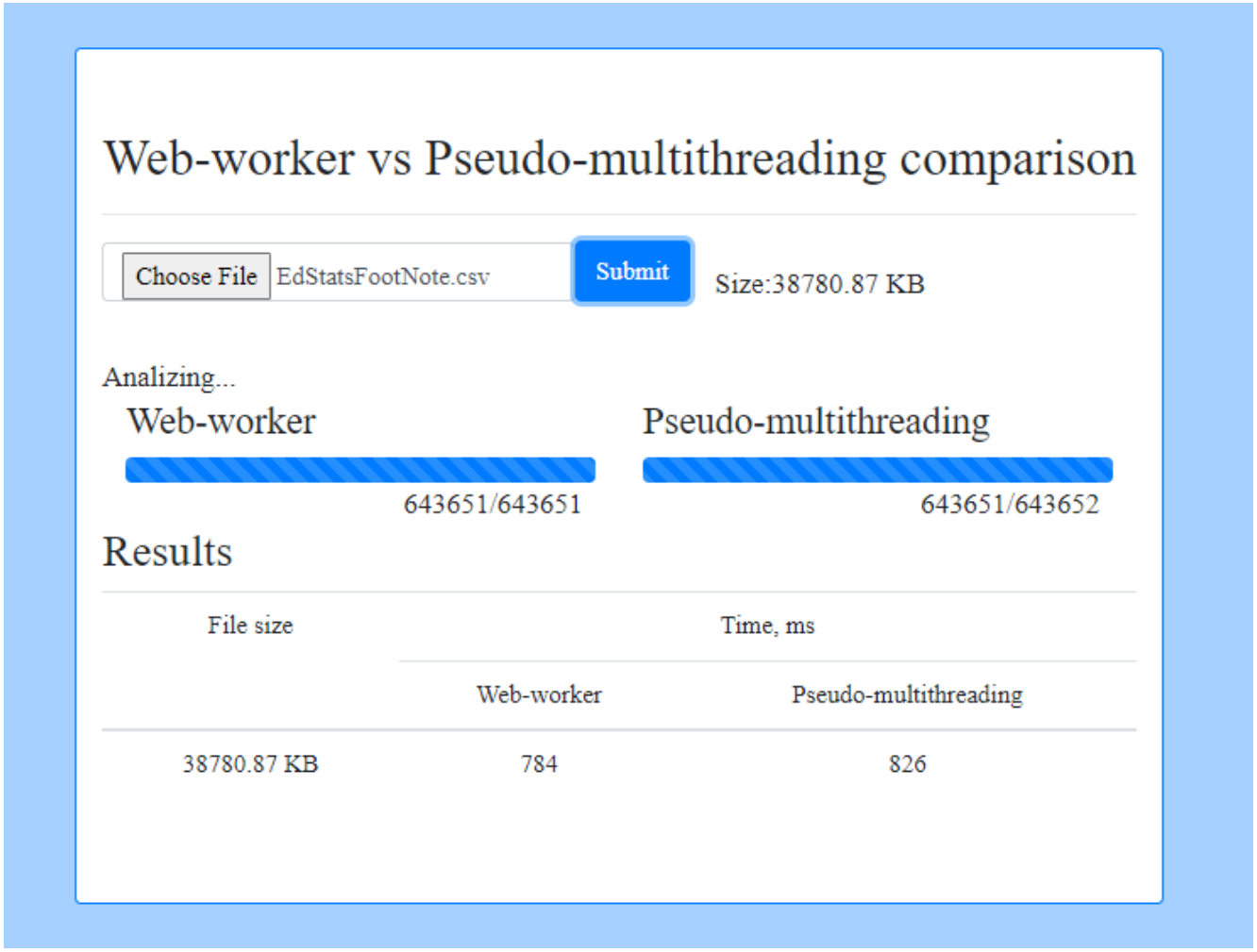
Форма для завантаження файла

Додаток Б

Поява статусу виконання

Додаток В

Результат виконання



Додаток Г

Реалізація класу *ProgressBar*

```
export class ProgressBar {

  _name = "";
  _totalLines = 0;
  _drawingCallback;
  progressBar;
  handled;

  constructor(name, totalLines) {
    this._name = name;
    this._totalLines = totalLines;

    this.progressBar = document.getElementById("progress_bar_" + name);
    this.handled = document.getElementById("handled_" + name);
    console.log("progress bar with name ", name, "created");
    document.getElementById("total_" + name).innerHTML = totalLines;
  }

  //schedule redrawing
  update(index) {
    this._requestAnimationFrame(() => {
      this.progressBar.style.width = Math.round((index * 100) / (this._totalLines -
1)) + "%";
      this.handled.innerHTML = index;
    });
```

```
  }

  _requestAnimationFrame(callback) {
    if (this._drawingCallback) {
      this._cancelAnimationFrame(this._drawingCallback);
    }

    this._drawingCallback = window.requestAnimationFrame(callback);
  }

  _cancelAnimationFrame(handlerIdx) {
    if (handlerIdx <= 0) return;
    window.cancelAnimationFrame(handlerIdx);
  }
}
```