Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**Розробка гіпертекстового браузеру на мові програмування Rust**»

Виконав: студент 4-го року навчання,

Спеціальності
121 «Інженерія Програмного Забезпечення»

Мединський Ярема Тарасович

Керівник Бабич Т. А.
спеціаліст комп'ютерних наук, асистент

Рецензент _____

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«11» травня 2023 р.

Київ – 2023

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Освітній ступінь бакалавр

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма бакалавр

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“10” жовтня 2022 року

ЗАВДАННЯ

ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТУ

Мединському Яремі

1. Тема роботи «**Розробка гіпертекстового браузеру на мові програмування Rust**», керівник роботи Бабич Трохим Анатолійович, спеціаліст комп'ютерних наук, асистент

2. Строк подання студентом роботи 24 травня 2023

3. План роботи

- Анотація
- Вступ
- Розділ 1. Дослідження та аналіз предметної області
    - Визначення та історія розвитку веб-браузерів
    - JavaScript-рушій V8

    - Структура веб-браузера
    - Виклики під час розробки веб-браузера
    - Обробка веб-сторінки браузером
    - JavaScript і його роль в сучасному веб-браузері
    - Аналіз сучасних веб-браузерів та їх рушіїв
    - Експериментальний браузер на Rust - Servo
- Розділ 2. Практична реалізація веб-браузера
    - Розробка архітектури браузера
    - Парсинг HTML
    - Парсинг CSS
    - Реалізація структури об’єктів CSS
    - Реалізація структури DOM-дерева
    - Реалізація інтеграції V8 з браузером
    - Реалізація структури DOM-дерева
    - Перетворення DOM-дерева на LayoutBoxes
    - Виведення на екран(рендеринг)
    - Демонстрація роботи
- Висновки
- Список використаних джерел

ГРАФІК ПІДГОТОВКИ КУРСОВОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	жовтень			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	жовтень — листопад			
3.	Складання плану каліф. роботи та узгодження з науковим керівником	листопад			
4.	Написання розділів роботи	листопад — березень			
5.	Проміжний контроль виконання роботи	лютий			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	січень — березень			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)				
	Розділ 2 (аналітично-дослідницька частина)				
	Розділ 3 (проектно-рекомендаційна частина)				
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	квітень — початок травня			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	середина травня			
9.	Подання на зовнішню рецензію	середина травня			
10.	Підготовка до захисту кваліфікаційної роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративно гоматеріалу	до 9 травня			
11.	Попередній захист кваліфікаційної роботи на засіданні кафедри	до 11 травня			
12.	Подання кваліфікаційної роботи на кафедру з усіма супроводжувальними документами	до 24 травня			
13..	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 10 жовтня 2022 р.

Науковий керівник Бабич Трохим Анатолійович

Виконавець кваліфікаційної роботи Мединський Ярема Тарасович

ЗМІСТ

Анотація **8**

Вступ **8**

Розділ 1. Дослідження та аналіз предметної області **9**

1.1. Визначення та історія розвитку веб-браузерів 9

1.2. JavaScript-рушій V8 11

1.3. Структура веб-браузера 13

1.4 Виклики під час розробки веб-браузера 16

1.5 Обробка веб-сторінки браузером 18

1.6 JavaScript і його роль в сучасному веб-браузері 19

1.7 Аналіз сучасних веб-браузерів та їх рушіїв 21

1.8 Експериментальний браузер на Rust – Servo 23

Розділ 2. Практична реалізація веб-браузера **25**

2.1. Розробка архітектури браузера 25

2.2 Парсинг HTML 28

2.3 Парсинг CSS 30

2.4 Реалізація структури об’єктів CSS 31

2.5 Реалізація структури DOM-дерева 33

2.5 Реалізація інтеграції V8 з браузером 35

2.6 Перетворення DOM-дерева на LayoutBoxes. 38

2.7 Виведення на екран(рендеринг) 42

2.8 Демонстрація роботи 52

Висновки **54**

Список використаних джерел **55**

Анотація

У даній роботі розглядаються особливості створення веб-браузера на основі Rust з використанням OpenGL для рендерингу та інтеграції JavaScript через V8. Обговорюється використання Rust для підвищення продуктивності та безпеки, а також причини вибору OpenGL і GFX для рендерингу. В роботі детально описується процес створення браузера, його архітектура та функціональність.

Вступ

В умовах сучасного швидкого розвитку технологій, веб-браузери стають все більш важливими інструментами для доступу до інформації та сервісів. Однак, більшість сучасних браузерів стикаються з проблемами продуктивності та безпеки, що обумовлено використанням старіших мов програмування та технологій. Внаслідок цього виникає потреба в більш сучасних та ефективних рішеннях.

Метою цієї роботи є створення власного веб-браузера на основі мови програмування Rust, який використовує OpenGL для рендерингу та інтегрує JavaScript за допомогою V8. Робота включає детальний аналіз використаних технологій, їх переваг та недоліків, а також процес створення браузера з нуля.

У рамках виконання цієї роботи було вирішено наступні завдання:

• Проаналізувати поточні технології, що використовуються у веб-браузерах, і обґрунтувати вибір Rust, OpenGL та V8.

• Детально описати процес створення веб-браузера на Rust, включаючи рендеринг за допомогою OpenGL та GFX, інтеграцію JavaScript через V8.

• Розробити архітектуру браузера, яка б об'єднувала всі ці компоненти в єдиний продукт.

• Оцінити продуктивність та безпеку створеного браузера та порівняти його з існуючими веб-браузерами.

В ході роботи над дипломним проєктом було використано сучасні методики розробки програмного забезпечення, а також актуальні наукові дослідження в області веб-технологій.

Розділ 1. Дослідження та аналіз предметної області

1.1. Визначення та історія розвитку веб-браузерів

Веб-браузер – це програмне забезпечення, призначене для отримання, відображення і навігації вмісту, доступного через Всесвітню мережу, в основному веб-сторінок, які складаються з HTML, CSS та JavaScript коду.

Історія веб-браузерів починається з ранніх 1990-х. Перший браузер, WorldWideWeb, був створений Тімом Бернерсом-Лі у 1990 році. У 1993 році був створений Mosaic, браузер, який зробив великий вплив на

популяризацію Інтернету серед звичайних користувачів завдяки своєму графічному інтерфейсу і здатності відображати текст і зображення на тій же сторінці.

Згодом було створено багато інших веб-браузерів, зокрема таких відомих як Netscape Navigator і Internet Explorer. Конкуренція між ними, відома як "Війна браузерів", призвела до швидкого розвитку технологій і стандартів.

З 2000-х років почалась ера сучасних браузерів, таких як Firefox, Chrome, Safari, кожен з яких має свій власний рушій для відображення веб-сторінок: Gecko, Blink і WebKit відповідно.

Сьогодні веб-браузери використовуються не тільки для перегляду веб-сторінок, але і для виконання складних веб-застосунків, відтворення мультимедіа, веб-ігор, роботи з онлайн-сервісами та інтерактивних додатків. Значна кількість сучасних технологій та стандартів, таких як HTML5, CSS3, ECMAScript 6 і багато інших, з'явилися з метою розширення можливостей браузерів і підтримку вимог розробників і користувачів.

Останнім часом все більше розробників вдаються до створення власних веб-браузерів на основі відкритого коду існуючих браузерних рушіїв, таких як Blink (на основі якого працює браузер Google Chrome) та Gecko (на основі якого працює браузер Firefox). Це дозволяє їм створювати

продукти, які відповідають специфічним вимогам їхніх користувачів або спрямовані на конкретні сегменти ринку.

Зважаючи на складність і динамічну природу веб-технологій, створення власного веб-браузера вимагає глибоких знань і досвіду в області програмування та веб-технологій, а також здатності враховувати вимоги безпеки, приватності користувачів і відповідність стандартам.

У рамках даного дослідження було вирішено створити власний веб-браузер, який використовує рушій V8 від Google, що забезпечує високу швидкість виконання JavaScript-коду, та інтегрує новий підхід до обробки HTML та CSS, що повинен забезпечити більш ефективну та стабільну роботу браузера.

1.2. JavaScript-рушій V8

Величезний вплив на сучасний Інтернет має JavaScript-рушій V8, розроблений компанією Google. Цей рушій слід розглядати як фундаментальний елемент веб-браузера Chrome, що грає ключову роль в інтерпретації та виконанні JavaScript-коду, надаючи користувачам високу продуктивність та різноманітність функціональності.

Однією з основних властивостей V8 є його здатність до JIT (Just-In-Time) компіляції. Він перетворює JavaScript безпосередньо в машинний код перед його виконанням, відкидаючи традиційний етап байт-коду, що забезпечує прискорення процесу виконання коду. Ця оптимізація робить

V8 ефективним для великих та складних веб-додатків, оскільки це дозволяє їм працювати плавно та ефективно.

V8 також володіє постійно оновлюваною підтримкою новітніх стандартів ECMAScript, що гарантує сумісність з найновішими технологіями і практиками розробки. Це забезпечує рушію статус сучасного та гнучкого інструменту для виконання JavaScript-коду.

V8 має вражаючий інструментарій для оптимізації виконання JavaScript. Це включає в себе технології як , оптимізація "гарячих" місць коду, управління пам'яттю та збирання сміття. Ці механізми працюють разом, щоб мінімізувати витрати ресурсів і підтримувати високу продуктивність.

Також, варто відмітити, що V8 не обмежується лише браузерами— він є основою для Node.js, відомої JavaScript-платформи для розробки серверних додатків.

Інтеграція V8 з Node.js відкрила нові горизонти для JavaScript, трансформувавши його з мови, яка в основному використовувалася для маніпуляцій з браузером, в мову, здатну розв'язувати різні задачі на серверній стороні. Це підтверджує універсальність V8 і показує його важливість як для веб-браузерів, так і для загальної екосистеми JavaScript.

І, нарешті, JavaScript-рушій V8 має відкритий код, що дає можливість спільноті розробників вносити вклад в його розвиток, розширювати функціонал і оптимізувати роботу. Відкритий код також забезпечує

більшу прозорість і контроль над тим, як рушій працює та використовує ресурси.

Загалом, JavaScript-рушій V8 відіграє ключову роль в розробці сучасного Інтернету, обслуговуючи як веб-браузери, так і серверні додатки. Його продуктивність, підтримка стандартів, гнучкість та активний розвиток спільноти роблять його незамінним інструментом в сучасному веб-розробці.

1.3. Структура веб-браузера

Веб-браузер - це багатокомпонентна програмна система, розроблена для відтворення інформації з Інтернету в зручному та доступному форматі. Вона є сукупністю декількох важливих модулів, які працюють разом, аби забезпечити плавне, ефективне та безпечне користувацьке досвід.

1. Користувацький інтерфейс: це частина браузера, з якою безпосередньо взаємодіє користувач. Він включає в себе не лише вікно перегляду вмісту, але й елементи керування, такі як адресний рядок, кнопки "назад" і "вперед", закладки, історія перегляду та ін. Ці інструменти допомагають користувачеві з навігацією по веб-сайтам та керувати своїми браузерними сесіями.

2. Рушій рендерингу: це компонент браузера, який перетворює HTML, CSS та JavaScript в зрозумілу та візуально привабливу структуру, що відображається на екрані користувача. Він виконує складний процес

парсингу коду, побудови DOM-дерева, визначення стилів CSS, виконання JavaScript та візуалізації кінцевої сторінки.

3. JavaScript-рушій: JavaScript є основою інтерактивності сучасних веб-сторінок. JavaScript-рушій браузера виконує код, що міститься в HTML-документах, та виконує різноманітні завдання, від простих, таких як зміна елементів DOM, до складних, наприклад, AJAX-запити та взаємодія з веб-API.

4. Менеджер ресурсів: Цей компонент браузера виконує мережеві запити до серверів за допомогою HTTP або HTTPS протоколів, отримує відповіді на ці запити, обробляє їх, а потім передає відповідні дані до рушія рендерингу для подальшої обробки та відображення. Він також керує кешуванням даних, що може забезпечити прискорення завантаження сторінок при повторному відвідуванні.

5. Менеджер сеансів і історії: цей модуль браузера відповідає за ведення історії перегляду користувача та управління активними вкладками. Він дозволяє користувачам відновлювати сеанси браузера після перезапуску або відновлення від збоїв, а також забезпечує функції такі як "назад", "вперед" і "оновити".

6. Безпека та приватність: сучасні браузери мають набір механізмів для забезпечення безпеки користувачів в Інтернеті. Це включає в себе політику того ж походження, керування сертифікатами, розблокування

небезпечного вмісту, захист від відстеження, захист від відомих шкідливих сайтів та інше.

7. Сервіси та API: сучасні браузери містять ряд додаткових служб та інтерфейсів програмування додатків, що розширюють їх функціональність. Вони включають речі, такі як геолокація, веб-сповіщення, сервісні робітники для роботи в офлайн-режимі та ін.

Важливо зазначити, що архітектура браузера може відрізнятися в залежності від конкретного браузера. Незалежно від використання веб-технологій, всі браузери мають загальну мету - відтворення веб-вмісту належним чином, надаючи користувачам багатий та безпечний досвід в Інтернеті.

8. Плагіни та розширення: Багато браузерів мають можливість розширювати їх функціональність за допомогою плагінів та розширень. Ці додаткові програми можуть забезпечити додаткові сервіси, такі як блокування реклами, управління паролями, автоматизацію завдань та багато іншого.

9. Підтримка протоколів: Браузери підтримують широкий спектр мережевих протоколів, включаючи HTTP, HTTPS, FTP, WebSocket, інші, що забезпечують обмін даними між клієнтом і сервером.

Кожен з цих компонентів виконує специфічну роль в загальному процесі відображення веб-сторінок і взаємодії з ними. Усі вони спільно працюють,

щоб забезпечити гладкий та ефективний досвід використання вебу. Розробка власного веб-браузера, тому, вимагає глибокого розуміння цих компонентів та їх взаємодії.

1.4 Виклики під час розробки веб-браузера

Розробка власного веб-браузера – це надзвичайно складне завдання, що ставить перед розробниками чимало викликів.

1. Сумісність: Одним з найбільших викликів є сумісність. Інтернет є універсальною платформою, і веб-сайти розроблені так, щоб працювати на різних браузерах, операційних системах та пристроях. Новий браузер повинен правильно відображати всі веб-сторінки, незалежно від того, як вони були розроблені.

2. Безпека: Браузери є основними цілями для хакерів, оскільки вони є посередниками між користувачем та вебом. Новий браузер повинен мати високий рівень безпеки, щоб захистити користувачів від шкідливих веб-сайтів та атак.

3. Продуктивність: Браузери повинні бути швидкими і ефективними, оскільки вони обробляють великі обсяги даних та виконують складні обчислення для відтворення веб-сторінок. Новий браузер повинен мати високу продуктивність та ефективно використовувати системні ресурси.

4. Розширюваність: Сучасні браузери включають системи плагінів та розширень, що дозволяють користувачам і розробникам додавати нові функції. Новий браузер повинен підтримувати цю розширюваність, щоб бути конкурентоспроможним.

5. Відповідність стандартам: Веб-стандарти встановлені організаціями, такими як World Wide Web Consortium (W3C), для забезпечення сумісності та інтероперабельності між різними браузерами. Новий браузер повинен строго дотримуватися цих стандартів, щоб забезпечити правильну роботу веб-сторінок.

6. Оновлення та підтримка: Веб-технології швидко змінюються і розвиваються, а браузери повинні бути в змозі адаптуватися до цих змін. Тому розробники повинні забезпечити ефективний процес оновлень та постійну підтримку браузера.

7. Реалізація прогресивних веб-додатків (PWA): Сучасні браузери повинні підтримувати прогресивні веб-додатки, які є гібридом звичайних веб-сайтів та мобільних додатків, що дає їм можливість працювати офлайн та надавати повідомлення, як нативні додатки.

8. Підтримка новітніх технологій: Браузери повинні підтримувати новітні веб-технології, такі як HTML5, CSS3, JavaScript і багато інших, щоб забезпечити найкращий досвід користувача.

Врахування цих викликів та їх подолання - важливий аспект розробки веб-браузера. Дослідження цих питань становить важливу частину цієї роботи.

1.5 Обробка веб-сторінки браузером

Веб-браузери, будучи ключовими інструментами для відображення веб-сторінок, виконують складний набір процедур для того, щоб правильно представити вміст сторінки користувачу. Основні етапи цього процесу включають парсинг HTML і CSS, побудову DOM і CSSOM, та рендеринг.

Парсинг HTML: Це перший крок у процесі обробки веб-сторінки. Парсер HTML аналізує вихідний код HTML сторінки, перетворюючи його в DOM-дерево, яке є об'єктним представленням структури веб-сторінки. Парсер HTML має здатність визначати початок і кінець тегів, їх атрибути та вміст, формуючи дерево відносин між елементами.

Парсинг CSS: Наступний крок - аналіз CSS. Подібно до парсингу HTML, парсер CSS аналізує CSS-код, щоб створити CSSOM - об'єктне представлення CSS правил. Це допомагає браузеру визначити, як відобразити кожен елемент DOM.

Побудова DOM і CSSOM: Після парсингу HTML і CSS, браузер починає побудову DOM і CSSOM. DOM - це деревоподібна структура, що представляє HTML структуру документа, а CSSOM відображає стилі, що будуть застосовані до DOM. Ці два об'єктні моделі з'єднуються для

створення "render tree", який відображає весь вміст веб-сторінки з відповідними стилями.

Рендеринг: Останнім кроком є рендеринг сторінки, де браузер відображає сторінку на екрані користувача. Під час цього процесу, браузер проходить через "render tree", відображаючи кожен елемент відповідно до його властивостей і стилів. Цей процес включає в себе не лише відображення елементів на екрані, але й виконання JavaScript, який може змінювати DOM на льоту, додаючи інтерактивність на веб-сторінку.

JavaScript та інтерактивність: JavaScript відіграє важливу роль у створенні динамічних і інтерактивних веб-сайтів. Він може маніпулювати DOM і CSSOM, дозволяючи розробникам створювати анімації, реагувати на дії користувача, виконувати AJAX-запити для отримання даних з сервера без перезавантаження сторінки та виконувати інші складні завдання.

1.6 JavaScript і його роль в сучасному веб-браузері

JavaScript - це потужний інструмент в сучасному веб-браузері. Як мова програмування, що використовується на стороні клієнта, JavaScript відіграє вирішальну роль у формуванні інтерактивного, багатого та гнучкого досвіду користувача в веб-середовищі.

Одним з основних використань JavaScript є маніпуляції з Document Object Model (DOM) і CSS Object Model (CSSOM). JS надає можливість

розробникам динамічно змінювати структуру веб-сторінки, додавати, видаляти чи модифікувати елементи DOM. Крім того, JavaScript дозволяє міняти властивості CSSOM(часто є частиною DOM), що відповідають за стиль сторінки. Це забезпечує гнучкість веб-сторінок, здатних адаптуватися та відповідати на дії користувача.

JavaScript розроблений з урахуванням асинхронності, що дає можливість виконувати різні задачі паралельно або в фоновому режимі. Це включає в себе такі процеси, як отримання даних від сервера, обробка подій користувача, без того, щоб перешкоджати відображенню сторінки. Результатом є підвищення швидкості і плавності інтеракції користувача з веб-сайтом без використання потоків.

Крім того, JavaScript відкриває широкі можливості для створення анімацій та візуальних ефектів на веб-сторінках. За допомогою складних анімацій та візуальних ефектів, створених за допомогою JavaScript, розробники можуть контролювати деталі анімації, створювати інтерактивні елементи, що реагують на дії користувача, а також використовувати анімацію для пояснення складних концепцій або показу процесів.

JavaScript також відіграє важливу роль у безпеці браузера. За допомогою механізму політики походження (Same-Origin Policy) та ізоляції JavaScript, браузери можуть обмежувати доступ до інформації в контексті домену, з

якого скрипт був завантажений, забезпечуючи таким чином захист від попереднього крос-сайтового скриптингу (XSS).

Окрім цього, JavaScript використовується для реалізації різноманітних веб-API, що дозволяють взаємодіяти з різними аспектами операційної системи та браузера. Це включає в себе роботу з файлами, контроль над аудіо та відео елементами, взаємодію з мережею, роботу з даними у фоновому режимі та навіть доступ до апаратних засобів, таких як гіроскоп або камера.

Загалом, JavaScript є критичним компонентом у веб-браузері, що відповідає за велику частину веб-функціональності, з якою ми взаємодіємо щодня. Він робить веб-сайти більш інтерактивними, реактивними та корисними для кінцевого користувача, покращуючи загальний досвід користувача.

1.7 Аналіз сучасних веб-браузерів та їх рушіїв

В сучасному цифровому світі існує багато різних веб-браузерів, кожен з яких має свої особливості, переваги та недоліки. На даний момент найпопулярнішими веб-браузерами є Google Chrome, Mozilla Firefox, Safari, Microsoft Edge та Opera.

Google Chrome, що базується на відкритому рушії Blink, зараз є найпопулярнішим веб-браузером. Він відомий своєю швидкістю, безпекою, простотою використання та високою сумісністю з веб-

стандартами. Крім того, в Chrome є великий магазин розширень, що дозволяє користувачам налаштовувати браузер відповідно до власних потреб.

Mozilla Firefox, заснований на рушії Gecko, теж є досить популярним вибором серед користувачів. Firefox надає велику увагу приватності та безпеці, надаючи багато інструментів для захисту даних користувача. Також Firefox відомий своїм активним співтовариством, що розробляє численні розширення та теми.

Safari, що використовує рушій WebKit, є стандартним браузером для пристроїв Apple. Він відомий своєю енергоефективністю, що особливо важливо для мобільних пристроїв, і гарною інтеграцією з екосистемою Apple.

Microsoft Edge, який недавно перейшов на рушій Blink, також зарекомендував себе як надійний та швидкий браузер. Він активно розробляється і постійно отримує нові оновлення і функції, включаючи недавно введені вкладки вертикального розташування.

Накінець, Opera, так само заснована на рушії Blink, виділяється своїми інноваційними функціями та унікальним інтерфейсом. Вона має вбудований VPN, безкоштовну блокувальну систему реклами, а також режим економії енергії для ноутбуків.

Усі ці браузери мають свої унікальні особливості та можуть служити різним потребам користувачів. Однак вони всі використовують складні рушії для обробки веб-сторінок, що передбачає парсинг HTML і CSS, побудову DOM та CSSOM дерев, виконання JavaScript, рендеринг та відображення веб-сторінок. Розуміння цих процесів дозволяє краще розуміти, як функціонують веб-браузери, і які можливості вони можуть надати розробникам та користувачам.

Також варто відзначити, що деякі інші веб-браузери, такі як Brave та Vivaldi, хоча і менш популярні, проте пропонують унікальні функції та спеціалізовані інструменти для конкретних груп користувачів.

В цілому, розуміння особливостей сучасних веб-браузерів і їх рушіїв допомагає розробникам приймати обґрунтовані рішення при проектуванні та розробці веб-сайтів і веб-застосунків, а також при виборі відповідного браузера для використання.

1.8 Експериментальний браузер на Rust – Servo

Servo - це експериментальний веб-браузер, розроблений Mozilla Research. Це проект, що має на меті показати, як сучасні технічні рішення та різні аспекти мови програмування Rust можуть бути використані для побудови браузерів, які є безпечними, надійними, швидкими та здатними ефективно використовувати апаратні ресурси.

Servo розроблений з нуля, використовуючи мову програмування Rust, яка надає гарантії безпеки пам'яті, забезпечуючи в той же час високу продуктивність і паралельність. Це робить Servo ідеальною платформою для експериментування з новими технологіями в області веб-браузерів.

Servo використовує власний рушій CSS та рендеринг, які розроблені для паралельної обробки і використання вбудованих графічних прискорювачів сучасних комп'ютерів. Він також використовує бібліотеку SpiderMonkey від Mozilla для виконання JavaScript.

Одним із цікавих аспектів Servo є його архітектура. Замість традиційного монолітного дизайну, який є характерним для багатьох веб-браузерів, Servo є модульним, що дозволяє легко додавати, видаляти або змінювати окремі компоненти.

Наразі Servo є експериментальним і не призначений для загального використання. Однак, його технології активно вивчаються та інкорпоруються в інші проєкти. Наприклад, рушій стилю Servo, відомий як Stylo, було інтегровано в Firefox як частина проекту Quantum.

У цілому, Servo демонструє, як нові технології та мов програмування можуть бути використані для побудови наступного покоління веб-браузерів.

Проект Servo демонструє ряд ключових інновацій у веб-технологіях. Він розроблений так, щоб максимізувати паралельну обробку,

використовуючи конкурентні системи, які дозволяють багатопоточну обробку та асинхронні виклики, які значно зменшують блокування.

Servo також відкриває двері до експериментування з різними моделями виконання коду. Це може включати використання WebAssembly для побудови компонентів браузера, що прискорює їх виконання та забезпечує більшу безпеку від потенційних атак.

При розгляді рушіїв рендерингу, Servo використовує WebRender, рушій, що працює з GPU, який використовує техніки з комп'ютерних ігор для покращення продуктивності візуалізації.

Незважаючи на те, що Servo є експериментальним проектом, його вплив на індустрію вже відчутний. Він є джерелом інновацій та ідей, які вже використовуються і в діючих веб-браузерах, таких як Firefox, так і в наукових дослідженнях. Він також стимулює розвиток Rust як мови, що забезпечує безпеку пам'яті, безпечну багатопоточність та високу продуктивність - аспекти, які є життєво важливими для сучасних веб-браузерів.

Розділ 2. Практична реалізація веб-браузера

2.1. Розробка архітектури браузера

Архітектура веб-браузера, розробленого в рамках цього проєкту, є спрощеною і фокусується на основних складових веб-браузера, які власне і є 5 головними етапами:

- парсинг HTML, CSS
- виконання JavaScript коду
- побудова DOM-дерева
- перетворення DOM-дерева на коробки
- рендеринг вмісту сторінки.

Головною структурою(класом) є `Browser`, що містить основні поля:

- `html`: зберігає HTML-код, який має бути відображений;
- `stylesheet`: представляє таблицю стилів CSS, яка використовується для рендерингу HTML-коду;
- `js`: зберігає JavaScript-код, який має бути виконаний;
- `nodes`: вектор вузлів, що представляє структуру DOM;
- `title`: назва веб-сторінки.

Метод `run` виконує основний цикл виконання браузера: аналізує HTML-код, знаходить відповідні CSS-стилі та JavaScript-код, розбирає їх, виконує JavaScript, побудовує модель вмісту сторінки (DOM), опрацьовує CSS-стилі та накінець виконує рендеринг сторінки.

Використання Rust дозволяє створювати високопродуктивний та безпечний код завдяки системі власності та контролю життєвого циклу.

OpenGL і GFX використовуються для рендерингу вмісту сторінки, дозволяючи створювати високоякісне візуальне представлення вмісту. GFX дозволяє абстрагуватися від конкретних деталей графічного апі, забезпечуючи більш гнучкий та адаптивний до різних платформ код.

Інтерпретатор JavaScript V8, розроблений Google, використовується для виконання JavaScript-коду на веб-сторінці. Він надає високу швидкість виконання, підтримує сучасні стандарти JavaScript та надає потужні можливості для оптимізації виконання коду.

Браузер складається з наступних компонентів:

Парсинг HTML: HTML-парсер аналізує HTML-код, що був отриманий, і перетворює його на структуру DOM. DOM - це внутрішнє представлення веб-сторінки у вигляді дерева вузлів.

Парсинг CSS: CSS-парсер аналізує CSS-стилі, що були отримані, і перетворює їх на структуру, що може бути використана для застосування стилів до відповідних вузлів DOM.

JavaScript-інтерпретатор: JavaScript-інтерпретатор виконує JavaScript-код, що був отриманий. Це може прочитати структуру DOM api, або якось змінити її.

Побудова DOM-дерева: На основі HTML будується DOM-дерево, яке після парсингу CSS заповнюється стилями.

Побудова елементів для виведення на екран: Після побудови DOM-дерева відбувається побудова «коробок», які потім будуть виведені на екран. Ці об’єкти отримують такі властивості як колір, розмір та координати.

Рендеринг: Після того, як коробки було побудовано відбувається виведення на екран.

2.2 Парсинг HTML

Парсинг HTML реалізований як окрема структура, якій при створенні передається HTML код у вигляді слайсів(std::str), після цього ці слайси перетворюються на об’єкт класу std::iter::Peekable, який містить всі символи html файлу. Тобто це виходить як масив символів, до яких можна зручно доступитись. Також наявний список стрічок, який зберігається у полі node_q. Він використовується для додавання ім’я тегу, яке потім використовується у створенні елеметів DOM.

Запускається парсинг через метод parse_nodes. Основну частуну метода займає цикл while, який перевіряє, чи залишились ще символи. В тілі цикла спочатку наявний метод, який пропускає символи, якщо є пробілом(whitespace). Коли пробіли закінчуються, і починається трикутна дужка "<", тоді може бути три варіанти:

1) Початок тегу
2) Початок коментаря(якщо після йде знак оклику "!")
3) Закінчення тегу(якщо після йде символ бекслешу "/")

Якщо це початок тегу, тоді викликається метод parse_node. В ньому відбувається перевірка на валідне ім’я тегу. Також відбувається парсинг атрибутів та виклик метода parse_nodes для парсингу дитячих тегів. Конкретного елементу.

Якщо це просто текст, тоді запускається метод parse_text_node, який зчитує текст поки не зустріне відкриваючу трикутну дужку. Після цього виконання методу завершується і вертається об’єкт класу Node з типом Text.

Якщо це коментар, тоді викликається метод parse_comment_node. Коментар HTML має виглядати так: <!-- sometext-->. Оскільки парсер вже зчитав символи "<" та "!", тому метод очікує побачити два тире, після чого зчитує текст та якщо наявні два тире та трикутна закриваюча дужка, завершує своє виконання вертаючи об’єкт класу Node з типом Comment.

Парсинг атрибутів відбувається у методі parse_attributes. Метод працює поки не зустріне закриваючу трикутну дужку. Після зчитування ключа атрибута очікує зчитати символ дорівнює «=» і тоді викликає метод зчитування значень атрибутів: parse_attr_value. В цьому методі зчитується значення атрибуту разом з перевіркою на валідність, і вертається у вигляді

об’єкта String. Між кожним етапом викликається метод пропуску пробілів. Після цього метод parse_attributes вертає об’єкт типу AttrMap. AttrMap – це аліас на HashMap<String, String>.

2.3 Парсинг CSS

Парсинг CSS так само як й HTML був реалізований окремою структурою. Було так само використано клас Peekable для ітерації по CSS коду. CSS код передається при створенні парсера. Для запуску використовується метод parse_stylesheet, який повертає об’єкт Stylesheet(докладніше в наступному розділі). Якщо файл не пустий, то викликається метод parse_rule, який парсить окреме правило, тому відповідно цей метод повертає об’єкт класу Rule. Метод спочатку пропускає всі символи, які містять пробіл, після того, як метод доходить до селектора запускається метод parse_selector. Спочатку він перевіряє селектор на валідність. Якщо все добре, тоді дивлячись на символ перед селектором визначається його тип(селектор за типом, селектор за класом, селектор за id). Метод parse_selector вертає об’єкт класу Selector. Після парсингу селектора відбувається парсинг властивостей, для цього викликається метод parse_property. Кожна властивість складається з ім’я та значення, які розділені двокрапкою, а після закінчення властивості має бути крапка з комою. Тому відбувається зчитування ім’я та значення властивостей, і після того вони передаються в метод process_property_members. Цей метод вертає кортеж із двох елементів PropertyName та PropertyValue. На основі

тих даних, які йому надіслали він шукає відповідне ім’я властивості, і повертає відповідний клас. Значення в залежності від типу(колір, довжина, і тд.) окремо парситься функціями. Для кольору функція приймає значення у вигляді стрічки, і вертає об’єкт класу(енаму) Color. Він має 3 значення: hex, rgb, та іменований. Для парсингу довжини використовується функція parse_length, яка повертає об’єкт класу Length, який має 2 значення: відсотки, та пікселі.

2.4 Реалізація структури об’єктів CSS

Корінь всього – це структура Stylesheet, вона містить поле rules, а це вектор об’єктів класу Rule. Структура Rule містить два поля: selector типу Selector та properties типу HashMap<PropertyName, PropertyValue>. Структура Selector містить В CSS селектори є трьох типів: селектори за класом(починаються з крапки: .example-class), селектори за id(починаються з решітки: #example-id) та селектори за тегом(без знанків, просто назва тегу: body). Тому в дипломній було так само реалізовано структуру Selector. В ній наявні три поля: tag_name, id, class. Всі ці поля мають тип Option<String>. Це можна було б реалізувати через enum для простоти, але це би виключило можливість мати комплексні селектори, наприклад які можуть мати як і селектори за класом так і селектори за тегом.

PropertyName – це енам, який містить імена властивостей такі як color, backgroundColor та ін. Він містить метод to_str, який перетворює об’єкт класу PropertyName на його текстову інтерпретацію.

PropertyValue – це енам з даними, який містить 4 поля: Color зі значеням Color, Length із значенням Length, Display із значенням DisplayType та Other із значенням String.

Color – це теж енам з даними. Він має 3 поля:

1) Rgb – це кортеж із трьох елементів типу u8
2) Named – це об’єкт класу String
3) Hex – це значення типу u32

Також в енамі Color реалізований метод get_rgb, який вертає кортеж із трьох елементів типу u8. Якщо об’єкт на якому викликається цей метод є Rgb, тоді просто вертаються значення кортежу Rgb. Якщо це об’єкт типу Named, тоді вертається захардкоджені значення кортежів на кожну текстову інтерпретацію кольору(якщо колір буде red, тоді вернеться такий кортеж: (255, 0, 0)). Якщо об’єкт класу Hex, тоді виконуються обчислення із зсувом біт, та накладання побітової маски 0xFF на кожну змінну. Ось код цього методу:

Length – це також енам з даними. В ньому є 2 поля: Px із значенням i16 та Percent із значенням u8.

2.5 Реалізація структури DOM-дерева

В корені лежить структура Node, яка містить наступні поля:

1) children – список дочірніх елементів
2) node_type – енам NodeType
3) styles – мапа стилів елементу, яка містить PropertyName як ключ та PropertyValue як значення

NodeType – це енам з даними, який може бути трьома об’єктами:

1) Text – містить дані типу String
2) Element – містить дані типу ElementData
3) Comment – містить дані типу String

ElementData – це структура, яка містить 2 поля: це tag_name типу String та attributes типу AttrMap(аліас на HashMap<String, String>.

Ось реалізація:

Структура об’єктів класу Node(DOM-дерево) створюється в структурі HtmlParser. Після цього на дерево додаються стилі з наявного об’єкту класу Stylesheet. Для цього в реалізації структурі Node є метод add_styles, який і приймає на вхід посилання на об’єкт класу Stylesheet. Цей метод в свою чергу викликає інший метод, який називається add_styles_rec. Метод add_styles можна «запускачем» для рекурсивного методу add_styles_rec.

Він, так само як і add_styles, приймає посилання на об’єкт класу Stylesheet. Проте, окрім цього, add_styles_rec приймає ще параметр parent_styles типу &HashMap<PropertyName, PropertyValue>. Це зроблено для того, щоб передавати батьківські стилі до дитячих елементів.

Встановлення стилів кожному елементу відбувається наступним чином:

1) Встановлення стилів за замовчуванням
2) Наслідуванням стилів від батьківського елементу
3) Встановлення стилів, які призначені цьому елементу

Встановлення стилів за замовчуванням встановлює усім елементам стовідсоткову ширину, та всім текстовим елементам "display: inline", як і у справжніх браузерах.

Наслідування стилів було реалізовано тільки для кольору контенту(тексту)

Встановлення стилів, які призначені конкретному елементу відбувається тільки для типу Element, тому що для тип Comment та Text не потребують стилів(стилі для Text визначаються в батьківському об’єкті Element). Встановлення відбувається через цикл, в якому програма проходиться по кожному об’єкту класу Rule, і бере з нього селектор, і якщо щось одне співпаде(назва тегу, класи, id), тоді стиль буде встановлений.

В процесі розробки було важливо бачити яка структура утворюється після парсингу, тому був доданий функція pretty_print, яка приймає посилання

на об’єкт Node, а також розмір відступів типу usize(використовується при рекурсії, тому для старту треба передати 0). Ця функція друкує HTML код із наявного дерева DOM.

Також, окрім add_styles, наявний ще метод add_js, який ініціалізує js. Він приймає на вхід тип String, і викликає функцію js::init(описана в наступному розділі) передаючи туди js код, та посилання на об’єкт, який викликає цю функцію(&self). Виклик цієї ініціалізації відбувається у безіменному колбеку, який викликається через panic::catch_unwind. Це було створено для того, якщо в js якась помилка, то програма не завершувалась аварійно, а продовжувала працювати. Повідомлення про помилку в js буде виведене в консоль.

2.5 Реалізація інтеграції V8 з браузером.

Для інтеграції було використано бібліотеку з crates.io, назва якої - v8. Для початку треба ініціалізувати об’єкт платформи. Це можна зрозбити через функцію v8::new_default_platform. Першим аргументом передається розмір thread_pool. В моєму випадку я передаю 0. Другим аргументом йде idle_task_support, в мене тут false. Після цього платформа ініціалізується методами. Наступний етап – це створення змінних: isolate, scope, global, context.

Приклад ініціалізації:

Далі йде створення та прив’язування колбеку функції log в об’єкті console.

Для створення функції, яка буде доступна з JS треба 4 речі:

1) Колбек(функція в Rust)
2) FunctionTemplate(шаблон функції, куди передається колбек)
3) v8::String(Об’єкт, який відповідає за назву функції)
4) Function(Виводиться з FunctionTemplate)

Для початку треба створити колбек функцію. Вона має приймати 3 параметри:

1) scope(Об’єкт HandleScope, який був ініціалізований з самого початку)
2) args(Аргументи, які приймає функція)
3) rv(Об’єкт ReturnValue, який дозволить повертати значення із функції)

Для реалізації функції log треба отримати значення із змінної args. Після цього вивести його через макорос println!.

Потім треба створити об’єкт FunctionTemplate. При створенні передати об’єкт scope та колбек.

Після цього треба створити об’єкт v8::String, і передати туди scope та назву функції. В даному випадку це буде "log".

Останній етап створення функції – це видобуток її з об’єкту FunctionTemplate. Для цього треба використати функцію get_function.

Функція у нас вже є, але вона нікуди не підв’язана. В оригінальному js вона є у об’єкті console, тому я вирішив зробити так само.

Для створення об’єкту нам треба все те саме, що й при створенні функції, окрім FuncitonTemplate. Спочатку створюємо ідентифікатор об’єкту(ключ). Для цього використовуємо v8::String, куди передаємо scope та значення "console". Потім створюємо сам об’єкт console через v8::Object. Після цього прив’язуємо функцію log до цього обєкту. Тепер нам залишилось прив’язати сам об’єкт до глобального простору(scope).

Наступним етапом створюється об’єкт document, та теги, які наявні в dom. Тут все так само як і з об’єктом console. Проте, тут був використаний цикл для ітерації по об’єктах nodes. До кожного об’єкту node створюється свій відповідник у об’єкті document. Також в кожного об’єкта node створюється мапа styles, яка заповнюється стилями, які є у об’єктів. Доступ до об’єктів через js буде відбуватись наступним чином:
document.body.styles.backgroundColor = 'red';

Якщо використовувати унікальні теги, такі як body чи main, тоді проблем не буде. Проте, якщо в HTML є декілька тегів div, тоді js не буде знати до

якого тегу звертатись(буде звертатись до останнього ініціалізованого тегу div).

Вирішенням цієї проблеми став метод getElementById, який належить об’єкту document. Для цього ми просто створюємо функцію get_by_id_callback, який шукатиме об’єкт node по id, яке міститься у атрибутах. Якщо об’єкт знайдено, то він вертається через rv(ReturnValue):

rv.set(element_obj.into());

Після цього ми підв’язуємо створюємо функцію getElementById, до якої підв’язуємо колбек, а після цього встановлюємо цю фунцію у об’єкт document.

Залишилось тільки скомпілювати(зробити перевірку на помилки) скріпт та запустити його.

2.6 Перетворення DOM-дерева на LayoutBoxes.

Після всіх етапів ми отримало повністю готове DOM-дерево, але є проблема, ми не можемо його так зразу вивести на екран. Елементи мають отримати свої розміри та розташування на екрані. Для цього була створена структура LayoutBox. Ось поля, які вона містить:

1) dimensions – Dimensions(попереднє розташування елемента, без урахування відступів)

2) actual_dimensions – Dimensions(остаточна позація елемета на екрані)
3) content – Option<Content>(Текст, який може міститись в елементі)
4) color – Color(Колір контену елемента)
5) background_color – Color(Колір елементу)
6) name – String(Назва елементу)
7) margin – Indentations(Зовнішні відступи)
8) padding – Indentations(Внутрішні відступи)
9) box_type – BoxType(Тип коробки або Block або Inline)
10) children – Vec<LayoutBox>(Дитячі елементи)
11) v_elements – i16(Кількість елементів із типом Block)
12) h_elements – i16(Кількість елементів із типом Inline)

Структура Dimensions містить 4 поля: x, y, width, height. Всі вони типу i16.

Структура Content містить 3 поля: x, y, text; x та y потрібні, оскільки на текст впливають внутрішні відступи, тому йому потрібно мати свої координати.

Структура Indentations мітить 4 поля: top, right, bottom, left. Всі вони типу i16.

Структура Color містить 4 поля: r, g, b, a. Всі вони типу u8. Також в реалізації наявний метод to_array, який перетворює елементи в типу f32. Це потрібно для рендерингу на екран.

Енам BoxType містить тільки 2 значення: Block або Inline(за замовчуванням Block).

Побудова дерева LayotBox вібувається через функцію build_layout_tree, яка знаходиться у структурі LayoutBox. У функції створюється об’єкт body через функцію-конструктор build_box. Далі викликається функція build_layout_tree_helper, в яку передаються дочірні елементи body, а також мутоване посилання на body та нульовий номер елементу. Ця функція є рекурсивна, вона в циклі проходиться по кожному дитячому вузлу, і якщо він є елементом, а не текстом, тоді створюється його коробка(LayoutBox) та відбувається виклик цієї ж функції. Якщо вузл є текстом, тоді викликається метод set_content, який встановлює батьківському елементу контент на основі переданого тексту, вираховує тексту розташування та розширює батьківський елемент.

Створення коробок на основі вузлів відбувається у функції build_box. Функція приймає посилання на об’єкт структури Node, батьківський елемент, дані елементу та номер елементу. З самого початку створюється об’єкт структури LayoutBox через метод default, а далі змінюється на основі переданого об’єкту структури Node. Власне ці зміни відбуваються в циклі, де інетрується кожен стиль елементу. Стиль – це об'єкт HashMap, де ключами є об'єкти PropertyName, а значеннями – PropertyValue. В кожній ітерації ми розбиваємо стиль на ім’я та значення, і використовуємо ім’я в match case, для того, щоб знайти потрібну властивість. Після того, як

властивість знайдена ми оновлюємо коробці значення, опираючись на значення стилю. Після того, як цикл пройшовся по всіх стилях, викликається метод calculate_position, в який передається мутоване посилання на батька. Першим же чином збільшується висота елемента на основі наявних внутрішніх відступів, а саме верхніх та нижніх. Далі йде розгалуження: якщо елемент блокового типу, тоді початкова координата по осі Y визначається як координата по осі Y батьківського елементу плюс батьківська змінна v_elements(vertical elements). Початкова координата по осі X визначається як справжня координата по осі X у батька. Також до змінної v_elements у батьківського елементу додається висота, верхні та нижні зовнішні відступи поточного елементу. Змінна v_elements зберігає в собі розміри усіх дочірніх елементів по вертикалі. Із додаванням нового елементу вона буде збільшуватись, і наступний елемент буде вимальовуватись зразу після попереднього.

Якщо елемент має тип inline, тоді відбувається все те саме, як і з типом блок, тільки навпаки. Координату за Y елемент отримує таку саму як і батьківський, проте, координата за X обчислюється з урахуванням змінної h_elements(horizontal elements). Тобто додається змінна h_elements до батьківської позиції. І в кінці змінна h_elements збільшується на ширину об’єкта плюс зовнішні відступи зліва та справа.

Обраховується остаточна позиція через метод calculate_actual_dimensions. Він приймає ті ж параметри, що й calculate_position. Тобто це мутоване

посилання на батьківський елемент. Метод містить тільки 4 рядка, на кожен з полів структуру Dimensions:

1) Визначення справжньої координати за X – початкова координата за X плюс зовнішній відступ зліва плюс батьківський внутрішній відступ зліва.
2) Визначення справжнього координати за Y – початкова координата за Y плюс зовнішній відступ зверху плюс внутрішній батьківський відступ зверху
3) Визначення справжньої ширини – початкова ширина мінус зовнішні відступи зліва та справа та мінус внутрішній батьківський відступ справа помножений на 2.
4) Визначення справжньої висоти відбувається копіюванням значення початкової висоти(значення висоти не змінюється)

В результаті цих дій функція build_layout_tree вертає список коробок(LayoutBox), які далі будуть використані у рендерингу.

2.7 Виведення на екран(рендеринг)

Над вибором графічної бібліотеки було проведено багато часу, оскільки Rust – мова молода, і в основному не використовується для створення GUI. В загальному бібліотеки можна поділити на 2 типи: високорівневі та низькорівневі. Для того, щоб сконцентруватись на написанні самого рушія

для браузера було прийнято рішення взяти бібліотеку Iced. Ця бібліотека дуже добре підходить для написання графічних інтерфейсів для десктопних додатків. Проте, є одна проблема, яка б завадила використовувати цю бібліотеку в цій кваліфікаційній роботі. В цій бібліотеці є компоненти(так само як в React), і вимальовування елментнтів на екран, та надання їм координатів відбувається так само як в HTML. З одної сторони – це добре, оскільки не прийшлось би писати код, який розставляє елементи на свої місця, а з іншої, я не зможу каскадними таблицями стилів(CSS) змінювати коректно позиції, і також не зможу підключити JS.

Після цього вибір впав на бібліотеку egui.

Бібліотека egui - це незалежний від системи іммедіатний інтерфейс користувача, що написаний на Rust. Це вільна і відкрита альтернатива IMGUI-системам, таким як dear imgui, але із зосередженням на більш безпечних та зручних для використання Rust-ідіомах.

egui дозволяє створювати гнучкі і динамічні інтерфейси з простим синтаксисом і невеликою кількістю коду. Він включає в себе різноманітні віджети, такі як кнопки, слайдери, текстові поля, панелі та багато інших. Однак, незважаючи на багатий набір віджетів, egui простий у вивченні і використанні, що робить його прекрасним вибором для швидкої розробки і прототипування інтерфейсу користувача.

Однак, egui не підходить для браузера. По-перше, egui працює через WebGL, що потребує високого рівня контролю над відеокартою, що є непрактичним в контексті браузера. По-друге, egui працює в іммедіатному режимі, який відрізняється від більш традиційного ретейнового режиму, який використовується в браузері. Це може створювати конфлікти при спробах інтеграції та це не дало б можливості підключитись до JS.

Після цього вибір впав на рушій для ігор – Bevy. Bevy - це високопродуктивний, що розробляється, рушій для створення ігор і інтерактивного контенту, що написаний на Rust. Він використовує сучасний ECS (Entity-Component-System) для керування даними та поведінкою об'єктів в сцені. Це робить Bevy дуже гнучким та масштабованим, дозволяючи легко створювати складні інтерактивні сцени з великою кількістю об'єктів.

Додатково, Bevy включає різноманітні модулі для роботи з графікою, звуком, вводом користувача, фізикою, мережею і так далі. Це робить Bevy могутнім інструментом для створення різноманітних видів ігор та інтерактивного контенту.

Однак, Bevy не був розроблений специфічно для створення графічних користувацьких інтерфейсів (GUI), а був створений як рушій для ігор. Хоча він має деякі можливості для створення GUI, вони не є основною

ціллю і можуть бути обмеженими в порівнянні з бібліотеками, спеціально розробленими для цього.

Тоді вибір впав на бібліотеку Tauri.

Tauri - це фреймворк для створення легких, безпечних і високопродуктивних веб-додатків, використовуючи веб-технології та Rust. Він дає можливість розробникам створювати нативні додатки, використовуючи веб-стек (HTML, CSS і JavaScript), а потім загортати ці веб-додатки в безпечні, мінімалістичні бінарники за допомогою Rust.

Однією з ключових особливостей Tauri є його розмір і продуктивність. В порівнянні з аналогічними фреймворками, такими як Electron, додатки на Tauri мають значно менший розмір бінарного файлу та менші вимоги до системних ресурсів, що дозволяє створювати швидкісні і ефективні додатки.

Також, безпечність є важливою частиною філософії Tauri. Фреймворк містить ряд безпекових функцій, таких як ізоляція контексту, покращений сендбоксинг, безпечне виконання команд та ін.

Щодо створення браузера за допомогою Tauri, це може бути досить складною задачею. З однієї сторони, можна брати HTML файл, і рендерити за допомогою Tauri, але тоді це не вийде браузер, це вийде просто UI обгортка до вже наявного рушія. З іншої сторони, можна намагатись рендерити HTML з расту, а HTML файл з Tauri використати

виключно для стилізації. Проте, мені такий підхід здався дивним. По факту буде запущено 2 рушія: один з Tauri і один написаний мною. Тому, хоч дуже й хотілось взяти цю бібліотеку, але, все ж, прийшлось відмовитись.

Тому, зрозумівши, що варто брати більш низкорівневу бібліотеку вибір впав на OpenGL та Vulkan. Подивившись порівняння цих бібліотек вирішив взяти Vulkan, оскільки він в рази швидший. Проте, коли я побачив скільки всього треба зробити, щоб вивести один трикутник на екран(150-200 рядків коду) зразу відкинув цей варіант. Тому вже остаточний вибір впав на OpenGL. Як такої OpenGL на Rust нема. Є тільки rust-обгортки. Найпопулярніші є glium та gfx. Оскільки glium новіша, тому вирішив взяти її. Вдалось виводити один елемент на екран, але з багатьма була проблема. Коли я розбирався з цією бібліотекою, документація по вимальовуванню декількох компонентів ще не було, тому було прийнято рішення закинути цю бібліотеку, і перейти до старішої обгортки над OpenGL – GFX.

Для початку я створюю константи ширини, висоти, та базового кольору(фон). Потім, використовуючи макрос gfx_defines! визначаю базовий тип вимальовки – вершина(Vertex). Вона в собі містить 2 поля:

1) pos – масив з 2 елементів (x, y) типу f32(float)
2) color – масив із 4 елементів (r, g, b, a) типу f32

Після вершини йде ініціалізація пайплайну, тут також міститься 2 поля:

1) vbuf – це вершинний буфер, що зберігає дані вершин структури Vertex
2) out – це вихідний буфер, в якому міститеметься результат рендерингу

Для роботи із OpenGL треба ініціалізувати 2 шейдери:

1) box.glslf – цей шейдер приймає колір від вершинного шейдера (v_Color) і використовує його як колір для поточного пікселя (gl_FragColor). Це визначає кінцевий колір кожного пікселя на зображенні.
2) box.glslv – це вершинний шейдер. Він потрібен для рендерингу вершин. Ось, що означають його складові:
   - attribute vec2 a_Pos; - описує 2D позицію вершини.
   - attribute vec4 a_Color; - описує колір вершини в RGBA форматі.
   - varying vec4 v_Color; - встановлює колір, який буде переданий до фрагментного шейдера.
   - v_Color = vec4(a_Color); - передає колір вершини до фрагментного шейдера.

- gl_Position = vec4(a_Pos, 0.0, 1.0); - встановлює позицію вершини у 3D просторі. Оскільки в моєму браузері використовується тільки 2 площини x та y, тому значення z стоїться на нулі. Останнє значення 1.0 – це w вона потрібна для підтримки перетворень, таких як масштабування, поворотів та перенесень. Тому я залишив стандартне значення, яке рекомендують в документації, коли задача не стоїть щось масштабувати чи переносити.

Для рендерингу використовується функція render, яка приймає список коробок(LayoutBox). Хоч це й список коробок, але в ньому міститься тільки один елемент(body), всі інші елементи зберігаються всередині body у вигляді дерева. З деревом працювати не дуже зручно, тому треба перетворити дерево на чистий список. Для цього була написана функція layout_box_tree_to_vector. Це рекурсивна функція, тому вона приймає список коробок(якраз тому й у функцію render передається список з одною коробкою), і вертає теж список коробок. З самого початку у функції створюється список пустий список коробок, до якого в циклі додаються елементи зі списку, що був переданий в цю функцію, і після цього в цьому ж циклі запускається рекурсія: у функцію передаються дочірні елементи коробки. Результат роботи функції – це список, який додається до наявного списку створеного на початку функції, і після цього, цикл завершується та функція повертає цей список.

Наступний етап у рендері – це створення трьох списків:

1) список вершин - vertices
2) список байтів індексів – idex_data
3) список текстових елементів – text_vec(кортеж із 3 елементів, текст, масив з 2 елементами(координати), колір)

Всі ці списки створюються пустими. Далі йде цикл, в якому ми ітеруємось по коробках. Щоб наповнити список вершин ми викликаємо функцію render_content, яка приймає посилання на коробку, і вертає список вершин. Так як в об’єкті LayoutBox координати зберігаються в цілих числах, а OpenGL підтримує тільки числа з плаваючою крапкою. Для цього була створена функція transform_rectangle, яка також приймає на вхід посилання на об’єкт коробки. Функція, використовуючи константи висоти та ширини, перетворює координати коробки на кортеж з 4 елементів типу f32.

Після цього функція render_content працює вже з 4 елементами з типом числа з плаваючою крапкою:

1) w – ширина
2) h – висота
3) x – координата за віссю X
4) y – координата за віссю Y

Функція має повернути список із 4 вершин, які містять координати та колір. Координати для кожної вершини вираховуються окремо, а колір для всіх береться той самий – це фоновий колір конкретної коробки. На фоновому кольорі викликається метод to_array, який перетворює значення байтів (u8) в значення, які приймає OpenGL, тобто це тип f32.

1) Перша вершина отримує координати такі як і в коробки, тобто просто x, y
2) Орієнтація осей у OpenGL відбувається від 1.0 до -1.0. Зверху вісь y має значення 1.0, а знизу -1.0. Тому для вимальовування другої вершини x залишаємо на міці, і рухаємось вниз. Для цього потрібно від значення y відняти висоту коробки.
3) Третя ж вершина рухається по двох осях. До x додаємо ширину, а до y, як і минулого разу віднімаємо y.
4) Остання ж вершина рухається тільки по вісі x. Для цього просто додаємо до x ширину.

Далі отриманий список додаємо до вже наявного списку vertices. В кожній ітерації створюємо змінну index_base – ця змінна – це номер ітерації помножений на 4. Потім додаємо у список індексів масив із 6 елементів: чисту змінну index_base, змінна збільшена на 1, збільшена на 2, збільшена знову на 2, збільшена на 3, чиста змінна.

Наступний етап – це перевірка на те, що коробка має контент(текст), якщо це істина, тоді додаємо в список текстів цей текст.

Ось весь коді циклу:

Далі нам треба створити вікно. Для створення вікна було використано бібліотеку glutin.

Glutin - це бібліотека для Rust, яка слугує для створення і керування вікнами і подіями. Її основна мета - надати простий, мінімальний, прямий інтерфейс до створення вікна для OpenGL та подій, таких як натискання клавіш та рухи миші. Це альтернатива більш традиційним бібліотекам для створення вікон та обробки подій, таким як GLFW чи SDL, але її особливістю є те, що вона написана повністю на Rust і вона прямо інтегрується з іншими бібліотеками Rust, такими як gfx або winit. Glutin забезпечує функціональність для створення вікна, визначення його розміру та положення, обробки подій вводу, таких як клавіатура, миша та сенсорний екран, та управління контекстом OpenGL для вікна.

Після створення об’єкту класу WindowBuilder треба ініціалізувати OpenGL через наявний builder. Після нам треба завантажити шейдери через макрос include_bytes!. Далі йде ініціалізація бібліотеки gfx_text. Ця бібліотека не входить в gfx, тому її треба ініціалізувати та підключати окремо.

gfx_text дозволяє легко додавати текст до графічних проєктів, використовуючи функції gfx для рендерингу тексту. Вона включає в себе підтримку широкого спектра символів, включаючи Unicode, та дозволяє виконувати різноманітні стилізації та масштабування тексту.

Ця бібліотека добре інтегрована з gfx і дозволяє використовувати текст як частину графічного рендерингу в gfx, надаючи можливість легко додавати елементи інтерфейсу, описи та інші текстові елементи до графічних додатків.

Для цього ми створюємо об’єкт Renderer, який через метод gfx_text::new, куди передаємо раніше створений об’єкт factory, який ми отримали із glutin.

Основний цикл – це while, який працює поки змінна running має правдиве значення. Бібліотека glutin дає нам багато івентів з вікном, але я використав тільки 2 – це кнопка Escape та закривання вікна мишкою. В обох випадках воно переводить змінну running в значення false, і програма зупиняється.

Кожну ітерацію основного циклу ми стираємо зміст на екрані, і вимальовуємо знову. Потім ми вимальовуємо весь текст, який є у списку текстів. Щоб не створювати надмірне навантаження на процесор – я після вимальовування помістив функцію sleep, в яку передав значення 10 мс.

2.8 Демонстрація роботи

На даному знімку екрану відображається файл запуску браузера. Для того, щоб запустити браузер, треба створити об’єкт структури Browser, і передати текст html файлу. Для цього спочатку треба зчитати вміст файлу index.html. Оскільки файл лежить в корені проєкту, тому треба створити об’єкт структури PathBuf через метод env::current_dir, та «розпакувати» об’єкт через unwrap. Далі треба через метод push треба зайти у файл index.html, та після цього зчитати його через об’єкт структури BufReader, та записати це значення у змінну html_input. Після цього передаємо цю змінну при створенні об’єкту Browser, та запускаємо рендеринг через метод run.

index.html:

style.css:

Ось результат рендерингу:

А це результат рендерингу браузера Chrome:

Відображені елементи не сильно відрізняються. Різниця полягає тільки у відступах(margin: 10px), які браузер Chrome додає автоматично до будь

елементу body, а також різниця є в кольорах. Це відбувається через те, що Chrome та бібліотка gfx мають різні колірні гами. Ось до прикладу різниця колірних гамм між Adobe RGB та sRGB:

Висновки

В ході виконання дипломної роботи було проведено розробку та реалізацію веб-браузера з використанням мови програмування Rust. Детально розглянуто архітектуру браузера та етапи його роботи, включаючи парсинг HTML та CSS, реалізацію DOM-структури та структури об'єктів CSS.

Особлива увага приділена розробці алгоритму парсингу HTML, який було реалізовано з дотриманням стандартів W3C. Було також виконано розробку парсера CSS, що підтримує основні селектори та властивості.

Складовою роботи є інтеграція з JavaScript-движком V8, що дозволило забезпечити повноцінну підтримку JavaScript в браузері.

Подальша реалізація передбачала створення структури DOM-дерева на основі отриманих HTML та CSS даних, а також перетворення DOM-дерева на LayoutBoxes для подальшого відображення в браузері.

Завершальним етапом стала реалізація рендерингу, що включала відображення LayoutBoxes на екрані.

Всі етапи роботи браузера було реалізовано враховуючи оптимізацію та ефективність.

Результатом роботи став повнофункціональний веб-браузер, створений за допомогою Rust, який відображає HTML сторінки, інтерпретує CSS стилі та підтримує виконання JavaScript коду.

У майбутньому, планується розширення функціоналу браузера, включаючи підтримку роботи з мережею, куки, історією перегляду, та іншими важливими компонентами сучасного веб-браузера.

Список використаних джерел

1. The Rust Programming Language [Електронний ресурс] // Керол Ніколс та Стів Клабник // 2018 – Режим доступу до ресурсу: https://doc.rust-lang.org/stable/book
2. Engineering the servo web browser engine using Rust [Електронний ресурс] // Brian Anderson, Lars Bergstrom // 2016 – Режим доступу до ресурсу: https://dl.acm.org/doi/abs/10.1145/2889160.2889229
3. How to make a semantic web browser [Електронний ресурс] // R. Karger, D. A. Quan// 2004 – Режим доступу до ресурсу: https://dl.acm.org/doi/abs/10.1145/988672.988707
4. The Cost of Speculation: Revisiting Overheads in the V8 JavaScript Engine [Електронний ресурс] // Alberto Parravicini, Rene Mueller // 2021 – Режим доступу до ресурсу:

https://ieeexplore.ieee.org/abstract/document/9668283

5. JavaScript docs Mozilla [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript
6. OpenGL Programming Guide: The Official Guide to Learning OpenGL [Електронний ресурс] // Dave Shreiner // 2009 – Режим доступу до ресурсу: https://books.google.com.ua/books?hl=uk&lr=&id=xPu3mN2FPl4C&oi=fnd&pg=PT24&dq=opengl&ots=mRcXdo2avm&sig=JgI61cU2wabQ6hCceykLCnIM40Q&redir_esc=y#v=onepage&q=opengl&f=false
7. Що таке браузер і як він працює. Історія розвитку [Електронний ресурс] // Владислава Рикова // 2021 – Режим доступу до ресурсу: https://vlada-rykova.com/ua/idei-giperteksta-i-gipermedia-sistemy-xanadu-i-world-wide-web/
8. A Brief History of Web Browsers and How The Work [Електронний ресурс] // Alex McPeak// 2018 – Режим доступу до ресурсу - https://smartbear.com/blog/history-of-web-browsers/