Evaluación del rendimiento de WebAssembly como tecnología para la optimización del renderizado de imágenes en el desarrollo web del lado del cliente: caso de estudio de Ray Tracing

Daniel Felipe Jaimes Blanco

Trabajo de Grado para optar al título de Ingeniero de Sistemas

Director

Gabriel Rodrigo Pedraza Ferreira

Doctor en Ciencias de la Computación

Codirector

Jathinson Meneses Mendoza

Magister en Gestión, Aplicación y Desarrollo de Software

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería de Sistemas e Informática

Bucaramanga

2023

Dedicatoria

Dedico este proyecto a mi familia, por su apoyo inquebrantable y amor constante, y a todos aquellos que creen en el poder de la tecnología para cambiar el mundo. Sin ustedes, nada de esto sería posible. Espero ser parte de la innovación en un futuro y contribuir al progreso juntos.

Agradecimientos

Quiero expresar mi sincero agradecimiento a todos aquellos que han sido parte de este proyecto.

En primer lugar, quiero agradecer a mi familia por su apoyo constante durante todo el proceso. Su presencia y apoyo fueron fundamentales para mantenerme enfocado y motivado.

Agradezco profundamente a mis asesores, por su orientación experta y su inestimable ayuda en el desarrollo de este proyecto. Sus conocimientos y consejos fueron fundamentales para alcanzar nuestros objetivos y llevar a cabo esta investigación de manera exitosa.

No puedo pasar por alto la comunidad de desarrolladores y entusiastas de WebAssembly y Rust en Reddit. Sus respuestas a mis preguntas y su disposición para compartir su experiencia fueron un recurso inestimable y enriquecedor. Agradezco a cada uno de ustedes por contribuir al conocimiento y el aprendizaje colectivo en este campo.

Además, quiero expresar mi gratitud a Mozilla por la creación de Rust. Este lenguaje no solo desempeñó un papel crucial en este proyecto, sino que también me he enseñado a ver la programación de una manera nueva y emocionante. Gracias por ser tan geniales y por guiar mi camino hacia un desarrollo seguro y eficiente.

Aprecio profundamente la comunidad de desarrollo y las mentes brillantes que han contribuido a expandir los horizontes de la tecnología. Gracias por ser parte de este emocionante viaje y por inspirar nuevas posibilidades en el mundo de la programación y la web.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Glosario

Ray Tracing Técnica de renderizado que simula la forma en que la luz interactúa con los objetos en una escena 3D, utilizada en gráficos por computadora para crear imágenes realistas.

WebAssembly Un estándar web que permite en los navegadores web la ejecución de código de alto rendimiento, escrito en lenguajes como C, C++, Rust, AssemblyScript, Haskell, entre otros.

Rendimiento La medida de qué tan eficaz y rápido es un sistema o programa en términos de velocidad y capacidad de respuesta.

Algoritmo Un conjunto de pasos lógicos y definidos que se utilizan para resolver un problema o realizar una tarea específica.

Compilación El proceso de convertir código fuente legible por humanos en código ejecutable por un computador.

Optimización Proceso de mejora del rendimiento y la eficiencia de un sistema o algoritmo.

Formato de Instrucción Binaria Representación de las instrucciones de un programa en un formato de datos binarios, utilizado en el contexto de WebAssembly.

Implementación Ordinaria Se refiere a la programación o desarrollo de una solución de software de manera estándar o convencional, sin aplicar las mejores técnicas avanzadas o métodos de

optimización. Una implementación ordinaria es funcional y cumple con los requisitos, pero puede no ser la más eficiente o escalable.

Campo de Visión Ángulo que determina el rango visible desde la cámara en una escena tridimensional.

Lambert Shading Técnica de sombreado que modela la iluminación difusa en superficies mate.

Intersección de Rayos Proceso de determinar dónde un rayo de luz intersecta objetos en una escena.

Modelado de Iluminación Cálculo de cómo la luz interactúa con los objetos en una escena para determinar su apariencia.

Escena 3D Representación virtual tridimensional de un entorno con objetos, luces y cámaras.

Coeficientes de Iluminación Parámetros que controlan cómo se refleja la luz en una superficie, incluyendo el coeficiente de reflexión especular, el coeficiente de Lambert y el coeficiente ambiental.

Reflectividad La capacidad de un objeto para reflejar la luz incidente.

Fotograma Una imagen individual en una secuencia de imágenes que crea la ilusión de movimiento en animación.

Renderizado en Tiempo Real Proceso de generar imágenes o gráficos a medida que cambian en tiempo real en lugar de precalcularlos.

Píxel La unidad más pequeña de una imagen digital, representada por un solo punto en una matriz.

Ángulo de Incidencia El ángulo entre un rayo de luz incidente y la superficie en la que incide.

Resumen

Título: Evaluación del rendimiento de WebAssembly como tecnología para la optimización del renderizado de imágenes en el desarrollo web del lado del cliente: caso de estudio de Ray Tracing [*]

Autor: Daniel Felipe Jaimes Blanco [**]

Palabras Clave: WebAssembly, Ray Tracing, Desarrollo web, JavaScript, Rust, Evaluación de rendimiento, Tiempo de ejecución, Wasm, Optimización.

Descripción: Este proyecto se enfocó en la evaluación de la idoneidad de la tecnología WebAssembly (Wasm) como una herramienta para optimizar el renderizado de imágenes en el desarrollo web del lado del cliente. Se llevaron a cabo dos fases: la primera se centró en la implementación y comparación de algoritmos en lenguajes de alto nivel compilados a Wasm, mientras que la segunda fase se enfocó en la implementación de un motor de Ray Tracing en tiempo real utilizando WebAssembly y JavaScript. Los resultados revelaron que Wasm ofrece un rendimiento prometedor en comparación con JavaScript, especialmente en aplicaciones intensivas en cómputo como el Ray Tracing. Además, se observó que el rendimiento variaba según el navegador y el sistema operativo, con Firefox y macOS mostrando un desempeño particularmente sólido. Este proyecto destaca el potencial de Wasm en el desarrollo web del lado del cliente y sugiere que esta tecnología merece una atención más amplia en futuros proyectos de optimización y renderizado web.

---

[*] Trabajo de grado

[**] Facultad de Ingenierías Fisicomecánicas. Escuela de Ingeniería de Sistemas e Informática. Director: Gabriel Rodrigo Pedraza Ferreira. Codirector: Jathinson Meneses Mendoza

Abstract

Title: Performance Evaluation of WebAssembly as a Technology for Client-Side Web Development Image Rendering Optimization: A Case Study of Ray Tracing [*]

Author: Daniel Felipe Jaimes Blanco [**]

Keywords: WebAssembly, Ray Tracing, Web Development, JavaScript, Rust, Performance Evaluation, Runtime, Wasm, Optimization.

Abstract: This project focused on evaluating the suitability of WebAssembly (Wasm) technology as a tool for optimizing client-side web development image rendering. Two phases were carried out: the first focused on the implementation and comparison of algorithms in high-level languages compiled to Wasm, while the second phase focused on the implementation of a real-time Ray Tracing engine using WebAssembly and JavaScript. The results revealed that Wasm offers promising performance compared to JavaScript, especially in computationally intensive applications like Ray Tracing. Furthermore, performance varied depending on the browser and operating system, with Firefox and macOS showing particularly strong performance. This project highlights the potential of Wasm in client-side web development and suggests that this technology deserves wider attention in future web optimization and rendering projects.

---

[*] Bachelor's Thesis

[**] Faculty of Physicomechanical Engineering. School of Systems and Computer Engineering. Supervisor: Gabriel Rodrigo Pedraza Ferreira. Co-Supervisor: Jathinson Meneses Mendoza

Introducción

En el actual panorama del desarrollo web, las aplicaciones del lado del cliente han experimentado una notable evolución, incrementando su complejidad y sofisticación con el objetivo de brindar experiencias de usuario cada vez más satisfactorias. Sin embargo, este crecimiento en complejidad también ha introducido desafíos relacionados con el rendimiento y la eficiencia en el procesamiento de tareas intensivas, como el renderizado de imágenes a través ray tracing. Las tecnologías tradicionales, como JavaScript, que dominan ampliamente la web, enfrentan limitaciones significativas en términos de eficiencia, velocidad y seguridad en este tipo de procesamiento.

En este contexto, surge WebAssembly como una tecnología prometedora capaz de abordar los desafíos de rendimiento y eficiencia en aplicaciones web del lado del cliente.Tomando su definición: "WebAssembly (abreviado Wasm) es un formato de instrucción binaria para una máquina virtual basada en pila. Wasm está diseñado como un destino de compilación portátil para lenguajes de programación, lo que permite la implementación en la web para aplicaciones cliente y servidor"(WebAssembly, 2023). En otras palabras, Wasm es un formato binario en el que podemos compilar otros lenguajes como C/C++, Rust, Go, Java, entre muchos otros, para posteriormente ser ejecutados en el navegador web. Esta tecnología permite a los desarrolladores aprovechar las ventajas del rendimiento de bajo nivel sin sacrificar la portabilidad y seguridad inherentes a las aplicaciones web modernas (Mozilla, 2023).

En el presente proyecto de investigación, se realiza un enfoque centrado en una evaluación exhaustiva de la idoneidad de WebAssembly en el contexto del desarrollo web del lado del cliente con el abordaje principal del renderizado de imágenes de alta calidad en aplicaciones web del lado del cliente. El caso de estudio elegido es la implementación de un algoritmo de ray tracing, una concepto avanzado y ampliamente utilizado en campos como el diseño gráfico, los videojuegos y la visualización científica para generar imágenes realistas y detalladas. Sin embargo, su complejidad computacional y demanda de recursos representa un desafío para el procesamiento en tiempo real en aplicaciones web. La motivación de realizar esto es poder explorar cómo WebAssembly puede impulsar la optimización en el desarrollo web evaluando su implementación en escenarios como el proceso de renderizado y, por ende, mejorar tanto el rendimiento de la aplicación como la experiencia del usuario.

Para alcanzar estos objetivos, se dividió este proyecto en dos fases estratégicas. En la primera fase, se realiza una comparativa meticulosa de algoritmos, implementados en diversos lenguajes, tales como JavaScript, Rust y C. Se llevará a cabo pruebas exhaustivas para medir el tiempo de ejecución, la eficiencia y la escalabilidad de cada implementación. Esta fase permitirá realizar un estudio preliminar de las tecnologías involucradas e identificar las implementaciones más prometedoras y seleccionar las opciones óptimas para el caso de estudio.

La segunda fase se centrará específicamente en el caso de estudio del ray tracer y abordará el análisis comparativo de las implementaciones de WebAssembly y JavaScript en este contexto. El enfoque principal de este análisis estará dirigido a ver cuánto podía exigir a la web a través de

implementaciones ordinarias para poder evaluar el rendimiento, la eficiencia y la calidad de los resultados generados. Durante esta etapa, se medirá variables críticas, como el tiempo de renderizado, los fotogramas por segundo (FPS) y la precisión visual, entre otras métricas relevantes. El objetivo principal será cuantificar el rendimiento y la calidad de las imágenes generadas para determinar si WebAssembly es una opción viable para implementar en la web para escenarios exigentes en cómputo como el renderizado de imágenes de alta calidad en aplicaciones web del lado del cliente. A través de este enfoque riguroso y detallado, se aspira a obtener una comprensión profunda de las capacidades y limitaciones de WebAssembly en el contexto de aplicaciones web.

Al análisis de los resultados obtenidos en ambas fases se espera obtener conclusiones fundamentadas y resultados significativos que puedan contribuir al avance tecnológico y brindar información valiosa para futuros proyectos. Esto no solo aportará conocimientos valiosos al desarrollo web del lado del cliente, sino que también permitirá a la comunidad de desarrolladores e investigadores comprender mejor el potencial de WebAssembly como tecnología para el desarrollo web del lado del cliente. Con este enfoque meticuloso y riguroso, se aspira a contribuir al avance tecnológico y proporcionar herramientas útiles para futuros proyectos en el campo de las aplicaciones web de sofisticada complejidad.

1. Planteamiento y Justificación del Problema

En las últimas décadas, el desarrollo de aplicaciones web del lado del cliente ha experimentado una evolución notable debido al avance tecnológico y las crecientes expectativas del mercado. Estas aplicaciones se han vuelto cada vez más complejas, lo que ha generado una demanda creciente en términos de eficiencia y seguridad del código en la web (Haas et al., 2017). Sin embargo, esta complejidad ha introducido desafíos significativos, especialmente en el campo de la exigencia computacional como la generación de imágenes realistas y detalladas mediante técnicas de Ray Tracing. El proceso de renderizado por Ray Tracing es conocido por su intensivo uso de recursos, lo que puede tener un impacto negativo en el rendimiento y la eficiencia de la aplicación, convirtiéndose en factores críticos para garantizar una experiencia de usuario óptima.

Si bien JavaScript es ampliamente utilizado en el desarrollo web debido a su capacidad para interactuar con aplicaciones web, su capacidad de procesamiento no es suficientemente eficiente cuando se trata de altos procesos de cálculo como lo es renderizar imágenes con Ray Tracing. Esta ineficiencia puede dar lugar a problemas de desempeño en las aplicaciones web.

Aunque existen alternativas, como las librerías de renderizado en el servidor, su complejidad y costos adicionales pueden limitar su viabilidad en implementaciones a gran escala. Por lo tanto, es esencial explorar nuevas tecnologías que permitan una optimización más eficiente y rentable del proceso de renderizado en el cliente.

Es imperativo que todos los involucrados en el desarrollo, incluidos los ingenieros de sistemas, estén al tanto de las nuevas tecnologías y evalúen su potencial para mejorar los procesos de desarrollo como las aplicaciones web. Esto les permitirá mejorar el rendimiento y la eficiencia de las aplicaciones, lo que es fundamental en un entorno de desarrollo web en constante evolución.

En este contexto, WebAssembly (Wasm) ha surgido como una tecnología prometedora para impulsar significativamente el rendimiento y la eficiencia en el desarrollo de aplicaciones web del lado del cliente. El uso de WebAssembly abre un amplio abanico de posibilidades, como la capacidad de aprovechar directamente desde el navegador la aceleración por hardware, la creación de aplicaciones en lenguajes de programación no soportados directamente por los navegadores y la creación de aplicaciones web con una experiencia de usuario similar a aplicaciones de escritorio.

Por lo tanto, este proyecto se enfoca en evaluar a WebAssembly, tomando principalmente como caso de estudio la implementación de un ray tracer en un entorno web. Este análisis no solo permitirá mejorar el desarrollo de aplicaciones web con Ray Tracing, sino que también se espera poder contribuir a la comprensión del potencial de WebAssembly en términos de mantenibilidad, escalabilidad y facilidad de uso como tecnología para futuras aplicaciones web del lado del cliente.

2. Objetivos

Objetivo general

- Evaluar la idoneidad de la tecnología WebAssembly para mejorar el proceso de renderizado de imágenes mediante Ray Tracing en aplicaciones web del lado del cliente, mediante la implementación de un marco de evaluación utilizando un caso de estudio.

Objetivos específicos

- Caracterizar lenguajes de alto nivel implementados en WebAssembly y comparar su rendimiento en una variedad de algoritmos.
- Establecer un escenario específico para el caso de estudio y definir las métricas de referencia utilizando una implementación en JavaScript.
- Desarrollar una implementación del caso de estudio utilizando WebAssembly.
- Realizar un análisis comparativo exhaustivo de los resultados obtenidos en ambas implementaciones.

3. Marco de Referencia

En esta sección, se proporcionará un contexto esencial para comprender y analizar el proyecto de investigación. Se abordarán temas fundamentales como el Ray Tracing y su relevancia en el mundo computacional, el desarrollo de las aplicaciones web y el impacto de WebAssembly en el mercado actual. Además, se explorarán estudios previos y casos destacados relacionados con el tema.

3.1. Introducción al ray tracing

En el ámbito del renderizado en tiempo real y la generación de imágenes computacionales, existen varias técnicas diferentes[1], cada una con sus propias ventajas y desventajas. Algunas de estas técnicas incluyen el ray casting, radiosity y ray marching, por mencionar algunas. Cada una de estas técnicas tiene aplicaciones específicas y puede ser adecuada en diferentes contextos.

En este proyecto, hemos elegido enfocarnos en el ray tracing como nuestra técnica principal de renderizado. Esta elección se basa en varias razones importantes. El ray tracing es conocido por su capacidad para producir resultados visualmente impresionantes al simular con precisión la forma en que la luz interactúa con los objetos en una escena. Además, el ray tracing ofrece la

---

[1] ¿Qué es el renderizado 3D?: https://3dalia.com/que-es-el-renderizado-3d/

flexibilidad necesaria para simular efectos avanzados, como reflexiones y sombras realistas, lo que lo convierte en una elección atractiva para aplicaciones de renderizado de alta calidad.

A pesar de que el ray tracing no es la forma más rápida de generar imágenes en 3D, su atractivo radica tanto en los efectos realistas que se pueden lograr como en la elegancia de la simplicidad de la técnica. Esto nos permite explorar la intersección entre el rendimiento y la calidad visual en el contexto de la web, lo que lo convierte en un caso de estudio valioso para evaluar el potencial de WebAssembly en aplicaciones de renderizado en tiempo real.

El Ray Tracing es una técnica avanzada de renderizado ampliamente utilizada en gráficos por computadora para generar imágenes fotorrealistas a partir de la simulación del comportamiento de la luz en una escena tridimensional (Rademacher, 1997). Desde su concepción, esta técnica ha sido esencial para la creación de efectos visuales impresionantes en una variedad de campos, incluyendo los profesionales, académicos y de entretenimiento, así como los videojuegos y la simulación de ambientes virtuales (Shirley & Morley, 2008).

Figura 1. Ejemplo ray tracing videojuego Minecraft

En su esencia, el Ray Tracing es una técnica de renderizado que traza rayos de luz virtuales desde el punto de vista de la cámara, que determina su vista en la escena tridimensional. Estos rayos se propagan desde la posición de la cámara a través del plano de visualización 2D, que representa la superficie de la imagen o el plano de píxeles. A medida que estos rayos avanzan, interactúan con los objetos presentes en la escena 3D, y es en estas interacciones donde reside la magia del Ray Tracing.

Cada rayo lanzado desde la cámara representa un rayo de luz que simula el camino que la luz real tomaría si viajara desde la cámara hacia la escena. A medida que estos rayos alcanzan los objetos en la escena, se calcula cómo interactúan con cada superficie. Esto incluye determinar si un rayo golpea un objeto, cómo se refleja o refracta la luz en función de las propiedades de los materiales, y cómo se ilumina una superficie en particular.

El proceso completo de Ray Tracing es un cálculo extremadamente preciso y detallado que involucra un seguimiento meticuloso de cada rayo de luz a medida que viaja a través de la escena y

interactúa con los objetos. Estas interacciones individuales se acumulan para crear la iluminación y el color finales de cada píxel en la imagen resultante. El resultado es una representación visual asombrosamente realista de la escena, con efectos de luz, sombras y reflexiones que se asemejan estrechamente a cómo los objetos interactúan con la luz en el mundo real (Buck & Hogan, 2019).

Este enfoque de cálculo preciso y detallado es lo que distingue al Ray Tracing y lo que permite la creación de imágenes de alta calidad y una experiencia visual impactante en campos como la industria del entretenimiento, el diseño gráfico y los videojuegos.

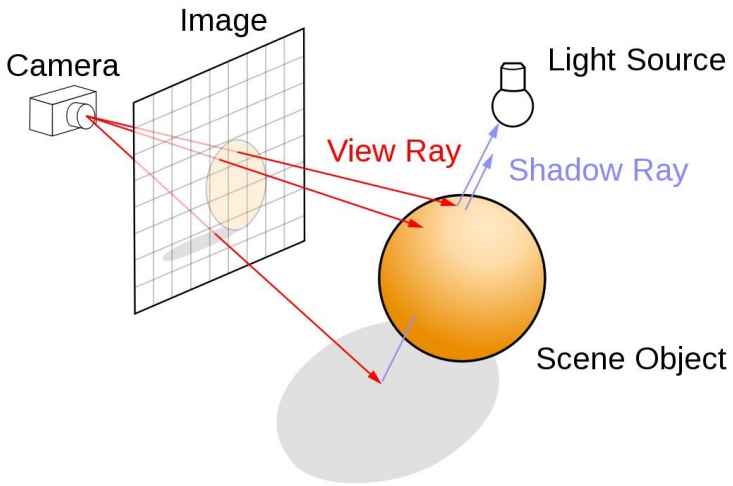


Figura 2. Concepto básico de ray tracing[2]

El objetivo principal del Ray Tracing es obtener resultados altamente precisos y visuales de alta calidad, incluso en escenas complejas con múltiples fuentes de luz, materiales diversos y efectos especiales. Esta técnica ha sido fundamental en la producción de efectos visuales para pe-

---

[2] Ray Tracing By Nvidia: https://developer.nvidia.com/discover/ray-tracing

lículas y animaciones, así como en la creación de escenarios inmersivos y realistas en videojuegos modernos.

Con el avance tecnológico, el Ray Tracing ha evolucionado, dando lugar al Ray Tracing en tiempo real, una verdadera revolución en la industria de los videojuegos que permite gráficos de alta calidad y efectos visuales fotorrealistas en tiempo real (Cook et al., 1984). Además, ha encontrado aplicaciones en otras áreas, como la simulación de fenómenos físicos, la creación de imágenes médicas detalladas y la arquitectura y diseño de interiores, donde la precisión y el realismo son cruciales para la toma de decisiones (Aila & Laine, 2009).

En un mundo donde las aplicaciones web evolucionan hacia la entrega de contenido más inmersivo y rico en gráficos, el Ray Tracing emerge como una tecnología atractiva para mejorar la calidad visual y la experiencia del usuario en aplicaciones web del lado del cliente. La integración de Ray Tracing con tecnologías web modernas, como WebAssembly, abre la puerta a nuevas posibilidades creativas y funcionales para desarrolladores y diseñadores web, permitiéndoles ofrecer experiencias visuales sorprendentes directamente en el navegador.

En el contexto de este proyecto de investigación, se explora el potencial de WebAssembly para optimizar el proceso de renderizado de imágenes en aplicaciones web del lado del cliente. Se analizará las implementaciones en WebAssembly y JavaScript, con el objetivo de determinar el rendimiento, la eficiencia y la viabilidad de la utilización de WebAssembly en aplicaciones muy exigentes en cómputo.

3.2. Desarrollo web y la complejidad en las aplicaciones del lado del cliente

En la última década, el desarrollo de aplicaciones web del lado del cliente ha experimentado una evolución significativa impulsada por la creciente demanda de experiencias de usuario más ricas y dinámicas. Este avance se ha visto facilitado por la aparición de tecnologías clave como HTML5, CSS3 y JavaScript, que han permitido a los desarrolladores crear aplicaciones web cada vez más complejas y con una mayor interacción.

La evolución del desarrollo web ha llevado a la incorporación de numerosas tecnologías y herramientas que simplifican la creación de aplicaciones web sofisticadas. Entre ellas, se destacan los frameworks y bibliotecas, las cuáles ofrecen una arquitectura de desarrollo robusta y modularizada para la construcción de interfaces de usuario dinámicas y altamente interactivas.

El papel dominante de JavaScript en el desarrollo web se debe a su capacidad para ejecutarse directamente en el navegador del cliente, lo que lo ha convertido en el lenguaje principal para la interacción de las aplicaciones web («Top lenguajes de programación 2023», 2023). Sin embargo, a medida que las aplicaciones web se vuelven más complejas y la demanda de rendimiento aumenta, JavaScript puede enfrentar limitaciones en términos de velocidad y eficiencia (Osmani, 2018).

La complejidad y los desafíos en las aplicaciones web actuales radican en la necesidad de equilibrar una experiencia de usuario rica y altamente interactiva con un rendimiento óptimo. Aplicaciones avanzadas como los videojuegos o la visualización de datos en tiempo real requieren un procesamiento rápido y eficiente para proporcionar una experiencia fluida y sin interrupciones.

Además, el aumento de la complejidad en las aplicaciones del lado del cliente también ha generado demandas más exigentes en términos de recursos y rendimiento, lo que puede afectar negativamente la experiencia del usuario en dispositivos con capacidades limitadas, como dispositivos móviles y navegadores de baja potencia.

En este contexto, WebAssembly surge como una nueva perspectiva en el desarrollo web, y ha sido ampliamente adoptado por la comunidad hasta convertirse en el más reciente estándar web según el World Wide Web Consortium (W3C, 2019). WebAssembly permite la ejecución de código de alto rendimiento directamente en navegadores web, lo que abre nuevas posibilidades en términos de rendimiento y eficiencia para aplicaciones web del lado del cliente. En la siguiente sección, se introducirá a WebAssembly, sus características y ventajas, su funcionamiento en los navegadores, y se explorarán ejemplos de proyectos web reales que han aprovechado esta tecnología para lograr un rendimiento excepcional.

3.3. WebAssembly: Una nueva perspectiva en el desarrollo web

En este apartado, se presenta a WebAssembly, una tecnología innovadora que ha revolucionado el desarrollo web de alto rendimiento en el lado del cliente al permitir la ejecución de código de otros lenguajes de programación en navegadores web. WebAssembly es una solución que aborda los desafíos de rendimiento y eficiencia que pueden surgir al desarrollar aplicaciones web avanzadas y complejas.

3.3.1. Introducción a WebAssembly y su propósito.

WebAssembly, también conocido como Wasm, es un formato binario y lenguaje de bajo nivel que ofrece una alternativa a JavaScript para la ejecución de código en navegadores web. Su objetivo principal es mejorar el rendimiento de las aplicaciones web al permitir que el código se compile de manera eficiente y se ejecute más rápido que el código JavaScript tradicional. WebAssembly no tiene como objetivo principal el reemplazar a JavaScript, sino de actuar como un complemento para aplicaciones intensivas en cálculos y procesamiento (Zakai et al., 2017).

3.3.2. Características y ventajas de WebAssembly.

Una de las características sobresalientes de WebAssembly es su portabilidad, ya que puede ejecutarse en cualquier navegador moderno sin necesidad de realizar cambios significativos en el código fuente. Además, WebAssembly es independiente de la arquitectura del sistema y ofrece un rendimiento consistente en diversas plataformas, lo que lo convierte en una opción atractiva para desarrolladores que desean crear aplicaciones web de alto rendimiento y multiplataforma (WebAssembly, 2023).

WebAssembly se destaca por su similitud con el lenguaje ensamblador (de ahí su nombre). Gracias a esta naturaleza de bajo nivel y su ejecución altamente eficiente, permite velocidades de ejecución cercanas al rendimiento nativo con una representación compacta del tamaño del código, lo que resulta en una experiencia de usuario más fluida y receptiva.

Una de las características más notables de WebAssembly es su capacidad para funcionar en un entorno de sandbox[3], lo que protege el navegador contra código malicioso o inseguro. Al ejecutarse en un espacio aislado, las aplicaciones WebAssembly no tienen acceso directo a recursos peligrosos, garantizando así una mayor seguridad en el navegador y la integridad de los datos del usuario.

Además, WebAssembly es compatible con múltiples lenguajes de programación, lo que permite a los desarrolladores utilizar herramientas y bibliotecas existentes en lenguajes como C/C++, Rust, entre otros, para implementar funcionalidades específicas integradas para Wasm (WebAssembly, 2023). Esto fomenta la reutilización de código y facilita el mantenimiento y la escalabilidad de las aplicaciones web complejas a medida que evolucionan con el tiempo.

La creciente popularidad de WebAssembly ha llevado al desarrollo de frameworks y herramientas que facilitan la implementación y el uso de WebAssembly en aplicaciones web complejas. Frameworks como Blazor[4] y Yew[5] permiten la creación de aplicaciones web interactivas utilizando puramente C# y Rust, eliminando la necesidad de utilizar JavaScript. Además, proyectos como Figma y AutoCAD web, herramientas de diseño y dibujo técnico en la web, aprovechan WebAs-

---

[3] WebAssembly Security: https://webassembly.org/docs/security/

[4] Enlace a Blazor: https://dotnet.microsoft.com/es-es/apps/aspnet/web-apps/blazor

[5] Enlace a Yew: https://yew.rs/

sembly para proporcionar una experiencia de usuario fluida. Estos ejemplos destacados demuestran cómo WebAssembly se integra en el desarrollo web avanzado.

3.3.3. Implementación y funcionamiento en el navegador.

La implementación de WebAssembly en los navegadores es fundamental para comprender cómo esta tecnología permite la ejecución eficiente de código de alto rendimiento en el entorno web. A continuación, se explicará el proceso.

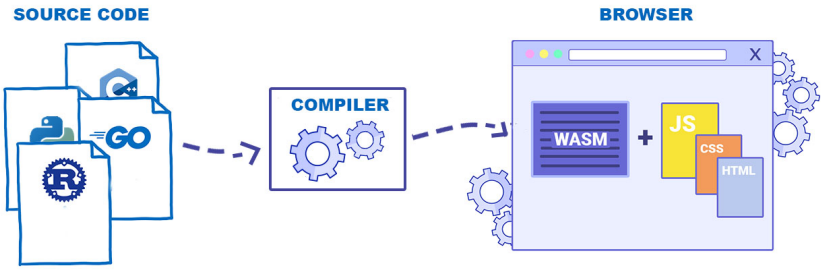


Figura 3. Ilustración WebAssembly

Cuando se carga un módulo WebAssembly en un navegador, este se descarga en forma de código binario en formato wasm. El navegador no puede ejecutar directamente este código binario, ya que funciona en un nivel más bajo que JavaScript. Por lo tanto, se requiere un proceso de traducción y compilación.

1. **Escritura de Código Fuente**: El proceso comienza con la escritura de código fuente en un lenguaje de programación de alto nivel, como C/C++, Rust o entre muchos otros[6].

2. **Compilación y traducción**: Una vez que se ha escrito el código fuente, se utiliza un compilador específico o argumentos propios con los que cuentan algunos compiladores, esto con el fin de convertir en código WebAssembly (Wasm). Aquí es donde entran en juego herramientas como Emscripten[7] o wasm-bindgen[8]. Estas herramientas traducen el código fuente original en lenguaje de alto nivel a un formato binario compacto que representa el código WebAssembly. Este proceso puede implicar la optimización del código para que se ejecute de manera eficiente en la máquina virtual de WebAssembly.

3. **Tipos de datos y comunicación**: WebAssembly utiliza un sistema de tipos más simple en comparación con JavaScript. Para permitir la interoperabilidad entre WebAssembly y JavaScript, se necesitan reglas y conversiones precisas. Herramientas como wasm-bindgen generan código JavaScript que actúa como un puente entre las funciones de WebAssembly y el código JavaScript. Esto facilita la transferencia de datos entre los dos entornos de manera eficiente.

---

6 Algunos lenguajes que soporta Wasm: https://github.com/appcypher/awesome-wasm-langs

7 Emscripten Documentation: https://emscripten.org/docs/index.html

8 The “wasm-bindgen“ Guide: https://rustwasm.github.io/wasm-bindgen/

4. **Optimización**: Una vez que se ha generado el código WebAssembly, se aplica una serie de optimizaciones. Esto puede incluir la eliminación de código redundante, la reducción del tamaño del archivo wasm y la mejora del rendimiento general. La optimización es crucial para garantizar que las aplicaciones web ejecuten el código de manera eficiente y rápida.

5. **Compilación (JIT o AOT)**: El código binario Wasm se carga en el navegador. En este punto, el navegador puede utilizar dos enfoques principales para ejecutar el código WebAssembly:

    - **Compilación JIT (Just-in-Time)**: En este enfoque, el navegador toma el código binario Wasm y lo compila en código máquina nativo para la arquitectura específica del sistema en el que se está ejecutando. Esto se realiza en tiempo real, antes de ejecutar el código. La compilación JIT permite un rendimiento cercano al de las aplicaciones nativas y es comúnmente utilizada en navegadores modernos.

    - **Compilación AOT (Ahead-of-Time)**: En algunos casos, en lugar de realizar una compilación JIT, el código binario Wasm se compila de antemano en código máquina nativo para una plataforma específica antes de su ejecución. Esto puede ser útil en situaciones donde se requiere una carga mínima de tiempo de ejecución y una alta eficiencia.

6. **Ejecución**: Finalmente, el código WebAssembly optimizado y compilado se ejecuta en el navegador. Esto permite que las aplicaciones web aprovechen la velocidad y la eficiencia del código, lo que resulta en una experiencia de usuario más rápida y receptiva.

En resumen, WebAssembly permite que los desarrolladores utilicen lenguajes de alto rendimiento en aplicaciones web mediante un proceso que incluye la traducción del código fuente, la optimización, la compilación y la ejecución en el navegador. La comunicación entre WebAssembly y JavaScript, así como el manejo de tipos de datos, se facilita mediante herramientas como wasm-bindgen. Esta implementación y funcionamiento son esenciales para comprender cómo WebAssembly se ha convertido en una tecnología clave en el desarrollo web de alto rendimiento.

Es importante destacar que WebAssembly es una tecnología versátil que admite múltiples enfoques y puede ser utilizada en una variedad de situaciones, desde aplicaciones web hasta aplicaciones de servidor y más allá. Su capacidad para proporcionar un alto rendimiento y su interoperabilidad con JavaScript lo convierten en una herramienta poderosa para el desarrollo web y más allá.

Esta innovadora idea amplía considerablemente las posibilidades para los desarrolladores, ya que si su lenguaje de programación favorito cuenta con un compilador que genere módulos de WebAssembly, pueden utilizarlo directamente en el navegador. Además, WebAssembly permite llevar a cabo tareas que anteriormente eran limitadas o complejas de realizar en el navegador, como el procesamiento paralelo, gracias a características como hilos y SIMD (Single Instruction, Multiple Data).

3.3.4. Casos de uso y aplicaciones de WebAssembly en proyectos web.

A pesar de que WebAssembly es relativamente nuevo como tecnología, ha logrado ganarse un puesto en una amplia variedad de proyectos web. Esta tecnología ha demostrado ser una solución valiosa, mejorando significativamente el rendimiento y la eficiencia de aplicaciones críticas y complejas. Su capacidad para transformar código existente con ciertas modificaciones y llevarlo a los navegadores web ha abierto un abanico de posibilidades en términos de desarrollo web de alto rendimiento.

A continuación, se presentan algunos casos de uso destacados en los que WebAssembly ha desempeñado un papel fundamental:

1. **Juegos en línea de alto rendimiento**: WebAssembly ha revolucionado la industria de los videojuegos en línea al permitir que los juegos se ejecuten con un rendimiento cercano al de las aplicaciones nativas directamente en el navegador. Esto ha llevado a la creación de experiencias de juego más fluidas y realistas que anteriormente solo eran posibles en aplicaciones de escritorio. La plataforma de desarrollo de juegos Unity permite crear juegos que se ejecutan en el navegador gracias a WebAssembly.

2. **Aplicaciones de edición**: Herramientas de edición de imágenes, video y audio se han beneficiado enormemente de WebAssembly. La capacidad de procesamiento rápido y eficiente de WebAssembly ha mejorado la velocidad y la calidad de las aplicaciones de edición en línea, lo que permite a los usuarios realizar tareas de edición de manera más efectiva y rápida. Un

ejemplo es “Photopea“, una aplicación de edición de fotos en línea que utiliza WebAssembly para proporcionar una experiencia de edición similar a Adobe Photoshop.

3. **Procesamiento de imágenes**: En el ámbito de la manipulación de imágenes, WebAssembly se ha utilizado para acelerar operaciones como el procesamiento de filtros, la detección de objetos y la generación de imágenes en tiempo real. Esto ha mejorado la experiencia en aplicaciones de diseño gráfico y visión por computadora. “PixiJS“ es un ejemplo de una biblioteca de renderizado 2D que utiliza WebAssembly para optimizar el procesamiento de imágenes en aplicaciones web.

4. **Visualización de datos en tiempo real**: Las aplicaciones que requieren visualización de datos en tiempo real, como paneles de control interactivos y representaciones gráficas dinámicas, han encontrado en WebAssembly una solución efectiva para garantizar un rendimiento óptimo y una respuesta rápida a las acciones del usuario. “Plotly.js“ es una biblioteca de visualización que utiliza WebAssembly para garantizar un rendimiento óptimo en la creación de gráficos interactivos.

5. **Simulaciones y cálculos científicos**: WebAssembly se ha utilizado en aplicaciones científicas y de simulación que requieren un alto poder de procesamiento, como la simulación de fenómenos físicos complejos, cálculos matemáticos intensivos y modelado científico. “Foldit“ es un ejemplo de una aplicación de resolución de rompecabezas de proteínas que utiliza WebAssembly para realizar cálculos científicos intensivos.

6. **Ejemplos destacados**: Proyectos como Figma[9], eBay[10] y Google Earth[11] son ejemplos notables de cómo WebAssembly ha permitido la migración de aplicaciones desarrolladas en otros lenguajes, como C#, C/C++ y otros, al entorno web. Estas aplicaciones han experimentado mejoras significativas en el rendimiento y la velocidad de ejecución gracias a WebAssembly.

3.4. El tándem de WebAssembly y JavaScript en acción

En esta sección, exploraremos la poderosa alianza entre WebAssembly y JavaScript. WebAssembly tiene una dualidad interesante que no tienen otros lenguajes web: es un lenguaje en sí mismo y también un objetivo de compilación. Esto significa que podemos compilar WebAssembly desde otros lenguajes como C/C++, Rust y muchos otros. Para ilustrar esta sinergia, presentaremos un sencillo escenario que tiene como objetivo principal mostrar la implementación de WebAssembly junto con Rust y C/C++, brindando una perspectiva de su facilidad de uso en aplicaciones web con JavaScript.

Antes de sumergirnos en el ejemplo práctico, es esencial conocer las tecnologías a utilizar, como es el caso de Rust y su idoneidad para trabajar con WebAssembly.

---

[9] WebAssembly and Figma: https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/

[10] WebAssembly at eBay: https://innovation.ebayinc.com/tech/engineering/webassembly-at-ebay-a-real-world-use-case/

[11] How we're bringing Google Earth to the web: https://web.dev/earth-webassembly/

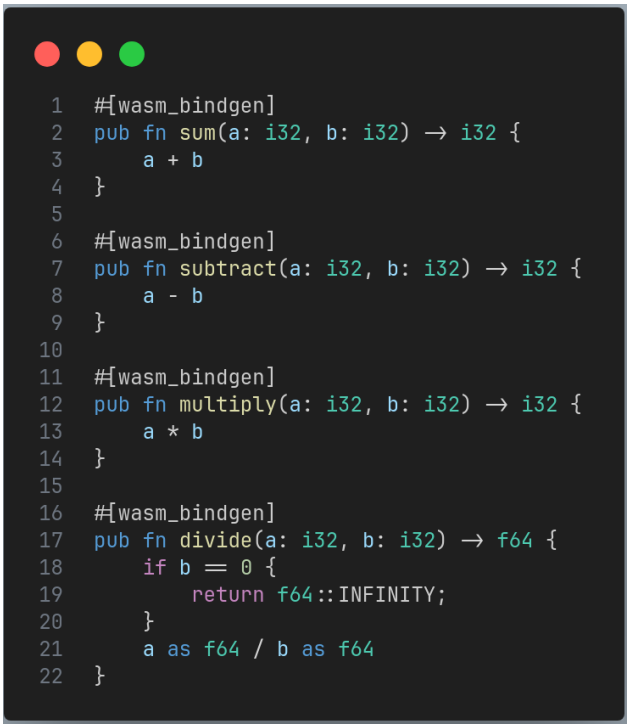
3.4.1. Rust y WebAssembly: Una gran simbiosis para el Desarrollo Web.

Rust es un lenguaje de programación moderno, multiparadigmático, seguro, versátil y de proposito general que ha ganado un lugar destacado en la comunidad de desarrolladores. Su capacidad para garantizar la ausencia de errores de memoria y su alto rendimiento lo convierten en una elección popular (Rodríguez, 2019). Rust ha experimentado un crecimiento constante gracias a su amplia y activa comunidad y respaldado por la fundación Mozilla, llevándolo a ocupar el primer lugar en la encuesta de lenguajes más amados en Stack Overflow durante ocho años consecutivos, desde 2016 (Overflow, 2023).

Una de las razones fundamentales por las cuales Rust es ampliamente elegido para trabajar con WebAssembly es su ecosistema robusto de paquetes conocidos como “crates“ en Rust. Estos paquetes proporcionan soporte para la mayoría de las API web, como el DOM, WebGL y WebAudio, y permiten una total integración con JavaScript. Esto facilita la comunicación bidireccional entre Rust y JavaScript (Rust+Wasm, 2023).

Para el ejemplo que presentaremos a continuación, llevaremos a cabo las cuatro operaciones matemáticas básicas para dos valores. En lugar de implementar estas operaciones directamente en JavaScript, las realizaremos en Rust y, con la ayuda de la crate wasm-bindgen, las convertiremos en un archivo .wasm para su posterior uso en JavaScript.

WebAssembly es ideal para realizar cálculos intensivos debido a su naturaleza, lo que nos lleva a la necesidad de transferir valores entre JavaScript y WebAssembly. Supongamos que las cuatro operaciones matemáticas básicas son funciones que requieren un esfuerzo significativo de cálculo. Nuestra aplicación puede experimentar un aumento significativo en el rendimiento si delega estas tareas en WebAssembly. Tareas como estas y otras se simplifican gracias al ecosistema de Rust. Lo primero que haremos será crear estas funciones “complejas“ en Rust.



```rust
#[wasm_bindgen]
pub fn sum(a: i32, b: i32) -> i32 {
    a + b
}

#[wasm_bindgen]
pub fn subtract(a: i32, b: i32) -> i32 {
    a - b
}

#[wasm_bindgen]
pub fn multiply(a: i32, b: i32) -> i32 {
    a * b
}

#[wasm_bindgen]
pub fn divide(a: i32, b: i32) -> f64 {
    if b == 0 {
        return f64::INFINITY;
    }
    a as f64 / b as f64
}
```

Figura 4. Ejemplo operaciones matemáticas básicas en Rust+Wasm

Como se puede observar en el código anterior, la macro wasm-bindgen es la que ayuda a exponer el código como un módulo exportado de WebAssembly. Esto permite utilizar estas funciones en JavaScript como si fueran funciones convencionales, pero realizando al final su ejecución en WebAssembly.

La siguiente imagen ilustra una pequeña parte del código Rust que fue transformado a WebAssembly. Esto ilustra por qué no es agradable “programar“ directamente en Wasm.

```
(module
    (type (;0;) (func (param i32 i32) (result i32)))
    (type (;1;) (func (param i32 i32)))
    (type (;2;) (func (param i32) (result i32)))
    (type (;3;) (func (param i32 i32 i32) (result i32)))
    (type (;4;) (func (param i32)))
    (type (;5;) (func (param i32) (result i64)))
    (type (;6;) (func (param i32 i32 i32)))
    (type (;7;) (func (result i32)))
    (type (;8;) (func (param i32 i32 i32 i32 i32)))
    (type (;9;) (func (param i32 i32 i32 i32)))
    (type (;10;) (func))
    (type (;11;) (func (param i32 i32 i32 i32) (result i32)))
    (func (;0;) (type 2) (param i32) (result i32)
      (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i64)
      global.get 0
      i32.const 16
      i32.sub
      local.tee 11
      "continua el código"
```

Figura 5. Ejemplo de una parte del código Wasm equivalente al de Rust

La comunicación entre Rust+Wasm y la aplicación JavaScript es extremadamente sencilla debido a las características distintivas de Rust y su amplio ecosistema de WebAssembly.

```
import init, { add, subtract, multiply, divide } from './rust-examples/pkg/rust_examples.js';

async function main() {
await init();
const add_res = add(17,17);
const sub_res = subtract(100,50);
const mul_res = multiply(17,17);
const div_res = divide(100,5);

console.log(`add: ${add_res}`);
console.log(`subtract: ${sub_res}`);
console.log(`multiply: ${mul_res}`);
console.log(`divide: ${div_res}`);
}

main();
```

Figura 6. Ejemplo uso de Rust+Wasm en JavaScript

3.4.2. C/C++ y Emscripten: Ampliando las Posibilidades con WebAssembly.

Otra poderosa combinación que nos permite aprovechar la potencia de WebAssembly en el desarrollo web es la utilización de C/C++ junto con Emscripten. C/C++ es un lenguaje de programación de propósito general que cuenta con un largo historial respaldando sus capacidades, habiendo sido ampliamente utilizado en una variedad de aplicaciones, incluyendo el desarrollo de sistemas operativos y aplicaciones de alto rendimiento. Al combinar C/C++ con Emscripten, podemos compilar código C/C++ a WebAssembly y ejecutarlo en el navegador («Main — Emscripten 3.1.44-git dev documentation», 2015).

Emscripten es una herramienta que permite compilar C/C++ a WebAssembly, lo que facilita la portabilidad del código existente a la web sin tener que reescribirlo en JavaScript. Además, Emscripten proporciona una interfaz que nos permite llamar a funciones escritas en C/C++ desde JavaScript y viceversa, lo que facilita la comunicación bidireccional entre ambas capas.

Continuando con el ejemplo de las operaciones matemáticas básicas, vamos a implementar las mismas funciones en C/C++ utilizando Emscripten para compilar el código a WebAssembly. Luego, utilizaremos JavaScript para llamar estas funciones implementadas en C/C++ y mostrar cómo se pueden usar en una aplicación web.

La siguiente muestra de código ilustra cómo se vería la implementación de las funciones en C utilizando Emscripten:

```c
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE
int add(int a, int b) {
    return a + b;
}

EMSCRIPTEN_KEEPALIVE
int subtract(int a, int b) {
    return a - b;
}

EMSCRIPTEN_KEEPALIVE
int multiply(int a, int b) {
    return a * b;
}

EMSCRIPTEN_KEEPALIVE
int divide(int a, int b) {
    if (b == 0) {
        // Manejo de división por cero
        return 0;
    }
    return a / b;
}
```

Figura 7. Ejemplo operaciones matemáticas básicas en C/C++ Wasm

Una vez que hemos implementado estas funciones en C y las hemos compilado a WebAssembly con la ayuda de Emscripten, podemos importarlas en JavaScript y utilizarlas como funciones convencionales, para esto hay diversas formas. A continuación, se muestra un ejemplo de cómo usar estas funciones en JavaScript:

```
let module,
  functions = {};
fetch("mathModule.wasm")
  .then((response) => response.arrayBuffer())
  .then((buffer) => new Uint8Array(buffer))
  .then((binary) => {
    let moduleArgs = {
      wasmBinary: binary,
      onRuntimeInitialized: function() {
        functions.add = module.cwrap("add", "number", ["number", "number"]);
        functions.subtract = module.cwrap("subtract", "number", [
          "number",
          "number",
        ]);
        functions.multiply = module.cwrap("multiply", "number", [
          "number",
          "number",
        ]);
        functions.divide = module.cwrap("divide", "number", [
          "number",
          "number",
        ]);
        cpp_load = true;
        console.log("Math module loaded");
        onReady();
      },
    };
    module = Module(moduleArgs);
  });

// Luego, puedes usar las funciones importadas de la siguiente manera:
const resultAdd = functions.add(5, 3);
const resultSubtract = functions.subtract(10, 4);
const resultMultiply = functions.multiply(6, 7);
const resultDivide = functions.divide(20, 4);
console.log("Add:", resultAdd);
console.log("Subtract:", resultSubtract);
console.log("Multiply:", resultMultiply);
console.log("Divide:", resultDivide);
```

Figura 8. Ejemplo uso de C Wasm en JavaScript

Como se puede observar en el ejemplo anterior, la comunicación entre JavaScript y C/C++ con Emscripten es sencilla en principio y permite una integración fluida de código C/C++ en aplicaciones web. Con esta poderosa combinación de tecnologías, se puede llevar al navegador aplicaciones complejas y de alto rendimiento que se beneficien del rendimiento y la eficiencia de WebAssembly.

3.5. Más allá del código: Las Métricas y Criterios detrás de la aplicación web

En esta sección, profundizaremos en el análisis de rendimiento y eficiencia de la aplicación web, explorando las métricas y criterios fundamentales que sustentan la evaluación del proyecto. Más allá de la implementación del código, es crucial comprender cómo medir el éxito y la efectividad de la aplicación web, especialmente en el contexto del trabajo de WebAssembly y JavaScript.

3.5.1. Métricas y Variables Críticas para Medir el Rendimiento en Aplicaciones Web.

Dentro del universo de las aplicaciones web, el rendimiento es un aspecto primordial para ofrecer una experiencia de usuario satisfactoria. En esta sección, examinaremos las métricas clave que nos permiten cuantificar y comparar el desempeño de diferentes implementaciones. Analizaremos aspectos como el tiempo de carga, la velocidad de ejecución, el consumo de recursos y otros indicadores esenciales para entender la eficacia del proyecto (Šipek et al., 2021).

Para evaluar el rendimiento de las implementaciones, es fundamental definir métricas y variables que nos permitan medir de manera objetiva la eficiencia de cada una. Algunas de las métricas a considerar posiblemente incluyen:

- **Tiempo de carga**: El tiempo que tarda la aplicación web en cargar y estar lista para su interacción.
- **Tiempo de ejecución**: El tiempo requerido para llevar a cabo cada implementación.

- **Velocidad de renderizado**: Cuántos fotogramas por segundo puede generar la implementación en tiempo real.
- **Precisión del resultado**: Comparar la calidad visual de los resultados obtenidos por cada implementación.
- **Escalabilidad**: Evaluar cómo cada implementación maneja escenas más complejas.
- **Fiabilidad**: Determinar la estabilidad de cada implementación en diferentes escenarios.
- **Usabilidad**: Evaluar la facilidad de uso de cada implementación.
- **Mantenibilidad**: Evaluar la facilidad de mantenimiento de cada implementación.

3.5.2. Métodos y Herramientas Utilizadas para la Evaluación.

Una evaluación precisa y rigurosa requiere herramientas adecuadas y enfoques metodológicos consistentes. En esta sección, expondremos las herramientas y técnicas que utilizaremos para recolectar datos precisos, realizar pruebas y generar resultados significativos.

Para llevar a cabo la evaluación del rendimiento, utilizaremos diferentes técnicas y herramientas, garantizando la precisión y objetividad en los resultados. Algunas de las herramientas y enfoques que emplearemos posiblemente son:

- **Benchmarks**: Implementación de pruebas de rendimiento específicas que nos permitirán comparar el desempeño de las diferentes tecnologías en situaciones controladas y reproducibles.

- **Herramientas de Desarrollo de Navegadores**: Utilizaremos las herramientas de desarrollo proporcionadas por los navegadores modernos para analizar el rendimiento y detectar posibles inconvenientes.

- **Perfiles de Rendimiento**: Se crearán perfiles de rendimiento utilizando herramientas como Edge DevTools, Firefox Developer Tools, entre otras, para identificar áreas de mejora en el código.

- **Herramientas de Monitoreo de Recursos**: Se utilizarán herramientas desde línea de comandos hasta GUI como son posiblemente: top, Powertop, Perfmon, administradores de tareas y monitoreos de actividad ofrecidos por los mismos sistemas operativos.

3.5.3. Importancia de la Eficiencia en el contexto de las implementaciones.

El proyecto se enfoca en demostrar el potencial de WebAssembly como tecnología para su implementación en aplicaciones web avanzadas. En este contexto, la eficiencia juega un papel crítico para asegurar un rendimiento óptimo y una experiencia visual de alta calidad en el caso de estudio. Se analizará cómo se mide la eficiencia en el contexto específico del Ray Tracing y cómo las métricas nos permiten evaluar la eficacia de lo implementado.

En el contexto del Ray Tracing, la eficiencia es un factor crítico para garantizar la generación rápida y precisa de imágenes fotorrealistas. La implementación de algoritmos intensivos en cálculos matemáticos, como los utilizados en el Ray Tracing, requiere una gestión cuidadosa de los recursos y tiempos de ejecución para lograr una experiencia fluida y atractiva para el usuario.

Al evaluar la eficiencia de las implementaciones tecnológicas, se podrá identificar cuál de las tecnologías es más adecuada para enfrentar los diversos desafíos que afronta la web de alto rendimiento. El análisis de las métricas y criterios recopilados permitirá comprender cómo cada tecnología se adapta a las demandas específicas de cálculos complejos y manipulación de datos, y cómo estas características influyen en el rendimiento general de la aplicación. Una implementación eficiente permitirá una mayor inmersión en la experiencia de usuario, especialmente en aplicaciones web que buscan ofrecer una experiencia gráfica de alta calidad, como los videojuegos y simulaciones interactivas.

4. Metodología

La metodología de este proyecto de investigación es el motor que impulsa el proceso de evaluación y comparación de WebAssembly y JavaScript en el desarrollo web avanzado. En un entorno donde la eficiencia y el rendimiento son críticos, esta metodología proporciona una guía para llevar a cabo un estudio exhaustivo y riguroso. A través de un modelo de prototipado iterativo conformado por 4 fases fundamentales, cada una destinada a cumplir un conjunto específico de objetivos. Esta metodología busca explorar, analizar y medir el potencial de WebAssembly en comparación con JavaScript en una variedad de contextos.

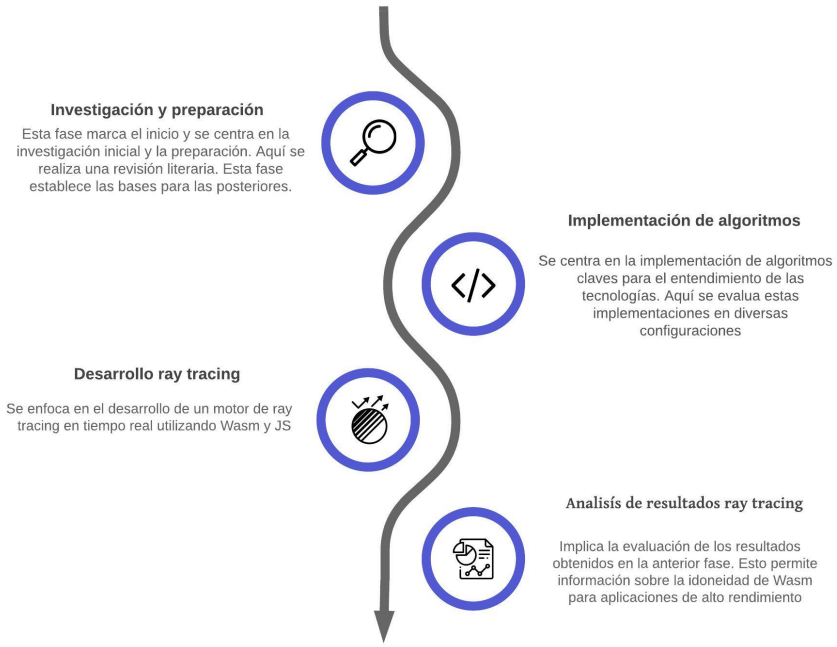


Figura 9. Diagrama metodología

Investigación y Preparación

En esta fase inicial, se llevará a cabo un estudio preliminar exhaustivo para comprender en profundidad las tecnologías involucradas en el proyecto, con un enfoque particular en WebAssembly. Se explorarán diversas fuentes, investigaciones y documentación relacionada para establecer una base sólida de conocimientos antes de abordar las implementaciones.

Objetivos.

1. Investigar y comprender a WebAssembly y su funcionamiento en el contexto del desarrollo web.

2. Explorar las tecnologías relacionadas con WebAssembly y su idoneidad para el proyecto.

Actividades.

1. **Revisión Bibliográfica**: Se llevó a cabo una revisión exhaustiva de la literatura y documentación disponible sobre WebAssembly y su aplicación en el desarrollo web. Esto incluyó la exploración de recursos académicos, tutoriales en línea y documentación oficial.

2. **Análisis de Tecnologías Complementarias**: Se investigaron tecnologías complementarias que podrían utilizarse junto con WebAssembly, como los lenguajes de programación C y Rust. Se evaluó su capacidad para garantizar el alto rendimiento y la eficiencia requeridos en el proyecto.

3. **Selección de Tecnologías y Plataformas**: Se eligieron cuidadosamente las plataformas de prueba para la evaluación de rendimiento, que incluyeron sistemas operativos (Linux, Windows, macOS) y dispositivos. También se seleccionaron los navegadores web para las pruebas (Google Chrome, Firefox, Microsoft Edge y Safari).

4. **Definición de Métricas Preliminares**: Aunque las métricas de rendimiento se medirán en fases posteriores, en esta etapa inicial se espera definir las métricas preliminares que se espera evaluar durante el proyecto. Esto puede incluir métricas como el tiempo de carga y el tiempo de ejecución.

Implementación de Algoritmos y Análisis de Rendimiento

En esta fase, se llevaron a cabo implementaciones de diversos algoritmos utilizando tres tecnologías distintas: JavaScript, C y Rust. El propósito de esta fase fue evaluar el rendimiento de cada tecnología al implementar algoritmos específicos y analizar cómo se comportaron en diferentes configuraciones y proporcionar una visión de la eficiencia relativa de estas tecnologías en una variedad de situaciones.

Descripción General de la Fase.

Se dio vida a los conceptos teóricos mediante la traducción de algoritmos clave a lenguajes de alto nivel que se compilaron a WebAssembly. Esta etapa implicó la programación detallada de algoritmos específicos, como detección de colisiones, cálculos matemáticos y algoritmos de ordenación, en lenguajes compatibles con WebAssembly como lo son C y Rust. Se abordó la com-

plejidad de garantizar que estos algoritmos funcionaran de manera eficiente en la plataforma de WebAssembly y se llevaron a cabo pruebas para evaluar su rendimiento en diversas configuraciones.

Algoritmos Implementados.

1. **Collision Detection**: Este algoritmo tiene como función detectar colisiones entre objetos en un espacio tridimensional. Es fundamental en aplicaciones de simulación y juegos, donde se utiliza para garantizar que los objetos virtuales interactúen de manera realista, evitando superposiciones no deseadas y garantizando una experiencia interactiva coherente.

2. **Fibonacci**: Calcula el enésimo número de la secuencia de Fibonacci, donde cada número es la suma de los dos números anteriores. Aunque aparentemente simple, este algoritmo demuestra su utilidad en diversas aplicaciones, como modelado de crecimiento, teoría de números y optimización.

3. **Multiply Double**: Realiza la multiplicación repetida de dos números en coma flotante, brindando una base sólida para operaciones matemáticas esenciales en aplicaciones científicas y de ingeniería. La precisión y eficiencia de la multiplicación en coma flotante son cruciales en una variedad de contextos.

4. **Multiply Int Vector**: Multiplica elementos de un vector de números enteros. Este algoritmo ofrece una herramienta valiosa en aplicaciones que involucran procesamiento de datos y álgebra lineal. Puede utilizarse para entender la base de operaciones en grandes conjuntos de datos.

5. **Multiply Double Vector**: Similar al algoritmo anterior, este algoritmo multiplica elementos de dos vectores de números en coma flotante. Es esencial en tareas de procesamiento de señales, gráficos por computadora y simulaciones físicas que involucran vectores de valores.

6. **QuickSort Int**: Es una técnica de ordenamiento altamente eficiente que se utiliza para ordenar arreglos de números enteros. Su capacidad para realizar particiones y ordenar elementos lo convierte en una elección común en aplicaciones que requieren una clasificación rápida y eficaz de datos.

7. **QuickSort Double**: Similar al algoritmo anterior, el Quicksort se aplica aquí para ordenar un arreglo de números en coma flotante. Esta implementación es crucial en aplicaciones que requieren una clasificación rápida y precisa de datos con decimales.

8. **Sum Int**: Calcula la suma de elementos en un arreglo de números enteros. Este proceso básico tiene aplicaciones en estadísticas, análisis de datos y cálculos matemáticos esenciales.

9. **Sum Double**: Al igual que el algoritmo anterior, se encarga de calcular la suma de elementos en un arreglo, pero en este caso, de números en coma flotante.

10. **Image Convolute**: Este algoritmo permite realizar una convolución de imagen utilizando un kernel de peso específico. La convolución de imágenes es esencial en procesamiento de imágenes y visión por computadora para aplicar efectos, realzar características y filtrar imágenes de manera personalizada.

11. **Video Convolute**: La aplicación de convolución a un video en tiempo real. Es una tarea avanzada que involucra la manipulación de múltiples cuadros por segundo. Esta idea básica se emplea en aplicaciones de procesamiento de video en tiempo real, como efectos visuales y análisis de video.

Desarrollo del Caso de Estudio - Ray Tracing

En esta fase, se explora la implementación del caso de estudio principal, que implica la creación de un algoritmo de Ray Tracing en un entorno web utilizando WebAssembly y JavaScript. El Ray Tracing es un enfoque poderoso para generar imágenes tridimensionales realistas al simular el comportamiento de la luz al rebotar en superficies y acumular colores a lo largo de su trayecto. Aunque la técnica en sí es muy regular y no es intrínsecamente rápida, su simplicidad conceptual permite modelar efectos visuales impresionantes, como reflejos, sombras y profundidad de campo.

Objetivos.

1. Implementar un algoritmo de Ray Tracing en Rust y JavaScript.
2. Evaluar el rendimiento y la eficiencia de la implementación del Ray Tracing en ambos lenguajes y su integración con WebAssembly.
3. Comparar la calidad visual de las imágenes generadas por ambas implementaciones.

Actividades.

1. Desarrollo de la Implementación del Ray Tracing:
   - Implementar el algoritmo de Ray Tracing en Rust y JavaScript.
   - Utilizar WebAssembly para compilar y ejecutar las implementaciones en los navegadores web compatibles.
   - Generar imágenes fotorrealistas utilizando ambas implementaciones.
2. Evaluación de Rendimiento:
   - Realizar pruebas de rendimiento utilizando escenas y modelos 3D complejos.
   - Medir el tiempo de renderizado y comparar el rendimiento en diferentes configuraciones (navegador, sistema operativo).

Fase de Análisis de Resultados del Ray Tracing

La fase final del proyecto implicó una evaluación detallada de los resultados obtenidos en la implementación del ray tracing. Se recolectaron datos sobre los tiempos de renderizado, las velocidades de fotogramas por segundo (FPS) y otros indicadores clave en diversas configuraciones de navegadores y sistemas operativos. Este análisis permitió obtener información valiosa sobre la idoneidad de WebAssembly para aplicaciones web de alto rendimiento, con un enfoque particular en el ray tracing. Los resultados fueron esenciales para extraer conclusiones y recomendaciones sobre el uso de WebAssembly en aplicaciones de renderizado en tiempo real.

5. Desarrollo

En esta sección, se profundiza en el proceso de desarrollo que sigue cada una de las fases de la metodología previamente presentada.

Las fases de implementación de algoritmos y del ray tracing en tiempo real representan el núcleo de la aplicación de la tecnología WebAssembly en el contexto de este proyecto. Estas etapas involucraron la traducción de conceptos teóricos en código práctico y eficiente, junto con la resolución de desafíos técnicos específicos relacionados con el rendimiento y la optimización. Los detalles de la implementación y las estrategias utilizadas se abordarán en sus respectivas secciones.

La fase de análisis de resultados del ray tracing es crucial para evaluar los resultados de rendimiento y establecer conclusiones significativas. Aquí se presentarán los datos recopilados durante las pruebas y se discutirán los hallazgos clave. La interpretación de los resultados es esencial para comprender la idoneidad de WebAssembly en aplicaciones de alto rendimiento como el ray tracing en tiempo real.

A medida que avanzamos en esta sección, exploraremos en detalle cada fase del desarrollo, examinando los desafíos y resultados obtenidos. Este enfoque proporcionará una visión completa del proceso de implementación y evaluación de WebAssembly en aplicaciones web de alto rendimiento, con un enfoque especial en el ray tracing en tiempo real.

5.1. Investigación y Preparación

Los resultados preliminares de esta fase proporcionaron información valiosa sobre las tecnologías y plataformas disponibles, lo que ayudará en las próximas etapas del proyecto. Los lenguajes de programación C y Rust se consideran candidatos para las implementaciones de algoritmos en WebAssembly, y JavaScript se utilizará como punto de referencia.

5.2. Implementación de Algoritmos y Análisis de Rendimiento

Con el fin de evaluar las capacidades de rendimiento de las tecnologías en una variedad de escenarios. Cada algoritmo se implementó en JavaScript, C y Rust, lo que permitió realizar comparaciones directas.

Escenarios de Pruebas.

Los algoritmos se ejecutaron en diferentes sistemas operativos, incluyendo Linux, Windows y macOS, y se evaluaron en cuatro navegadores web populares: Google Chrome, Firefox, Microsoft Edge y Safari (limitado a ser evaluado en macOS). El propósito de esta variedad de configuraciones no fue buscar una comparación exhaustiva de rendimiento e intentar concluir con una configuración “vencedora“, sino evaluar la capacidad de WebAssembly para funcionar de manera eficiente en una variedad de entornos. La decisión de probar solo Safari en macOS se basó en la consideración de que otros navegadores tienen un porcentaje de uso significativamente menor en este sistema operativo, lo que sugiere que Safari es el navegador predominante en macOS.

Obtención y Presentación de Resultados.

La obtención y presentación de los resultados se realizó de manera rigurosa para garantizar la precisión y la transparencia de las mediciones de rendimiento. Cada algoritmo implementado fue sometido a múltiples experimentos en diferentes configuraciones de sistemas operativos y navegadores web. A continuación, se describe el proceso para obtener y presentar los resultados:

Recopilación de Datos.

Experimentos Controlados: Se ejecutaron experimentos controlados para cada algoritmo y tecnología en distintos escenarios de sistemas operativos y navegadores. Cada algoritmo fue evaluado a través de 100 experimentos donde cada uno presenta un tiempo promedio de su ejecución y finalización.

Registro de Tiempos: En cada experimento, se registraron los tiempos de ejecución precisos en milisegundos para cada iteración del algoritmo.

Cálculo de Varianza: Se calculó la varianza de los tiempos de ejecución para evaluar la consistencia del rendimiento en cada configuración.

Promedios y Velocidades: Se calcularon tiempos promedio de ejecución para cada algoritmo en cada tecnología y se compararon con JavaScript para determinar las velocidades relativas.

Presentación de Resultados.

En esta sección, se presentan los resultados de los exhaustivos experimentos de rendimiento en los que evaluamos la ejecución de varios algoritmos en tres tecnologías diferentes: JavaScript, C y Rust. Los experimentos se llevaron a cabo en diversas configuraciones para evaluar el rendimiento en un amplio espectro de entornos.

En primer lugar, se mostrarán gráficos con los tiempos de ejecución de cada algoritmo, lo que proporciona una visión del rendimiento en diferentes contextos. Esto permitirá identificar tendencias y variaciones en los tiempos de ejecución para cada implementación.

Luego, se presentarán los datos de promedios de ciertas medidas, que ayudará a comprender cómo se comparan las implementaciones en términos de eficiencia relativa. Esto revelará cuán rápido o lento es el rendimiento de C y Rust en comparación con JavaScript.

Además, todos los datos recopilados durante esta fase, incluyendo los tiempos individuales de cada experimento, las varianzas y los promedios, están disponibles en un repositorio público en GitHub. Esto permite ser revisados los datos en detalle si lo desean.

1. **Collision Detection(CD)**: Cada experimento consta de 50 ciclos.
2. **Fibonacci(F)**: Cada experimento consta de 50 ciclos. Se evalua la obtención de fibonacci para el valor 40.

3. **Multiply Double(MD)**: Cada experimento consta de 50 ciclos, se procede a realizar una multiplicación de 1.0 por 1.0 en una cantidad de 100000000 ciclos.

```
function multiplyDouble() {
    let c = 1.0;
    for (let i = 0; i < 100000000; i++) {
    c = c * 1.0 * 1.0;
    }
    return c;
}
```

4. **Multiply Int Vector(MIV)**: Cada experimento consta de 50 ciclos, se procede a realizar la multiplicación entre 10000000 valores enteros aleatorios dentro de un arreglo.

5. **Multiply Double Vector(MDV)**: Cada experimento consta de 50 ciclos, se procede a realizar la multiplicación entre 10000000 valores de coma flotante aleatorios dentro de un arreglo.

6. **QuickSort Int(QI)**: Cada experimento consta de 50 ciclos, se procede a ordenar un arreglo de 5000000 valores enteros aleatorios.

7. **QuickSort Double(QD)**: Cada experimento consta de 50 ciclos, se procede a ordenar un arreglo de 5000000 valores coma flotante aleatorios.

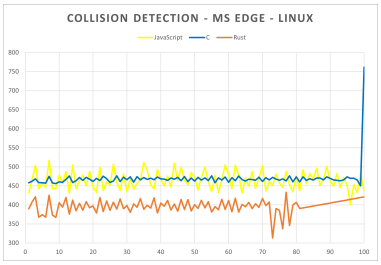
8. **Sum Int(SI)**: Cada experimento consta de 50 ciclos, se procede a sumar 100000000 valores enteros aleatorios que se encuentran dentro de un arreglo.

9. **Sum Double(SD)**: Cada experimento consta de 50 ciclos, se procede a sumar 60000000 valores coma flotante aleatorios que se encuentran dentro de un arreglo.

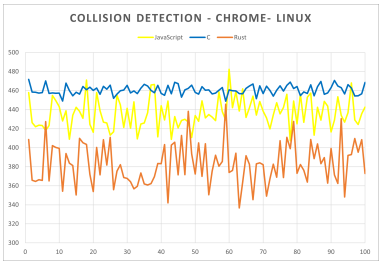
10. **Image Convolute(IC)**: Cada experimento consta de 1 ciclo donde se le realiza una convolución a una imagen utilizando un kernel de peso específico.

11. **Video Convolute(VC)**: Aplica convolución a un video en tiempo real utilizando la biblioteca Three.js. Cada experimento consta de 1 ciclo donde se le realiza una convoluciones en tiempo real a un video.

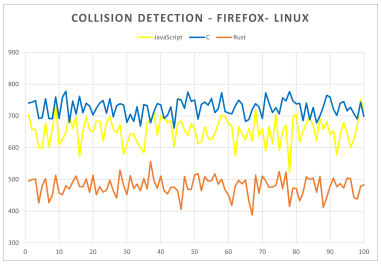
Gráficos de tiempos: Se generaron de los tiempos de ejecución de cada algoritmo en diferentes configuraciones, lo que proporciona una representación visual clara del rendimiento en distintos contextos.



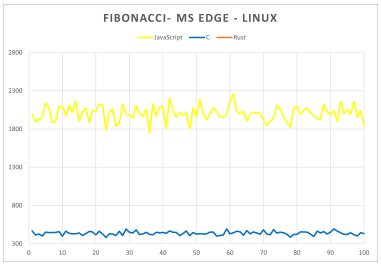
(a) CD-Edge-Linux



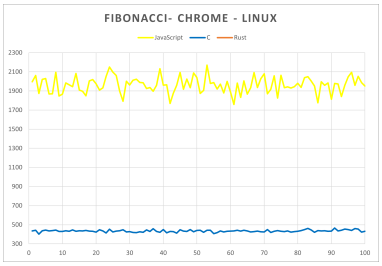
(b) CD-Chrome-Linux



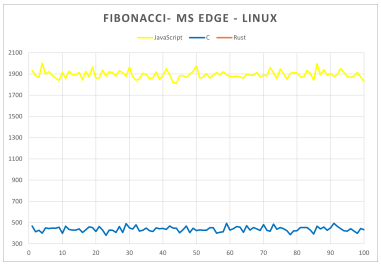
(c) CD-Firefox-Linux



(d) F-Edge-Linux

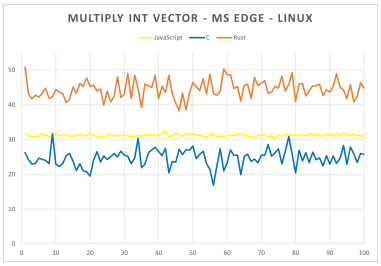


(e) F-Chrome-Linux

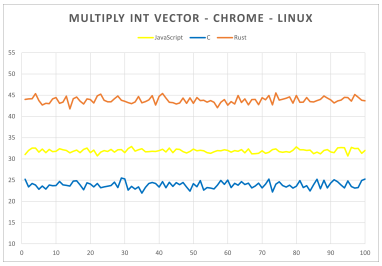


(f) F-Firefox-Linux

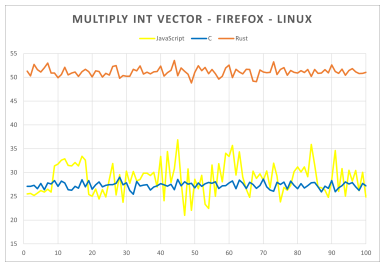
Figura 10. Gráficos de tiempos algoritmos Fase 1 - Parte 1



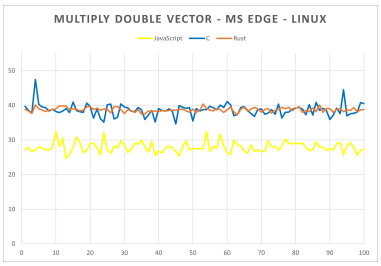
(a) MIV-Edge-Linux



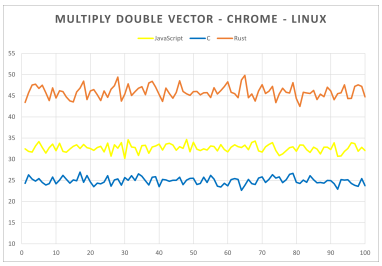
(b) MIV-Chrome-Linux



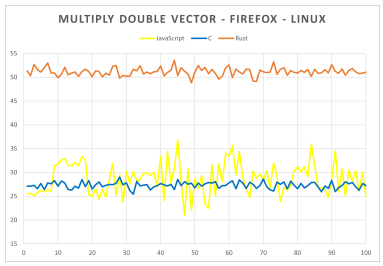
(c) MIV-Firefox-Linux



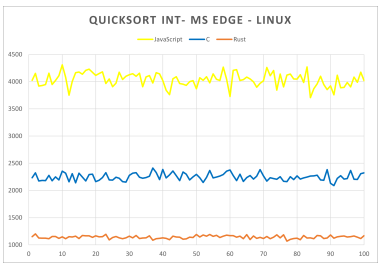
(d) MDV-Edge-Linux



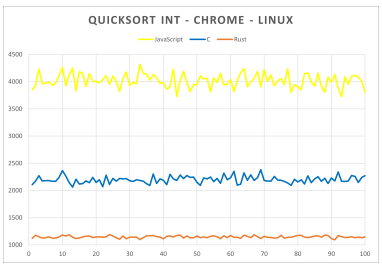
(e) MDV-Chrome-Linux



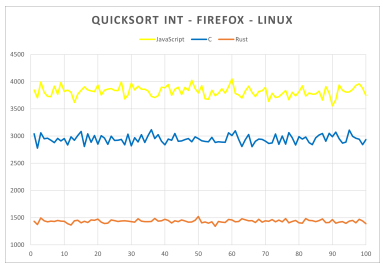
(f) MDV-Firefox-Linux



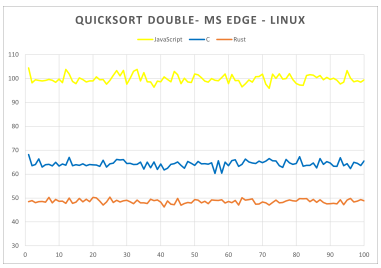
(g) QI-Edge-Linux



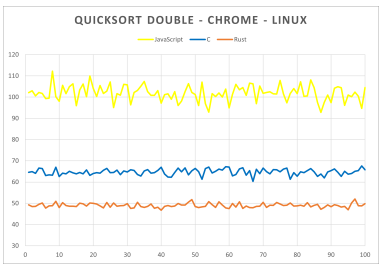
(h) QI-Chrome-Linux



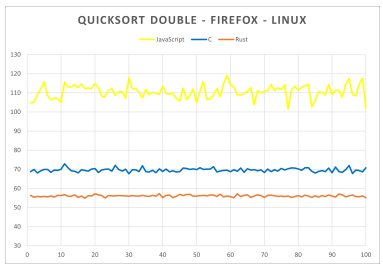
(i) QI-Firefox-Linux



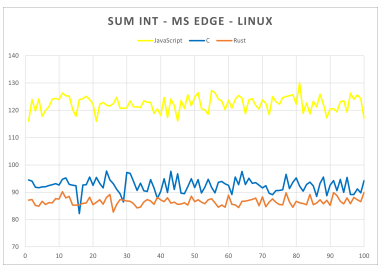
(j) QD-Edge-Linux



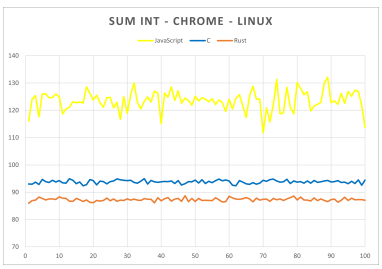
(k) QD-Chrome-Linux



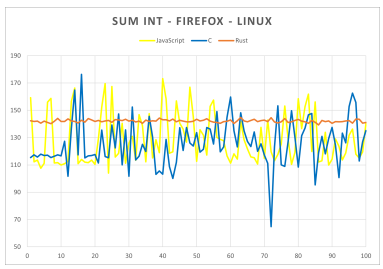
(l) QD-Firefox-Linux



(m) SI-Edge-Linux

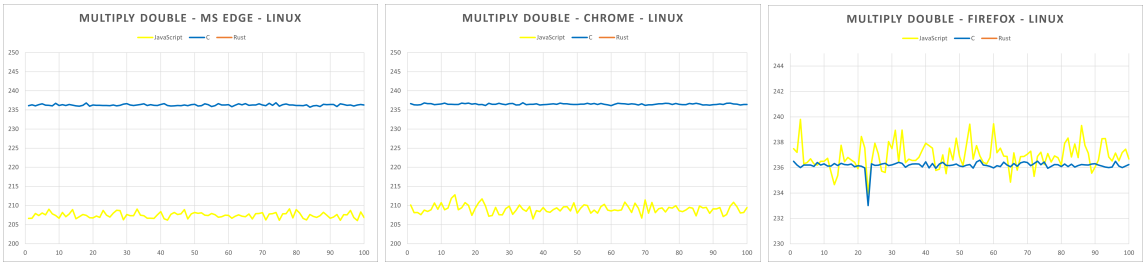


(n) SI-Chrome-Linux

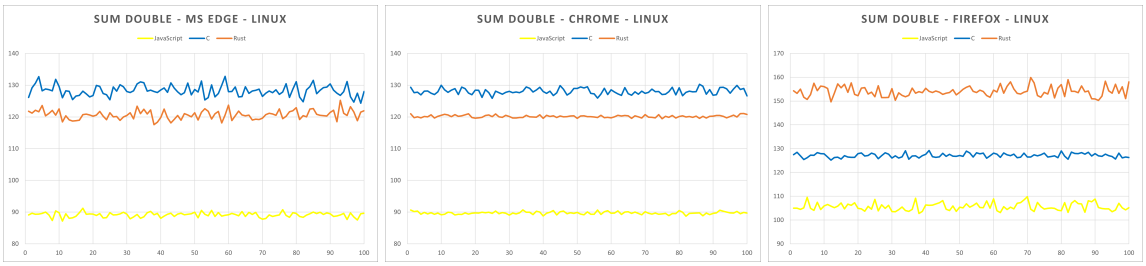


(ñ) SI-Firefox-Linux

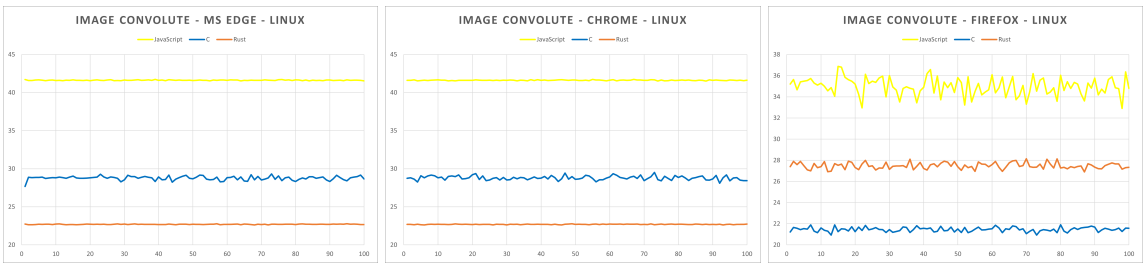
Figura 11. Gráficos de tiempos algoritmos Fase 1 - Parte 2



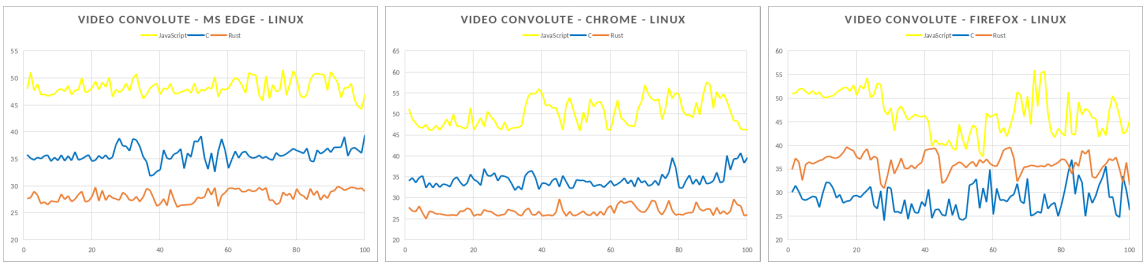
(a) MD-Edge-Linux (b) MD-Chrome-Linux (c) MD-Firefox-Linux



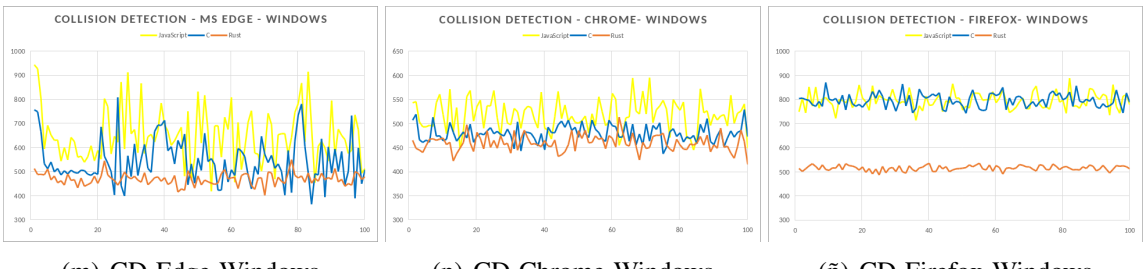
(d) SD-Edge-Linux (e) SD-Chrome-Linux (f) SD-Firefox-Linux



(g) IC-Edge-Linux (h) IC-Chrome-Linux (i) IC-Firefox-Linux



(j) VC-Edge-Linux (k) VC-Chrome-Linux (l) VC-Firefox-Linux



(m) CD-Edge-Windows (n) CD-Chrome-Windows (ñ) CD-Firefox-Windows

Figura 12. Gráficos de tiempos algoritmos Fase 1 - Parte 3

(a) F-Edge-WIndows (b) F-Chrome-Windows (c) F-Firefox-Windows

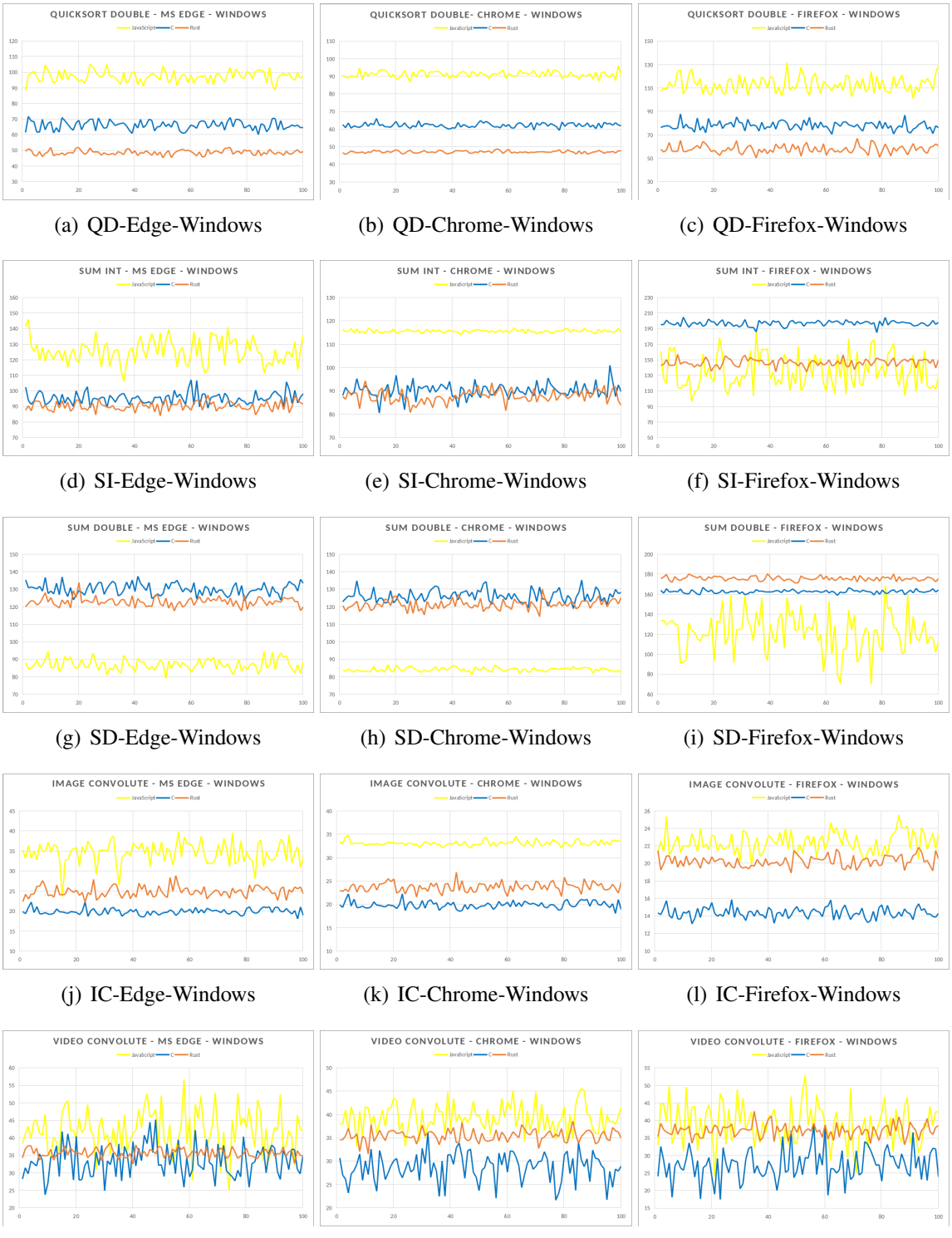
(d) MD-Edge-Windows (e) MD-Chrome-Windows (f) MD-Firefox-Windows

(g) MIV-Edge-Windows (h) MIV-Chrome-Windows (i) MIV-Firefox-Windows

(j) MDV-Edge-Windows (k) MDV-Chrome-Windows (l) MDV-Firefox-Windows

(m) QI-Edge-Windows (n) QI-Chrome-Windows (ñ) QI-Firefox-Windows

Figura 13. Gráficos de tiempos algoritmos Fase 1 - Parte 4



(a) QD-Edge-Windows (b) QD-Chrome-Windows (c) QD-Firefox-Windows

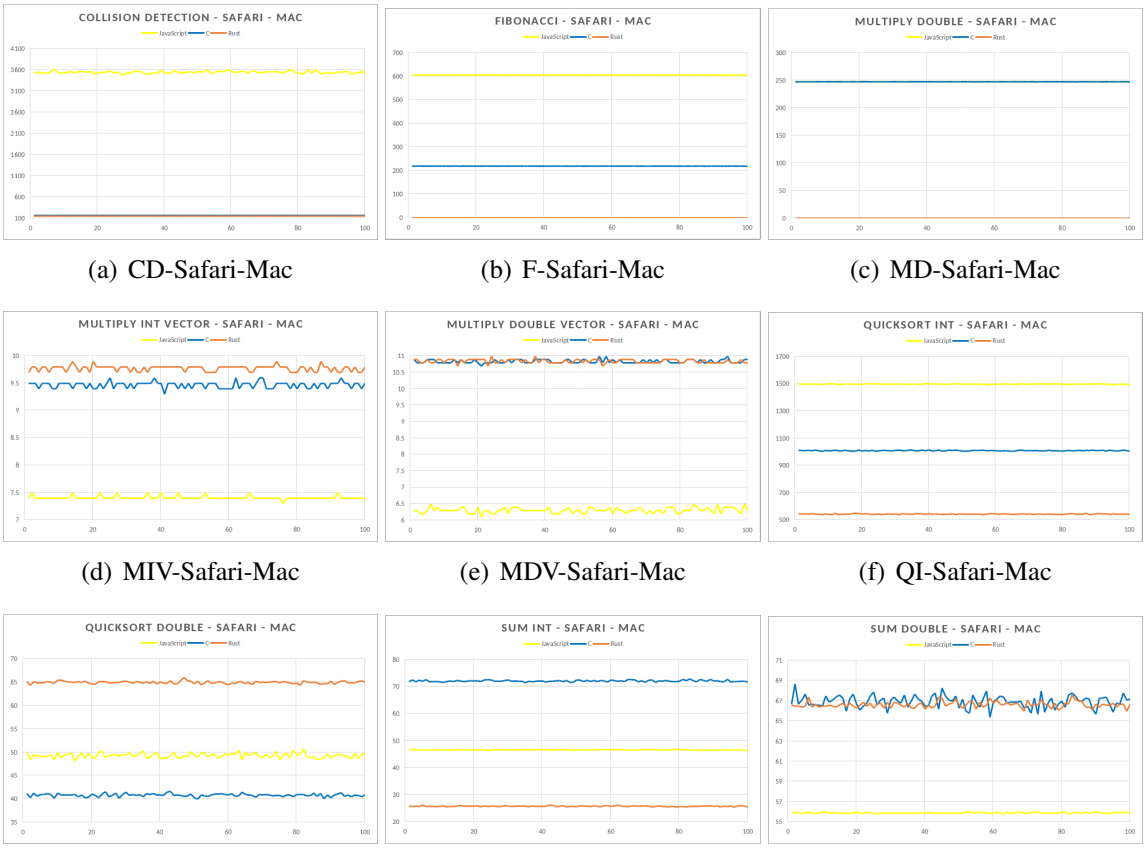
(d) SI-Edge-Windows (e) SI-Chrome-Windows (f) SI-Firefox-Windows

(g) SD-Edge-Windows (h) SD-Chrome-Windows (i) SD-Firefox-Windows

(j) IC-Edge-Windows (k) IC-Chrome-Windows (l) IC-Firefox-Windows

(m) VC-Edge-Windows (n) VC-Chrome-Windows (ñ) VC-Firefox-Windows

Figura 14. Gráficos de tiempos algoritmos Fase 1 - Parte 5



(a) CD-Safari-Mac (b) F-Safari-Mac (c) MD-Safari-Mac

(d) MIV-Safari-Mac (e) MDV-Safari-Mac (f) QI-Safari-Mac

(g) QD-Safari-Mac (h) SI-Safari-Mac (i) SD-Safari-Mac

Figura 15. Gráficos de tiempos algoritmos Fase 1 - parte 6

Repositorio de Datos.

Todos los datos recopilados durante esta fase, así como las gráficas y tablas de resultados, se encuentran disponibles en el siguiente repositorio público en GitHub[12]. Este repositorio proporciona acceso a la información completa y detallada para aquellos que deseen examinar los datos en profundidad.

---

12 Enlace al repositorio de GitHub: https://github.com/RockyCott/benchmark-wasm-vs-js

La presentación de resultados se centra en promedios para resaltar patrones de rendimientos. Sin embargo, se recomienda consultar el repositorio en GitHub para acceder a los datos completos y realizar análisis personalizados si es necesario por medio de un despliegue realizado del sitio en GitHub Pages[13].

Análisis de Resultados - Fase 1: Implementación de Algoritmos y Análisis de Rendimiento.

En esta fase de nuestro estudio, nos adentramos en la ejecución de implementaciones de algoritmos en tres tecnologías diferentes: JavaScript, C y Rust. Los experimentos se llevaron a cabo en diversas configuraciones de sistemas operativos y navegadores web para evaluar el rendimiento en un amplio espectro de entornos.

Tiempos de Ejecución.

Los resultados iniciales revelaron diferencias significativas en los tiempos de ejecución de los algoritmos en las diferentes configuraciones. En las pruebas, se observó que los tiempos de ejecución pueden variar considerablemente según el lenguaje de programación utilizado y la combinación de sistema operativo y navegador. A continuación, se destaca los hallazgos más notables:

1. **Collision Detection**: En general, Rust superó consistentemente en términos de tiempos de ejecución en esta tarea. Se observó una tendencia similar en todas las configuraciones.

---

[13] Enlace al sitio web: https://rockycott.github.io/benchmark-wasm-vs-js/

2. **Fibonacci**: Los algoritmos para calcular la secuencia de Fibonacci mostraron diferencias notables en el rendimiento. C y Rust demostraron una ventaja clara en comparación con JavaScript, aunque este último fue el mejor en la mayoría de las configuraciones.

3. **Multiply Double**: Al igual que en otros algoritmos, Rust logró tiempos de ejecución más rápidos que JavaScript y C en la multiplicación repetida de números en coma flotante. Esta tendencia fue constante en todas las configuraciones.

4. **Multiply Int Vector**: En general, C superó a JavaScript y Rust en términos de tiempos de ejecución en esta tarea.

5. **Multiply Double Vector**: En ciertas configuraciones, JavaScript superó a C y Rust en términos de tiempos de ejecución en esta tarea.

6. **QuickSort**: Los algoritmos de QuickSort en C y Rust también superaron a JavaScript en términos de velocidad de ordenación, pero Rust mostrando menor tiempo, lo que sugiere un rendimiento superior en tareas de procesamiento intensivo.

7. **Sum Int**: En las operaciones de suma, C y Rust mantuvieron una ventaja en términos de tiempos de ejecución sobre JavaScript en la mayoría de los escenarios evaluados.

8. **Sum Double**: En estas operaciones, JavaScript mostró altas capacidades frente a C y Rust para llevar a cabo dicha tarea.

9. **Image Convolute**: En la convolución de imágenes, C y Rust demostraron tiempos de ejecución más eficientes que JavaScript en todas las configuraciones.

10. **Video Convolute**: Los resultados de la convolución de video confirmaron la tendencia general: C y Rust superaron a JavaScript en términos de velocidad de procesamiento.

Velocidades y Comparaciones.

Para evaluar la eficiencia relativa de C y Rust en comparación con JavaScript como referente, se calculó las velocidades promedio en cada configuración. Estas velocidades representan cuántas veces más rápido fue un lenguaje en comparación con JavaScript para un algoritmo y una configuración específicos. Los resultados revelaron que, en la mayoría de los casos, C y Rust superaron significativamente a JavaScript en términos de velocidad.

Varianza.

La varianza nos proporciona una medida de la dispersión de los tiempos de ejecución en los experimentos. Una varrianza baja indica que los tiempos tienden a ser consistentes, mientras que una varianza alta sugiere una mayor variabilidad en el rendimiento.

En general, se observó que JavaScript tendía a tener una varianza más alta en comparación con C y Rust en la mayoría de las configuraciones. Esto indica que, en ciertos escenarios, JavaScript podría experimentar una variabilidad significativa en los tiempos de ejecución, lo que podría afectar negativamente la consistencia del rendimiento.

Promedio tiempos		Js			C			Rust		
		MS Edge	Chrome	FIrefox	MS Edge	Chrome	FIrefox	MS Edge	Chrome	FIrefox
Linux	Collision Detection	464.459	436.1708	653.4424	469.7461	460.5523	724.6824	397.08434	383.0266	478.1681
	Fibonacci	2004.936	1964.221	1893.407	438.0734	434.6262	444.313	0.0001	0.0002	0.0004
	Multiply Double	207.4703	209.1191	236.9391	236.2784	236.5117	236.1815	0.000000	0.00000	0.0000
	Multiply Int Vector	31.2472	31.9052	28.7323	24.7433	23.8606	27.3321	44.5045	43.8192	51.1423
	Multiply Double Vector	27.9357	32.6441	27.8664	38.6503	24.9045	46.7621	38.7315	45.908	46.378
	QuickSort Int	4038.51	4018.635	3818.885	2249.3977	2198.5571	2944.0551	1142.5527	1146.5444	1434.7619
	QuickSort Double	99.8149	101.8245	110.6327	64.3303	64.7437	69.623	48.5678	49.0034	56.0733
	Sum Int	122.3614	123.466	128.8632	92.4764	93.8132	125.7289	86.5807	87.3012	142.1506
	Sum Double	89.1676	89.7189	105.6191	128.2764	128.0413	127.1392	120.6776	120.1572	154.1128
	Image Convolute	41.6241	41.6207	34.9753	28.7766	28.8013	21.4632	22.7061	22.6988	27.4859
	Video Convolute	48.3266	50.2962	46.967	35.9197	34.4929	28.82	28.2533	26.9665	36.34
Windows	Collision Detection	651.7526	518.6268	797.5962	545.2671	480.6713	798.4421	471.4479	460.7712	517.6602
	Fibonacci	2047.743	1845.218	2831.312	436.6653	396.1663	477.328567	0.0003	0.0004	0.00034
	Multiply Double	216.1331	207.8804	206.3405	238.3116	237.84	244.6707	0.0003	0.0001	0.00000314
	Multiply Int Vector	25.0852	27.2596	32.1867	24.9947	23.644	33.3827	37.4255	35.0693	47.0758
	Multiply Double Vector	27.3933	25.7189	29.6855	39.715	37.8656	57.1291	38.1151	36.7563	54.6049
	QuickSort Int	3687.729	3478.794	3777.731	2286.7241	2249.4849	3043.9231	1093.7325	1049.0163	1642.0603
	QuickSort Double	97.5623	91.399	113.435	66.3631	62.4767	78.2458	48.9148	47.3145	58.2876
	Sum Int	125.7268	115.7711	135.369	95.6448	90.3208	197.3837	90.6654	87.8595	146.989
	Sum Double	87.2115	84.403	122.1437	130.3725	126.4444	163.0863	123.1369	121.5572	176.1543
	Image Convolute	34.6622	33.2022	22.4317	21.8342	19.9946	14.4018	24.9806	23.8298	20.2937
	Video Convolute	42.1636	39.3661	39.0071	33.5224	28.8018	27.5318	35.8668	35.5569	37.2942
Mac		Safari								
	Collision Detection	3555.5334			170.7042			155.1008		
	Fibonacci	606.5295			220.1089			0.00		
	Multiply Double	248.2198			248.3017			0.00		
	Multiply Int Vector	7.409			9.471			9.774		
	Multiply Double Vector	6.306			10.842			10.848		
	QuickSort Int	1499.686			1011.275			545.14		
	QuickSort Double	49.363			40.863			65.026		
	Sum Int	46.726			72.202			25.873		
	Sum Double	55.927			66.956			66.641		
	Image Convolute	127.703			8.10			9.10		
	Video Convolute	155.966			9.166			10.171		

Tabla 1*Promedio tiempos algoritmos - Fase 1*

Varianza		Js			C			Rust		
		MS Edge	Chrome	FIrefox	MS Edge	Chrome	FIrefox	MS Edge	Chrome	FIrefox
Linux	Collision Detection	563.1184	240.6666	1635.514	898.57707	23.40997	740.99	333.24802	479.519758	877.365359
	Fibonacci	9634.041	7254.163	1262.832	546.99655	134.4048	573.5478	0.000001	1.9798E-06	3.87879E-06
	Multiply Double	0.553023	1.306269	1.018717	0.048054	0.025709	0.12133	0.000000	0.00000	0.0000
	Multiply Int Vector	0.121451	0.200658	10.89564	5.7950829	0.589856	0.463926	6.6293482	0.553005414	0.720652232
	Multiply Double Vector	2.269764	0.8368	0.696955	3.0281848	0.74975	0.307304	0.3743826	2.08770101	0.382434343
	QuickSort Int	15939.78	16383.98	8596.352	4457.8506	4300.021	5726.388	739.55592	430.7609057	795.3293448
	QuickSort Double	3.076443	13.51323	12.28958	1.799213	2.086365	0.940676	0.7216476	1.007521657	0.252864758
	Sum Int	7.713319	13.3805	329.1461	6.5497667	0.384357	283.0049	1.9248551	0.312228848	1.047975394
	Sum Double	0.55982	0.182761	2.580556	3.0191748	0.918056	0.807949	1.9894346	0.149753697	4.468248646
	Image Convolute	0.00175	0.00198	0.719609	0.0686449	0.072795	0.044606	0.0009776	0.000962182	0.089036556
	Video Convolute	2.288158	10.49836	19.41132	2.1758595	3.440572	7.888283	1.1464345	1.173744192	3.758383838
Windows	Collision Detection	12353.09	1019.047	1155.726	8479.0663	310.894	794.108	625.49448	283.7162915	119.4081757
	Fibonacci	1456.733	1865.248	21344.13	219.3009	206.1328	1318.42	2.9394E-06	3.87879E-06	2.94E-06
	Multiply Double	99.57001	0.819992	261.4268	1.9247328	0.709089	28.0238	2.9394E-06	0.000001	3.14E-06
	Multiply Int Vector	4.507876	5.411521	13.98163	4.5657484	1.797558	7.570497	4.1445422	1.35705102	20.62764481
	Multiply Double Vector	3.715388	2.181541	2.240601	6.3551424	2.149605	5.173881	1.6690737	0.49176698	2.05569999
	QuickSort Int	4312.547	5238.084	8495.035	927.84522	1110	10411.06	370.22592	775.8844276	4988.963074
	QuickSort Double	12.11533	3.187981	43.91445	8.0521812	1.651572	12.37387	2.4086252	0.44552399	11.68779418
	Sum Int	59.8295	0.41762	459.9853	14.470482	8.793151	11.48121	8.3833039	7.022715909	21.67359697
	Sum Double	10.16428	1.323304	381.1522	10.467938	11.49764	2.585787	6.6785913	7.340295111	4.467715667
	Image Convolute	7.159545	0.284264	1.350552	2.3917398	0.572389	0.349247	1.7166926	0.991438343	20.2937
	Video Convolute	33.78729	8.241503	32.26544	19.789445	8.877112	18.98374	1.6290947	1.576775141	37.2942
Mac		Safari								
	Collision Detection	717.5134328			0.010448848			0.020696323		
	Fibonacci	0.064483586			0.049545242			0.00		
	Multiply Double	0.005634303			0.005000111			0.00		
	Multiply Int Vector	0.001029293			0.003493939			0.002751515		
	Multiply Double Vector	0.006832323			0.003268687			0.003531313		
	QuickSort Int	5.640812121			6.95219697			6.646464646		
	QuickSort Double	0.289627273			0.108011111			0.075478788		
	Sum Int	0.013660606			0.089692929			0.027243434		
	Sum Double	0.00259697			0.381882828			0.092544444		
	Image Convolute	2456.560496			0.00			0.00		
	Video Convolute	2403.868125			0.007923232			0.015009091		

Tabla 2*Varianzas algoritmos - Fase 1*

Velocidades		C			Rust		
		MS Edge	Chrome	FIrefox	MS Edge	Chrome	FIrefox
Linux	Collision Detection	0.988745	0.94706	0.901695	1.16967344	1.138748066	1.36655373
	Fibonacci	4.576711	4.519335	4.261425	20049355	9821106	4733516.75
	Multiply Double	0.878076	0.884181	1.003208	2.0747E+11	2.09119E+11	2369391000
	Multiply Int Vector	1.262855	1.33715	1.051229	0.70211327	0.728110052	0.56181087
	Multiply Double Vector	0.722781	1.310771	0.595918	0.72126564	0.711076501	0.60085385
	QuickSort Int	1.795374	1.827851	1.297151	3.53463818	3.504997364	2.6616854
	QuickSort Double	1.5516	1.572732	1.589025	2.05516618	2.077906839	1.97300141
	Sum Int	1.323164	1.316083	1.024929	1.41326416	1.414253183	0.9065259
	Sum Double	0.695121	0.700703	0.830736	0.73889106	0.74667935	0.68533633
	Image Convolute	1.446456	1.445098	1.629547	1.83316818	1.833607944	1.27248153
	Video Convolute	1.345407	1.458161	1.629667	1.7104763	1.865136373	1.29243258
Windows	Collision Detection	1.195291	1.078964	0.998941	1.38244883	1.125562535	1.54077173
	Fibonacci	4.689502	4.657685	5.931579	6825809.33	4613044.5	8327388.14
	Multiply Double	0.906935	0.874035	0.84334	720443.667	2078804	65763800.4
	Multiply Int Vector	1.003621	1.152918	0.964173	0.67027027	0.777306647	0.68372072
	Multiply Double Vector	0.689747	0.679215	0.519621	0.71869941	0.699714063	0.54364169
	QuickSort Int	1.612669	1.546485	1.241073	3.37169143	3.316243894	2.30060406
	QuickSort Double	1.470129	1.462929	1.449726	1.9945354	1.931733401	1.94612576
	Sum Int	1.314518	1.281777	0.685817	1.38671202	1.317684485	0.92094647
	Sum Double	0.668941	0.667511	0.748951	0.7082483	0.694348011	0.6933904
	Image Convolute	1.587519	1.660558	1.557562	1.38756475	1.393305861	1.10535289
	Video Convolute	1.257774	1.366793	1.416802	1.17556069	1.107129699	1.04592939
Mac		Safari					
	Collision Detection	20.82862285			22.92401716		
	Fibonacci	2.755588257			&		
	Multiply Double	0.999670159			&		
	Multiply Int Vector	0.782282758			0.758031512		
	Multiply Double Vector	0.581627006			0.58130531		
	QuickSort Int	1.482965563			2.75101075		
	QuickSort Double	1.208012138			0.759127118		
	Sum Int	0.647156588			1.805975341		
	Sum Double	0.835279885			0.839228103		
	Image Convolute	15.76580247			14.0332967		
	Video Convolute	17.01571023			15.33438207		

Tabla 3*Velocidades algoritmos - Fase 1*

Conclusiones Preliminares.

Esta fase ayudó a comprender las tecnologías y demostrando que las implementaciones en C y Rust con WebAssembly a menudo superan o pueden estar a la par de JavaScript en términos de rendimiento y eficiencia en una variedad de algoritmos y configuraciones. Estos resultados respaldan la idea de que WebAssembly, junto con lenguajes de alto rendimiento como C y Rust, presenta una gran adaptabilidad para el desarrollo web del lado del cliente.

Entre las razones que destacan la elección de Rust sobre C para avanzar a la siguiente fase, se incluyen los siguientes puntos:

1. **Rendimiento Sobresaliente de Rust**: Demostró consistentemente un rendimiento sobresaliente en la mayoría de las implementaciones, superando a C en varios casos, lo que sugiere que es una opción altamente competitiva para tareas de procesamiento intensivo en aplicaciones web avanzadas.

2. **Amplia Aceptación en WebAssembly**: Rust ha ganado una amplia aceptación en el ecosistema de WebAssembly, lo que facilita la integración y el despliegue de código altamente optimizado en el navegador.

3. **Gestor de Paquetes Robusto**: La presencia de un sólido gestor de paquetes en Rust simplifica la administración de dependencias y fomenta la colaboración eficiente en proyectos web complejos.

4. **Sintaxis Moderna y Seguridad Incorporada**: La sintaxis moderna y las características de seguridad incorporadas en Rust lo convierten en una elección sólida para el desarrollo web, ayudando a prevenir errores comunes y a garantizar la estabilidad del código

Basándose en estos resultados y consideraciones, se ha seleccionado a Rust como el lenguaje preferido para la próxima fase de nuestro estudio frente a JavaScript. Esta elección resalta el potencial y la idoneidad de Rust en comparación con C, y sentará las bases para comprender cómo estas tecnologías se desempeñan en una tarea compleja y exigente en términos de rendimiento.

Continuaremos con un análisis de los resultados del caso de estudio de Ray Tracing y se extraerá conclusiones finales sobre el potencial de WebAssembly en aplicaciones web avanzadas.

5.3. Desarrollo del Caso de Estudio - Ray Tracing

Para este caso de estudio de Ray Tracing, se desarrolló un ray tracer en tiempo real que genera imágenes de tamaño 480x360 píxeles. El escenario simulado en el ray tracer se asemeja a una "habitación ajedrezçon esferas flotantes que giran en torno a una central.

Esto proporciona una plataforma de prueba efectiva para evaluar el rendimiento y la eficiencia de las implementaciones de Ray Tracing en Rust y JavaScript con WebAssembly.

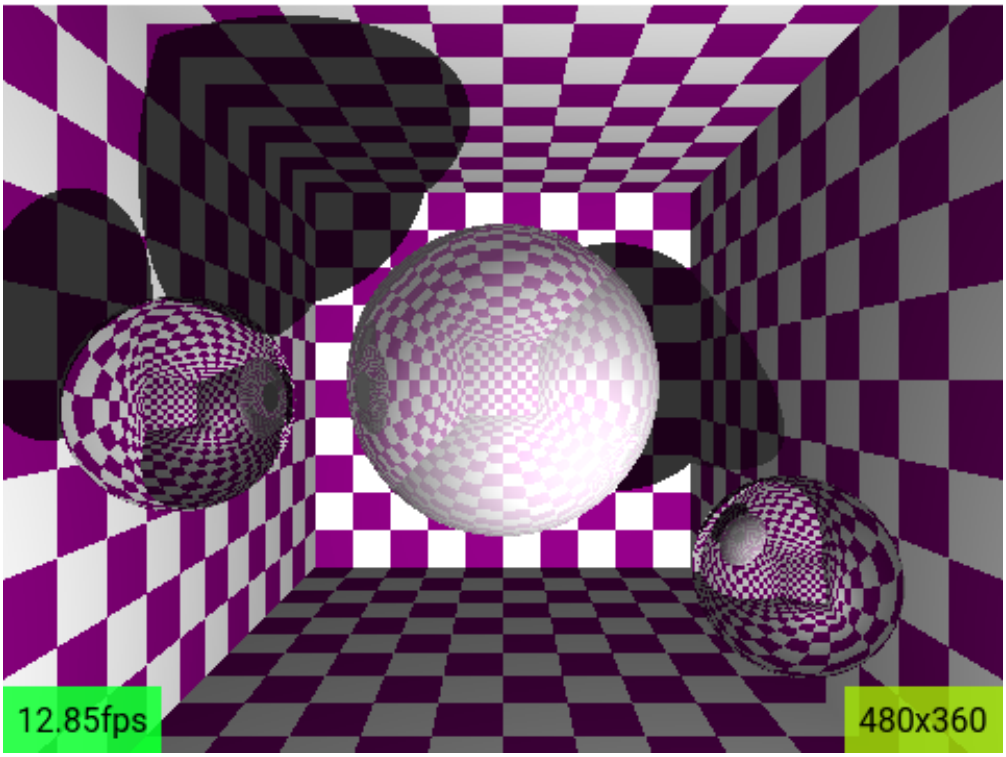


Figura 16. Ray tracer final - Linux

La implementación del Ray Tracing se basa en una implementación ingenua, ya que no quería hacer el trazador de rayos más rápido o con más funciones, por un lado, esa no es realmente la vía de este proyecto. El objetivo aquí era ver cuánto podía exigir a la web y cómo funcionaría en comparación con JavaScript. Entonces a continuación, se resumen los aspectos clave a tener en cuenta y cómo se han integrado:

Configuración Inicial del Lienzo (Canvas).

La base de nuestra de la implementación comienza con la configuración de un elemento HTML canvas, que actuará como el lienzo en el que se generará las imágenes tridimensionales. Para optimizar el rendimiento y la precisión, se trabaó directamente con los datos de píxeles en lugar de utilizar las funciones de dibujo tradicionales de HTML5 Canvas.

Definición de la Escena 3D.

La creación de la escena tridimensional es fundamental. Aquí se define el mundo virtual en el que se desarrollará el Ray Tracing. Los componentes principales de la escena incluyen:

- **Objetos en la Escena**: La escena está poblada con objetos, principalmente esferas, que interactúan con la luz. Cada objeto tiene propiedades únicas como su posición, tamaño, color, reflectividad y coeficientes de iluminación.
- **Fuentes de Luz**: Las luces en la escena actúan como fuentes de iluminación. Son puntos en el espacio que emiten luz y afectan la iluminación de los objetos circundantes.

Configuración de la Cámara.

La cámara en la escena se configura con los siguientes parámetros:

- **Posición de la Cámara**: Se define la ubicación de la cámara en el espacio virtual desde donde se capturarán las imágenes.
- **Campo de Visión (Field of View)**: Se establece el campo de visión de la cámara, que determina el ángulo desde el lado derecho hasta el lado izquierdo de su marco visual.
- **Vector de Dirección de la Cámara**: Se especifica un vector que determina la dirección en la que apunta la cámara, lo que define la perspectiva de la escena.

Generación de Imágenes con Ray Tracing.

El proceso central de la implementación es la generación de imágenes tridimensionales realistas utilizando Ray Tracing. Aquí hay una vista general de cómo funciona:

- **Lanzamiento de Rayos**: Para cada píxel en el lienzo, se traza un rayo que parte desde la cámara y atraviesa la escena virtual. Este rayo se utiliza para determinar cómo la luz interactúa con los objetos en su camino.
- **Intersección con Objetos**: Se calcula la intersección del rayo con los objetos en la escena. Esto implica comprobar si el rayo toca alguna superficie de objeto y, de ser así, dónde ocurre esa intersección.

- **Modelado de Iluminación**: Se utiliza técnicas de iluminación, como el sombreado lambertiano, para calcular el color de los píxeles. Esto tiene en cuenta la cantidad de luz que incide en cada punto de la superficie de los objetos y su capacidad para reflejarla.
- **Reflexiones Especulares**: Para objetos con propiedades especulares, se calcula las reflexiones especulares, lo que agrega reflejos y brillo a la imagen.

Actualización Continua y Renderizado.

La generación de imágenes se realiza en tiempo real, lo que significa que continuamente se actualiza los píxeles en el lienzo para representar la escena. Esto se logra mediante una función que actualiza los datos de píxeles en el lienzo en cada fotograma. Dicha función se llama repetidamente para crear la ilusión de movimiento y cambios en la escena.

Cada parte de esta implementación desempeña un papel esencial en la creación de imágenes tridimensionales realistas y atractivas.

Resumen.

La implementación del Ray Tracing en la aplicación web a pesar de ser creado como una implementaciín ingenua, este combina conceptos avanzados de gráficos 3D con una interfaz de usuario interactiva y amigable. Esta implementación personalizada permite explorar el potencial de Rust y JavaScript, especialmente cuando se utilizan en conjunto con WebAssembly, para lograr resultados grandiosos.

Métricas y Criterios de Evaluación.

Durante esta fase, se utilizarán las siguientes métricas y criterios para evaluar el rendimiento y la eficiencia de las implementaciones:

1. **Tiempo de renderizado**: El tiempo que se demora en renderizar la escena.
2. **Velocidad de fotogramas**: Cuántos fotogramas por segundo (FPS) puede generar cada implementación en tiempo real.
3. **Calidad visual**: Realizar comparaciones visuales para determinar la calidad de las imágenes generadas.
4. **Escalabilidad**: Evaluar cómo cada implementación maneja escenas y modelos más complejos.

5.4. Evaluación y Análisis de Resultados del Ray Tracing

En esta fase, se analizarán los resultados obtenidos de la implementación del ray tracing en tiempo real. Se presentarán datos relevantes, gráficos y análisis para evaluar la eficiencia, calidad y adaptabilidad de la implementación en una variedad de configuraciones.

Todo el código fuente y los recursos relacionados con el motor de Ray Tracing y sus resultados se encuentran disponibles en el repositorio público de GitHub[14]. Este repositorio contiene también acceso a un despliegue en GitHub Pages para su uso personalizado[15].

Presentación de Datos.

Los resultados se basan en mediciones de tiempo de renderizado y velocidad de cuadros por segundo (FPS) durante un período de 3 minutos en cada configuración.

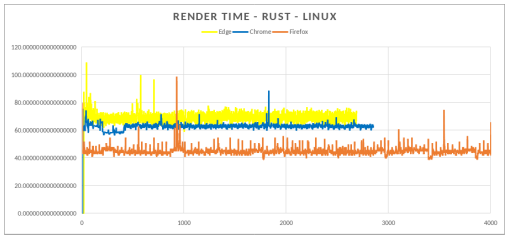
- En los sistemas Linux y Windows, se realizaron pruebas en tres navegadores web diferentes: Microsoft Edge, Google Chrome y Mozilla Firefox.
- En el sistema operativo macOS, se realizaron pruebas exclusivamente en el navegador Safari. La elección de Safari como único navegador se debe a que en este sistema operativo, el porcentaje de uso de otros navegadores es notablemente bajo, lo que limita la capacidad de realizar pruebas significativas en múltiples navegadores.

---

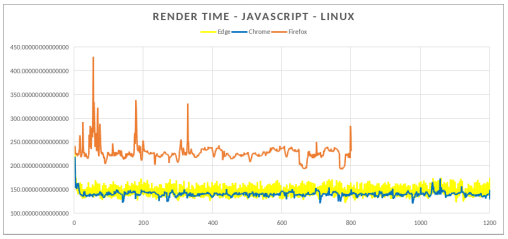
[14] Enlace al repositorio de GitHub: https://github.com/RockyCott/wasm-rust-raytracer-js

[15] Enlace al sitio web: https://rockycott.github.io/wasm-rust-raytracer-js/

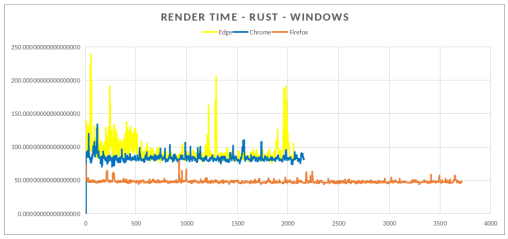
Rendimiento en tiempo de renderizado.



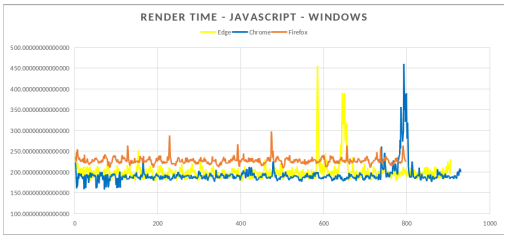
(a) Render Time - Rust - Linux



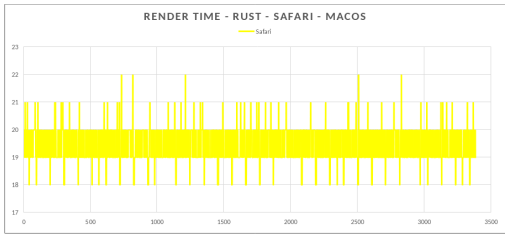
(b) Render Time - JavaScript - Linux



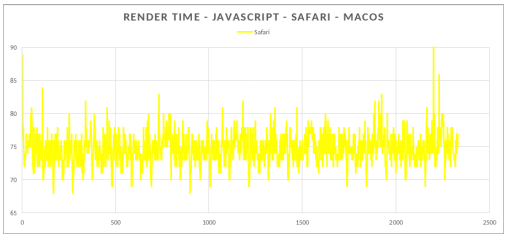
(c) Render Time - Rust - Windows



(d) Render Time - JavaScript - Windows



(e) Render Time - Rust - Safari - MacOS



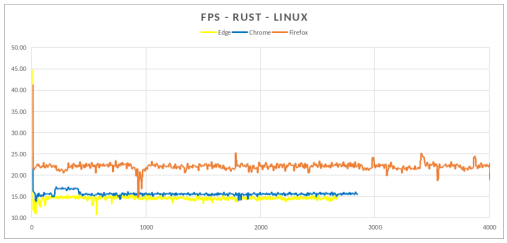
(f) Render Time - JavaScript - Safari - MacOS

Figura 17. Gráficos de tiempos renderizado ray tracing

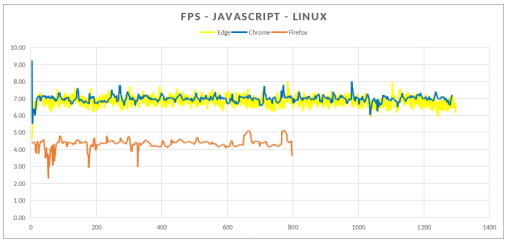
Promedio tiempos renderizado	Linux		Windows		MacOS	
	Rust	JavaScript	Rust	JavaScript	Rust	JavaScript
Edge	67.0456780924007000	144.47218798151200	91.71929824561430000	201.41692477876000000		
Chrome	63.0226123595502000	141.43919129082600	84.00440835266970000	194.33455328310100	X	
Firefox	44.97804637	228.7822278	48.49367089	228.7650754		
Safari	X		X		19	74.634178

Tabla 4*Promedios tiempos renderizados*

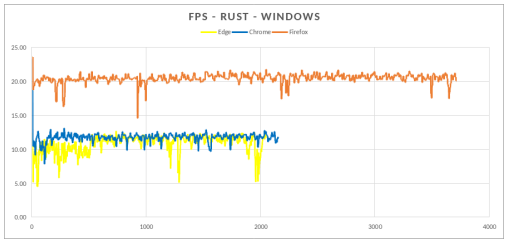
Rendimiento en tiempo real (FPS).



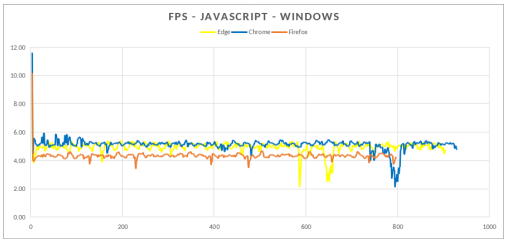
(a) FPS - Rust - Linux



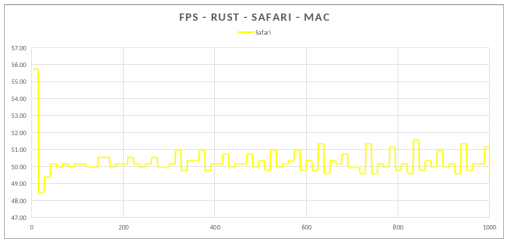
(b) FPS - JavaScript - Linux



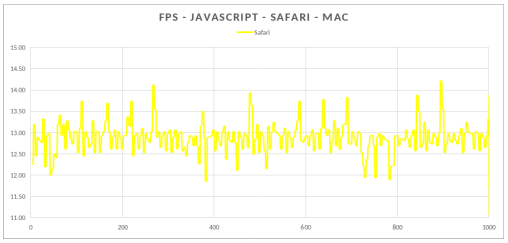
(c) FPS - Rust - Windows



(d) FPS - JavaScript - Windows



(e) FPS - Rust - Safari - MacOS



(f) FPS - JavaScript - Safari - MacOS

Figura 18. Gráficos de FPS ray tracing

Promedio FPS	Linux		Windows		MacOS	
	Rust	JavaScript	Rust	JavaScript	Rust	JavaScript
Edge	14.70	6.89	10.92	4.98		
Chrome	15.71	7.015	11.82	5.15	X	
Firefox	22.16	4.38	20.52	4.38		
Safari	X		X		50.30	12.85

Tabla 5*Promedios FPS*

Conclusiones Preliminares.

En base a los resultados obtenidos las diferentes configuraciones, se ha demostrado que WebAssembly se presenta como una tecnología idónea para la optimización de aplicaciones web de alto rendimiento, como el Ray Tracing en tiempo real.

El estudio comparativo entre las implementaciones de WebAssembly y JavaScript reveló tendencias interesantes. En promedio, las implementaciones basadas en WebAssembly demostraron un rendimiento superior en todas las configuraciones evaluadas. La cantidad de cuadros por segundo (FPS) fue consistentemente mayor, lo que se tradujo en una experiencia de usuario más fluida.

Además, el tiempo de renderizado fue significativamente menor con WebAssembly en comparación con JavaScript. Este resultado es especialmente relevante en aplicaciones intensivas donde el tiempo de respuesta es crucial.

- **Versatilidad de WebAssembly**: Los hallazgos subrayan a WebAssembly como una tecnología capaz de optimizar aplicaciones web de alto rendimiento. Es relevante destacar que no se busca determinar una configuración específica como ganadora, dado que las configuraciones de sistemas operativos y hardware variaban considerablemente y no estaban igualadas ni calibradas. En cambio, la exploración se centró en comprender cómo WebAssembly puede adaptarse y poseer un potencial significativo para mejorar el rendimiento en línea en una amplia gama de configuraciones.

- **Impacto del Sistema Operativo**: Aunque se observó que el sistema operativo puede influir en los resultados, con macOS mostrando un rendimiento sobresaliente en primera instancia, es esencial comprender que esto se debe en gran medida al hardware de alto rendimiento de macOS utilizado en comparación con otras configuraciones. Aún así los resultados refuerzan a WebAssembly como una tecnología capaz de funcionar de manera efectiva en diferentes entornos, independientemente del sistema operativo utilizado.

- **Consistencia de WebAssembly**: La consistencia en los resultados es un aspecto crucial que se destacó en este estudio. La implementación en WebAssembly proporcionó resultados más consistentes en comparación con JavaScript. Esto respalda la confiabilidad de WebAssembly para aplicaciones web de alto rendimiento, lo que sugiere su viabilidad en una amplia variedad de configuraciones y escenarios de uso. Esta característica es especialmente importante para garantizar una experiencia de usuario fluida y receptiva.

En conjunto, estas conclusiones preliminares resaltan la prometedora naturaleza de WebAssembly como una tecnología versátil y confiable que puede optimizar aplicaciones web de alto rendimiento, como el Ray Tracing en tiempo real, en diversas configuraciones del mundo real. Aunque no se buscó establecer una configuración específica como ganadora, los resultados indican claramente el potencial de WebAssembly para mejorar el rendimiento en línea y ofrecer experiencias de usuario más fluidas en una variedad de entornos. La evolución y adopción continua de WebAssembly en la industria prometen un futuro emocionante para el desarrollo web de alto rendimiento.

6. Conclusiones

En el transcurso de este proyecto, se ha navegado por un emocionante viaje a través del vasto mundo de la tecnología web a pesar de tratarse de implementaciones ordinarias. Se ha explorado, experimentado y analizado la implementación de aplicaciones de algoritmos y ray tracing en tiempo real, un desafío ambicioso.

Hemos evaluado la idoneidad de WebAssembly como una herramienta para mejorar los procesos que pueden estar presentes como es el renderizado de imágenes, y los resultados han sido esclarecedores. A través de la caracterización de lenguajes de alto nivel implementados en WebAssembly, se ha demostrado que ofrece un rendimiento impresionante, convirtiéndolo en un candidato prometedor para acelerar aplicaciones web.

Además, se ha establecido un escenario específico para el caso de estudio y definido métricas de referencia utilizando una implementación en JavaScript como punto de partida. Esta base permitió comparar el rendimiento de WebAssembly de manera justa y precisa. La implementación exitosa del caso de estudio en WebAssembly refuerza la viabilidad de esta tecnología en aplicaciones web del mundo real.

Se ha demostrado que WebAssembly ofrece un rendimiento significativamente mejorado en términos de velocidad y eficiencia. Esto sugiere que WebAssembly tiene un potencial revolucionario para impulsar la calidad y la interactividad de las aplicaciones web.

En estas travesías, se ha descubierto la sorprendente versatilidad de WebAssembly, una tecnología que emerge como un faro brillante en el horizonte de la web. WebAssembly, con su capacidad para ejecutar código de manera eficiente y aprovechar el poder de los navegadores modernos, nos ha demostrado que el límite entre las aplicaciones web y las aplicaciones nativas se desvanece gradualmente.

El ray tracing en tiempo real, una técnica que alguna vez se consideró exclusiva de las potentes estaciones de trabajo, ahora se encuentra al alcance de nuestros navegadores. Gracias a WebAssembly, hemos presenciado cómo los límites de la tecnología web se expanden, brindando a los desarrolladores la capacidad de crear experiencias visuales asombrosas y dinámicas directamente en la web.

Si bien WebAssembly no busca establecer una supremacía absoluta, representa un hito significativo en el camino hacia un futuro donde las aplicaciones web sean más rápidas, versátiles y poderosas que nunca.

Como comunidad, estamos en un punto de inflexión emocionante en el desarrollo web. La visión de explorar las posibilidades de WebAssembly ha demostrado ser una experiencia valiosa y prometedora. Espero que este proyecto inspire a otros a continuar explorando las vastas posibilidades que ofrece WebAssembly y en lo posible junto a Rust como un dúo dinámico a considerar su aplicación en sus propios proyectos.

7. Trabajo futuro

A medida que el mundo de la tecnología web sigue evolucionando, existen numerosas oportunidades y caminos a seguir para continuar explorando y aprovechando al máximo las capacidades de WebAssembly. Algunas áreas de trabajo a futuro incluyen:

- **Optimización Continua**: Dado el éxito de WebAssembly en los resultados de las implementaciones para la web, se podría considerar una optimización continua. Esto podría incluir la exploración de técnicas avanzadas de compilación y optimización de código en WebAssembly para lograr un rendimiento aún mejor como lo son aplicaciones web 3D.
- **Integración con Frameworks JavaScript**: Explorar la integración de WebAssembly en frameworks y bibliotecas populares de desarrollo web basadas en JavaScript como Angular, React, Vue, entre muchos otros. Esto permitiría a los desarrolladores aprovechar fácilmente las ventajas de WebAssembly en sus proyectos sin la necesidad de reescribir completamente su código, ya que hoy en día desarrollar la web es a través de frameworks o librerías.
- **Ampliación de Casos de Estudio**: Ampliar el conjunto de casos de estudio para abordar una variedad de aplicaciones web del mundo real. Esto proporcionaría una visión más completa de cómo WebAssembly puede impactar diferentes industrias y escenarios de uso.

- **Investigación en Seguridad**: Realizar investigaciones adicionales sobre las implicaciones de seguridad de WebAssembly en aplicaciones web. A medida que WebAssembly se convierte en una parte más integral de la web, es fundamental comprender y abordar los posibles problemas de seguridad.

- **Exploración de Nuevas Aplicaciones**: Investigar nuevas aplicaciones y casos de uso que aún no se han explorado con WebAssembly. Esta tecnología tiene el potencial de transformar una amplia gama de industrias, y descubrir nuevas formas de aplicarla podría conducir a avances innovadores.

- **Educación y Divulgación**: Contribuir a la educación y divulgación tanto de WebAssembly como de Rust. Esto podría incluir la creación de recursos educativos, tutoriales y presentaciones para ayudar a otros a comprender y adoptar esta tecnología.

- **Exploración de Frameworks Basados en WebAssembly**: La investigación y el desarrollo de aplicaciones web basadas en WebAssembly pueden beneficiarse enormemente de frameworks específicos como Blazor, Yew, Leptos, entre otros. Estos frameworks representan un cambio en el paradigma donde el lenguaje principal ya no es JavaScript, sino que se basa en lenguajes altamente eficientes como C# y Rust. Se recomienda profundizar en su uso y considerar cómo pueden aplicarse en futuros proyectos.

- **Optimización de Algoritmos y Almacenamiento de Datos**: A medida que las aplicaciones web se vuelven más complejas y demandantes en términos de rendimiento, se requerirá una investigación continua sobre la optimización de algoritmos y la gestión eficiente de datos.

Explorar técnicas de optimización específicas para WebAssembly puede llevar a mejoras significativas en la velocidad y eficiencia de las aplicaciones web.

- **Colaboración entre WebAssembly y Rust**: La sinergia entre WebAssembly y Rust es un campo emocionante para la investigación y el desarrollo. Explorar cómo estos dos se complementan mutuamente para crear aplicaciones web de alto rendimiento puede ser una dirección valiosa para futuros proyectos. El trabajo en equipo entre WebAssembly y Rust puede desbloquear aún más potencial en el desarrollo web.

- **Expansión de WebAssembly Más Allá de los Navegadores Web**: WebAssembly no se limita únicamente a aplicaciones web en navegadores. Su capacidad para ejecutar código de alto rendimiento de manera segura y portátil lo convierte en una opción atractiva para otras áreas de desarrollo. Explorar cómo WebAssembly puede aplicarse en entornos no web, como servidores, sistemas embebidos y aplicaciones de escritorio, es un campo prometedor. Investigar su utilidad en contextos más amplios puede desencadenar innovaciones significativas en la industria de la tecnología.

- **Alto Rendimiento en la Web**: WebAssembly ha demostrado ser una tecnología prometedora, en el futuro, explorar aún más las capacidades de WebAssembly para la computación de alto rendimiento (HPC) en la web podría llevar a avances significativos. Esto incluye investigar cómo WebAssembly puede utilizarse eficazmente para tareas intensivas en cómputo, como simulaciones científicas, aprendizaje automático, inteligencia artificial y procesamiento de datos a gran escala directamente en el navegador web. Estas investigaciones pueden

abrir nuevas posibilidades para el procesamiento de datos y la visualización en tiempo real en la web, allanando el camino hacia aplicaciones web aún más poderosas.

- **Evolución Continua de WebAssembly**: Al igual que JavaScript evolucionó desde sus humildes comienzos hasta convertirse en un lenguaje de programación versátil y ampliamente utilizado como base para demasiadas herramientas de desarrollo, WebAssembly también está en un camino de evolución constante. A medida que WebAssembly madura, es probable que veamos un crecimiento significativo en sus capacidades y su adopción en una variedad de aplicaciones web, y se convierta en algo común de ver para los desarrolladores.

En resumen, el proyecto actual ha demostrado la idoneidad de WebAssembly para optimizar la web en determinadas aplicaciones web. Este es solo el comienzo de un emocionante viaje en el vasto mundo de WebAssembly, donde las tecnologías basadas en WebAssembly con lenguajes alternativos están tomando protagonismo. A medida que continuamos explorando, esperamos ser parte de futuros desarrollos que aprovechen al máximo esta tecnología y sigan impulsando la innovación en el desarrollo web.

Bibliografía

Aila, T., & Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. *Proceedings of the Conference on High Performance Graphics 2009*, 145-149. https://doi.org/10.1145/1572769.1572792

Buck, J., & Hogan, B. P. (2019). *The Ray Tracer Challenge: A Test-Driven Guide to Your First 3D Renderer*. Pragmatic Bookshelf.

Cook, R. L., Porter, T., & Carpenter, L. (1984). Distributed Ray Tracing. *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, 137-145. https://doi.org/10.1145/800031.808590

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., & Bastien, J. (2017). Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.*, *52*(6), 185-200. https://doi.org/10.1145/3140587.3062363

Main — Emscripten 3.1.44-git dev documentation. (2015). https://emscripten.org/index.html

Mozilla. (2023, mayo). WebAssembly.

Osmani, A. (2018, agosto). The Cost Of JavaScript In 2018 - Addy Osmani - Medium. https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4

Overflow, S. (2023). Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages

Rademacher, P. (1997). Ray Tracing: Graphics for the Masses. *XRDS*, *3*(4), 3-7. https://doi.org/10.1145/270955.270962

Rodríguez, T. (2019, mayo). Por qué Rust es el lenguaje más amado por muchos programadores aunque es un gran desconocido aún. https://www.genbeta.com/desarrollo/que-rust-lenguaje-amado-muchos-programadores-gran-desconocido

Rust+Wasm. (2023). Introduction - Rust and WebAssembly. https://rustwasm.github.io/docs/book/

Shirley, P., & Morley, R. (2008). *Realistic Ray Tracing, Second Edition*. Taylor & Francis. https://books.google.com.co/books?id=knpN6mnhJ8QC

Šipek, M., Muharemagić, D., Mihaljević, B., & Radovan, A. (2021). Next-generation Web Applications with WebAssembly and TruffleWasm. *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 1695-1700. https://doi.org/10.23919/MIPRO52101.2021.9596883

Top lenguajes de programación 2023. (2023, septiembre). https://www.stackscale.com/es/blog/lenguajes-programacion-mas-populares/

W3C. (2019, diciembre). world wide web consortium w3c brings a new language to the web as webassembly becomes a w3c recommendation 2019. https://www.w3.org/press-releases/2019/wasm/

WebAssembly. (2023). WebAssembly. https://webassembly.org/

Zakai, A., Haas, A., Rossberg, A., Titzer, B., Gohman, D., Schuff, D., Bastien, J., Wagner, L., & Holman, M. (2017). Bringing the Web up to Speed with WebAssembly. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Apéndices

Apéndice A. Especificaciones de los Equipos Utilizados

Equipo 1: Portátil ASUS X507UF

Sistema Operativo: Windows 11, versión 22H2 (64 bits).

Sistema Operativo: Manjaro Linux, Kernel 6.1.51-1 (x86_64).

Memoria RAM: 16 GB.

Procesador: Intel(R) Core(TM) i5-8250U CPU @ 1.60 GHz - 1.80 GHz.

Tarjeta Gráfica: NVIDIA GeForce MX130.

Equipo 2: MacBook Pro

Sistema Operativo: macOS Ventura 13.2.

Memoria RAM: 16 GB.

Procesador: M1 Pro.

Apéndice B. Especificaciones de Navegadores Utilizados

Linux

```
Navegador Web #1: Microsoft Edge

Versión: 116.0.1938.54 (64 bits)

Navegador Web #2: Google Chrome

Versión: 116.0.5845.96 (64 bits)

Navegador Web #3: Firefox

Versión: 117 (64 bits)
```

Windows

```
Navegador Web #1: Microsoft Edge

Versión: 117.0.2045.43 (64 bits)

Navegador Web #2: Google Chrome

Versión: 117.0.5938.92 (64 bits)

Navegador Web #3: Firefox

Versión: 117 (64 bits)
```

Safari en macOS

```
Navegador Web: Safari

Versión: 16.3
```

Apéndice C. Implementaciones Y Despliegues

Repositorio de GitHub - Fase 1 de Algoritmos

Enlace al repositorio de GitHub:

https://github.com/RockyCott/benchmark-wasm-vs-js

Enlace a la versión desplegada en GitHub Pages:

https://rockycott.github.io/benchmark-wasm-vs-js/

Repositorio de GitHub - Fase 2 de Ray Tracer

Enlace al repositorio de GitHub:

https://github.com/RockyCott/wasm-rust-raytracer-js

Enlace a la versión desplegada en GitHub Pages:

https://rockycott.github.io/wasm-rust-raytracer-js/