INFORME TÉCNICO COMPLETO

Free Vector Image

Vectorizador Profesional con VTracer WASM + SVGO + IA

Responsable del Proyecto

Eduardo Andrés Fierro Duque

Santiago de Chile — Agosto 2026

Índice

1. Resumen Ejecutivo

Free Vector Image es una herramienta de vectorización de imágenes que opera 100% en el navegador. Convierte imágenes raster (PNG, JPG, WebP) a gráficos vectoriales (SVG, EPS) o PNG escalable utilizando tecnología WebAssembly (VTracer) con optimización posterior mediante SVGO. La herramienta implementa una arquitectura de Web Workers que mantiene la UI completamente responsive durante el procesamiento pesado de vectorización, que puede tomar desde milisegundos hasta varios segundos dependiendo del tamaño y complejidad de la imagen.

El proyecto consta de 14 archivos JavaScript modulares (ES modules), 1 archivo CSS, liberías WASM (vtracer.wasm), una librería de optimización SVG (svgo.browser.js) y un motor de vectorización alternativo (imagetracer.js) como fallback cuando VTracer no está disponible.

2. Stack Tecnológico

Capa	Tecnología
Motor Principal	VTracer 1.0.0-alpha.1 (VisionCortex) — Vectorizador en Rust compilado a WebAssembly
Motor Fallback	ImageTracer.js — Vectorizador en JavaScript puro para navegadores sin soporte WASM
Optimizador	SVGO (SVG Optimizer) — Reduce tamaño de SVG eliminando metadata, comentarios, agrupaciones innecesarias y precisión redundante
Workers	Web Workers API con ES modules — Procesamiento en hilo separado, transferencia de buffers por referencia (Transferable objects)
Frontend	HTML5 + CSS3 + JavaScript ES modules (14 archivos). Sin frameworks UI.
Preprocesamiento	Canvas 2D API — Filtros de color, blanco/negro con umbral, escala de grises, posterización, Gaussian blur
Entrada	File API + Drag & Drop + Portapapeles (Ctrl+V) — Formatos: PNG, JPG, WebP
Exportación	SVG (descarga directa), PNG rasterizado a resolución configurable (1x-4x), EPS (Illustrator)

3. Arquitectura del Sistema

La arquitectura se basa en el patrón de desacoplamiento Main Thread ↔ Worker Thread. El hilo principal (main.js) maneja exclusivamente la UI, el estado de la aplicación y la comunicación con el Worker. Toda la lógica pesada de vectorización (VTracer WASM, ImageTracer, SVGO) se ejecuta en un Web Worker dedicado, evitando el bloqueo de la interfaz de usuario.

[MAIN THREAD] [WORKER THREAD]
main.js worker.js
■■■ Estado (state) ■■■ import VTracer WASM
■■■ UI (botones, sliders, zoom) ■■■ import SVGO browser
■■■ DOM events ■■■ import ImageTracer
■■■ preprocessor.js → Canvas 2D ■■■ import preprocessor (filtros)
■■■ postMessage(imageData) ■■■■■■→ ■■■ onmessage:
■■■ preprocesar imagen
■■■ VTracer WASM → SVG string
■■■ SVGO optimizar
■■■ postMessage(result) → Main

4. Análisis por Archivo

4.1 main.js — Hilo Principal (578 líneas)

Módulo central que orquesta toda la aplicación. Responsabilidades:

- Gestión del ciclo de vida del Worker: creación, terminación y recreación tras cada vectorización o crash. La función _createWorker() (línea 17) implementa un patrón de "worker fresco": después de cada resultado o error, se termina el worker actual y se crea uno nuevo, evitando memory leaks acumulativos en el heap de WASM.
- Manejo de mensajes del Worker (líneas 24-64): onmessage despacha según msg.type: "ready", "progress", "result", "error", "eps-result". Cada tipo actualiza el estado y la UI correspondientemente.
- Sistema de timeout de seguridad (línea 108-116): si el worker no responde en 45 segundos (posible crash WASM), se muestra un mensaje de error y se recrea. Este es un patrón defensivo crucial al trabajar con WebAssembly que puede fallar silenciosamente.
- Transferencia de ImageData por referencia (línea 140): el buffer de píxeles se transfiere al worker usando la lista de transferibles `[imageData.data.buffer]`, lo que significa que el main thread pierde el acceso (neutered) pero el worker lo recibe sin copia de memoria. Luego se repinta el canvas desde la imagen original (HTMLImageElement) para restaurar la vista.
- Inicialización de módulos (líneas 572-578): imageLoader, sidebar, zoom, export, UI. Boot sequence que conecta todos los componentes.

4.2 worker.js — Web Worker de Vectorización

Ejecutado en un hilo separado. Importa dinámicamente los módulos necesarios (vtracerBridge, svgo.browser, imagetracer, preprocessor). Al recibir un mensaje "vectorize", ejecuta la tubería de procesamiento: preprocesar (filtros) → vectorizar (VTracer WASM o ImageTracer) → optimizar (SVGO) → devolver resultado. Todo ocurre sin tocar el DOM, lo que es la restricción fundamental de los Web Workers.

4.3 vtracerBridge.js — Puente WASM

Carga e inicializa el módulo WebAssembly de VTracer. Maneja la complejidad de la carga asíncrona de WASM (fetch + instantiateStreaming/compile). Expone una función vtrace(imageData, options) que configura los parámetros del algoritmo y ejecuta la vectorización. Los parámetros incluyen: mode (spline/polygon/pixel), hierarchical (stacked/cutout), cornerThreshold, maxIterations, filterSpeckle, colorPrecision, spliceThreshold, lengthThreshold, layerDifference, pathPrecision.

4.4 preprocessor.js — Preprocesamiento

Aplica transformaciones a la imagen antes de la vectorización. Soporta 4 modos: color (sin cambios), bw (blanco/negro con umbral configurable), grayscale (escala de grises vía luminance), y posterize (reducción de paleta a N niveles). También implementa un filtro Gaussian Blur opcional para suavizar ruido JPEG antes de vectorizar.

4.5 exporter.js — Exportación

Maneja la exportación en 3 formatos. SVG: descarga directa del string SVG como blob text/xml. PNG: renderiza el SVG en un canvas con la escala configurada (1x, 2x, 4x) y exporta como image/png. EPS: la conversión SVG→EPS se ejecuta en el Worker (para no bloquear la UI) y se descarga como application/postscript.

5. Fórmulas Matemáticas

5.1 Algoritmo de Vectorización VTracer

VTracer implementa un pipeline de vectorización basado en los siguientes principios matemáticos:

1. CUANTIZACIÓN DE COLOR:

Reduce la imagen a un número limitado de colores planos (controlado por colorPrecision).

Precisión de color = 2^colorPrecision niveles por canal.

Para colorPrecision=3: $2^3 = 8$ niveles por canal = $8^3 = 512$ colores máximos.

Para colorPrecision=6: 2■ = 64 niveles por canal = 64³ = 262,144 colores máximos.

2. DETECCIÓN DE CONTORNOS:

Identifica los bordes entre regiones de diferente color.

cornerThreshold (15°-180°): ángulo mínimo para considerar un punto como esquina.

Ángulos más pequeños = más esquinas detectadas = más fidelidad pero más nodos.

3. APROXIMACIÓN DE CURVAS (Modo Spline):

Ajusta curvas de Bézier cúbicas a los contornos detectados.

maxIterations (1-15): número de iteraciones de suavizado.

Más iteraciones producen curvas más suaves pero pueden perder detalle fino.

4. FILTRADO DE RUIDO:

filterSpeckle (0-32px): regiones más pequeñas que este umbral se ignoran.

Elimina "motas" de color aisladas que son artefactos de compresión JPEG.

5. OPTIMIZACIÓN DE TRAZOS:

spliceThreshold: umbral para unir segmentos de curva adyacentes.

lengthThreshold: longitud mínima de un trazo para ser incluido.

5.2 Filtros de Preprocesamiento

■■ *BLANCO Y NEGRO (Umbral)* ■■

Para cada píxel (R, G, B):

gris = 0.299×R + 0.587×G + 0.114×B (luminancia perceptual)

si gris < umbral: píxel = negro (0,0,0)

si gris ≥ umbral: píxel = blanco (255,255,255)

Los coeficientes 0.299, 0.587, 0.114 corresponden a la sensibilidad del ojo humano a cada canal de color (somos más sensibles al verde).

■■ *POSTERIZAR* ■■

Para cada canal C (R, G, B):

C_posterizado = floor(C × niveles / 256) × (256 / (niveles - 1))

Ejemplo con niveles=4: C ∈ {0, 85, 170, 255}

■■ GAUSSIAN BLUR ■■

Convolución 2D con kernel gaussiano 5×5:

G(x,y) = (1/2πσ²) × e^(-(x²+y²)/(2σ²))

σ ≈ 1.0 para un desenfoque sutil (anti-ruido JPEG)

5.3 Optimización SVG (SVGO)

SVGO aplica transformaciones algebraicas al código SVG para reducir su tamaño sin cambiar la apariencia visual:

- Redondeo de coordenadas a precisión configurable (ej. 1 decimal)

- Colapso de grupos anidados innecesarios

- Conversión de estilos CSS a atributos inline cuando es más corto

- Eliminación de metadata, comentarios, IDs no utilizados

- Fusión de paths adyacentes del mismo color

- Reducción típica de tamaño: 30-60%

6. Métodos Web Utilizados

- WebAssembly (WASM): VTracer compilado de Rust a WASM. Ejecución a velocidad casi nativa en el navegador. Carga asíncrona con fetch + WebAssembly.instantiateStreaming. Memoria lineal compartida.
- Web Workers (ES Modules): Procesamiento de vectorización en hilo separado. Comunicación vía postMessage con Transferable objects (ImageData.data.buffer transferido por referencia, cero copia). Worker recreado tras cada operación para prevenir memory leaks.
- Canvas 2D API: Preprocesamiento de imagen: drawImage, getImageData, putImageData. Filtros de color: iteración sobre Uint8ClampedArray (4 bytes por píxel). Gaussian blur vía convolución.
- File API + Drag & Drop: Entrada de imágenes: <input type="file"> + dragenter/dragover/dragleave/drop. Acepta PNG, JPG, WebP. También clipboard (Ctrl+V) vía evento paste.
- SVG DOM API: Renderizado de SVG en el panel de vista previa. Manipulación del DOM SVG para modo edición. Zoom y paneo con CSS transform.

- Blob / URL.createObjectURL: Descarga de archivos SVG, PNG y EPS. Creación de Blob con MIME type apropiado. Enlace <a> con download + click() programático.
- localStorage: No utilizado. Todo es volátil.
- CSS Transform: Zoom y paneo en paneles de vista previa: transform: translate(panX, panY) scale(zoom). Transiciones suaves con transition.
- Touch Events: Pinch zoom (2 dedos) en paneles: cálculo de distancia entre touches, ratio de escala. Pan con 1 dedo.

7. Modelo de RAM y Rendimiento

■■ *Memoria por Imagen* ■■

Imagen original (ImageData): W×H×4 bytes (ej. 1920×1080 ≈ 8 MB)

Buffer transferido al Worker: 0 bytes en main thread (transferido)

Código SVG resultante: 100 KB - 5 MB (dependiendo de complejidad)

Heap WASM: ~20-50 MB (memoria lineal de VTracer)

SVGO en Worker: ~2-5 MB adicionales durante optimización

Tiempos típicos de vectorización (hardware moderno): imagen 800×600: 200-500ms. Imagen 1920×1080: 1-3 segundos. Imagen 4000×3000: 5-15 segundos. El tiempo escala aproximadamente O(W×H) — lineal respecto al número de píxeles — pero con una constante que depende de la complejidad de la imagen (número de regiones de color).

8. Flujo de Datos Completo

[USUARIO] → Cargar imagen (File / Drag & Drop / Ctrl+V)
■
■■■ imageLoader.js → HTMLImageElement → handleImageLoaded()
■ ■■■ preprocessor.js → renderSource() → Canvas 2D con filtros
■ ■■■ sourceCanvas (panel izquierdo: vista previa)
■
■■■ Click "Vectorizar" → handleVectorize()
■ ■■■ ctx.getImageData() → ImageData { data: Uint8ClampedArray }
■ ■■■ worker.postMessage({ imageData, options }, [imageData.data.buffer])
■ ■ ■■■ [TRANSFERIDO] Buffer movido al worker sin copia
■ ■■■ _repaintCanvas() → redibuja desde HTMLImageElement original
■
■■■ [WORKER THREAD]
■ ■■■ preprocessor (filtros sobre los datos de imagen)
■ ■■■ vtracerBridge.vtrace(imageData, vtracerParams)

■ ■ ■■■ VTracer WASM → SVG string (splines/polygons/pixels)
■ ■■■ svgo.optimize(svgString) → SVG optimizado
■ ■■■ postMessage({ type:'result', svgCode, stats })
■
■■■ [MAIN THREAD] _onWorkerMessage({ type:'result' })
■■■ displayVector(svgContainer, svgCode)
■■■ _showStats(nodes, paths, timeMs)
■■■ _createWorker() ← Worker fresco para próxima operación
■■■ _updateUI() → habilitar exportación

[EXPORTACIÓN]
SVG: Blob([svgCode], {type:'image/svg+xml'}) → download
PNG: SVG → Image → Canvas(escala) → toBlob('image/png') → download
EPS: worker.postMessage({type:'export-eps', svgCode}) → worker genera EPS → download

9. Conclusiones

Free Vector Image demuestra el estado del arte en aplicaciones web de procesamiento gráfico intensivo. La combinación de WebAssembly (Rust → WASM) para el algoritmo de vectorización, Web Workers para el desacoplamiento de la UI, y Transferable Objects para paso de buffers sin copia, produce una experiencia que rivaliza con aplicaciones de escritorio como Adobe Illustrator o Inkscape en su funcionalidad de vectorización.

El patrón arquitectónico de "Worker fresco" es particularmente notable: al recrear el Worker después de cada operación, se evita la acumulación de memoria en el heap de WASM (que no tiene garbage collection automático en el sentido tradicional). Este es un aprendizaje práctico importante para cualquier aplicación web que utilice WebAssembly de larga duración.

La inclusión de ImageTracer.js como fallback cuando VTracer no está disponible demuestra un principio de diseño robusto: la herramienta nunca falla completamente. Si el navegador no soporta WASM, si la carga del módulo falla, o si ocurre un crash durante la vectorización, el usuario aún puede obtener un resultado funcional (aunque de menor calidad) mediante el motor JavaScript puro.

— Fin del Documento 06 — Free Vector Image —