

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAÍSO – CHILE



“CM-CHESSBOARD-2: OPTIMIZACIÓN DEL COMPONENTE CM-CHESSBOARD PARA CHESSMINDAI”

SEBASTIÁN IGNACIO TORREALBA CATALÁN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Raquel Pezoa
Profesor Co-guía: Gabriel Carmona
Profesor Correferente: Nelson Díaz

Agosto - 2026



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de Postgrado

Título del trabajo: CM-CHESSBOARD-2: OPTIMIZACIÓN DEL COMPONENTE CM-CHESSBOARD PARA CHESSMINDAI

Nombre del candidato(a): Sebastián Ignacio Torrealba Catalán

Carrera / Grado: Ingeniería Civil Informática

Campus: Campus Casa Central **Departamento:** Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Raquel Pezoa Rivera, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3.- EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar una opción**):

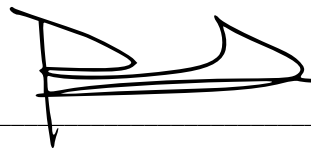
☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

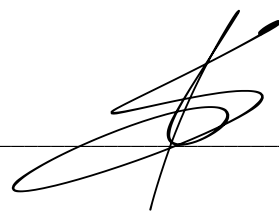
4.- FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 24 julio 2026

Firma: 

Estudiante o Candidato(a): Sebastián Torrealba

Fecha: 24 julio 2026 **Firma:** 

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

DEDICATORIA

A mi familia: a mi madre Gloria, mi padre Cristian y mi hermano Benjamín; a mis abuelos y a quienes no pudieron acompañarme en este proceso, pero siempre confiaron en mí.

Y especialmente a mi yo niño, que soñaba con ser ingeniero.

AGRADECIMIENTOS

En primer lugar, quiero agradecer a la Universidad Técnica Federico Santa María y al Departamento de Informática, por la formación que recibí durante estos años y por las oportunidades que me brindaron para desarrollarme tanto académica como personalmente.

Agradezco a mi profesora guía, Raquel Pezoa, por su orientación, apoyo y disposición durante el desarrollo de este trabajo, y también por la confianza que depositó en mí al permitirme ser su ayudante en varios ramos de la carrera. A los estudiantes que tuve durante esos años, gracias también: enseñar fue una de las experiencias que más me hizo crecer.

Agradezco también a mi profesor co-guía, Gabriel Carmona, quien ha revisado mi trabajo en varias ocasiones y me ha acompañado incluso desde antes de empezar la carrera. Ha sido un mentor que me apoyó en muchos momentos, entre ellos siendo mi coach en programación competitiva, y estuvo siempre cerca de mi proceso formativo.

A ChessMindAI, por recibirme y acompañarme durante todo este proceso. Más allá del espacio para llevar adelante mi memoria, encontré ahí un equipo que se interesó genuinamente por mi trabajo, que me escuchó cuando algo no funcionaba y que celebró conmigo cuando las cosas servían. Gracias por haber hecho de esta etapa una experiencia mucho mejor.

A mis compañeros de programación competitiva, Carlos Lagos y Abner Vidal, quienes caminaron a mi lado durante mucho tiempo en el desafío de mejorar como programador e ingeniero. Me empujaron a dar lo mejor de mí e hicieron que el proceso de la carrera fuera mucho más divertido.

A mis amigos de la carrera (ellos saben quiénes son), por hacer de mi vida universitaria algo mucho mejor de lo que esperaba. Compartimos estudios, risas, frustraciones y logros, y en cada uno de esos momentos me ayudaron a crecer también como persona.

A mi familia, que estuvo conmigo durante todo el proceso. A mi madre Gloria, a mi padre Cristian, a mi hermano Benjamín y a mis abuelos, por ayudarme y comprenderme cuando me sentía cansado o agobiado por las adversidades de la carrera. Cada uno me sostuvo a su manera, y sin ellos esto habría sido mucho más difícil.

Un agradecimiento especial a todas las personas que han sido parte de mi proceso formativo desde que era niño, durante la enseñanza básica y media. En particular, a mis tías de educación parvularia, que me cuidaron y guiaron cuando era muy pequeño.

A quienes estuvieron junto a mí hasta el final de este proceso, y también a quienes, sin poder estar presentes, siempre creyeron en mí.

Finalmente, gracias a todas las personas que en algún momento me dieron una palabra de aliento, un consejo o una ayuda: este logro también es de ustedes. Y gracias también a mí mismo, por no rendirme en el camino. La gratitud es algo que llevo siempre conmigo, y por eso hoy solo puedo decir: gracias.

RESUMEN

Resumen—ChessMindAI es una plataforma de enseñanza de ajedrez que utiliza cm-chessboard como componente central de tablero. La versión original de la librería presentaba tres limitaciones: ausencia de tipos estáticos, un sistema de extensiones limitado para las necesidades de ChessMindAI y un motor de animaciones serializado que acumulaba latencia bajo navegación rápida. Las alternativas existentes (Chessground, chessboard.js) no eran viables por licencia o mantenimiento.

Este trabajo documenta cm-chessboard-2, el fork que sustituye al componente original. Migra a TypeScript preservando la API pública, amplía el sistema de extensiones, rediseña el motor de animaciones con cancelación cooperativa basada en tokens y composite layers, optimiza el rendering SVG bajo redimensionamiento, reduce la presión sobre el garbage collector durante el drag y agrega una extensión que anima transiciones entre posiciones arbitrarias resolviendo el emparejamiento óptimo de piezas con el algoritmo Húngaro.

Para validar el resultado se construyó un harness de benchmark reproducible sobre 200 iteraciones por suite en Google Chrome de escritorio; el estudio de percepción con usuarios se propone como trabajo futuro. Sobre las ocho suites comparables, cm-chessboard-2 resulta entre 1,2 y 14,7 veces más rápido que el componente original en las operaciones intensivas, con un speedup de 10,89× en el escenario de carga sostenida.

Palabras Clave—cm-chessboard; TypeScript; animaciones web; rendimiento; benchmark

ABSTRACT

Abstract—ChessMindAI is a chess teaching platform that uses cm-chessboard as the core board component. The original library had three limitations: lack of static types, an extension system that fell short of ChessMindAI's needs, and a serialised animation engine that accumulated latency under fast navigation. Existing alternatives (Chessground, chessboard.js) were not viable due to licence or maintenance.

This work documents cm-chessboard-2, the fork that replaces the original component. It migrates to TypeScript preserving the public API, expands the extension system, rewrites the animation engine with token-based cooperative cancellation and composite layers, optimises the SVG rendering pipeline under resizing, reduces garbage-collector pressure during drag, and adds an extension that animates transitions between arbitrary positions by solving the optimal piece matching with the Hungarian algorithm.

To validate the result, a reproducible benchmark harness was built over 200 iterations per suite on desktop Google Chrome; user perception studies are left as future work. Across the eight comparable suites, cm-chessboard-2 is between 1.2 and 14.7 times faster than the original component on intensive operations, with a 10.89× speedup on the sustained-load scenario.

Keywords—cm-chessboard; TypeScript; web animations; performance; benchmark

GLOSARIO

Siglas

API: Application Programming Interface. Conjunto de operaciones que un componente expone para ser consumidas por código cliente.

CDN: Content Delivery Network. Red de servidores distribuidos geográficamente que sirven recursos estáticos con baja latencia.

CSS: Cascading Style Sheets. Lenguaje de estilos que controla la presentación visual de documentos web.

DI: Departamento de Informática.

DOM: Document Object Model. Representación en árbol que el navegador construye a partir del HTML; es la interfaz programática para leer y modificar el contenido de una página.

DPR: Device Pixel Ratio. Razón entre píxeles físicos del dispositivo y píxeles lógicos del navegador; un DPR mayor que uno indica pantalla de alta densidad.

FEN: Forsyth-Edwards Notation. Notación estándar para representar una posición de ajedrez en una sola cadena de texto; codifica la ubicación de las piezas, el turno, los enroques disponibles y la casilla de captura al paso.

FIFO: First-In, First-Out. Disciplina de cola en la que los elementos se procesan en el mismo orden en que fueron encolados.

FPS: Frames per second. Cuadros por segundo; medida de fluidez de animaciones.

GC: Garbage Collector. Componente del motor de JavaScript que libera automáticamente la memoria ocupada por objetos que ya no son referenciados.

GPL: GNU General Public License. Licencia de software de código abierto que exige que cualquier proyecto derivado o que integre el código licenciado se distribuya bajo los mismos términos, incluyendo la publicación del código fuente completo.

GPU: Graphics Processing Unit. Unidad de procesamiento gráfico del computador; los navegadores delegan en ella el trabajo del compositor para acelerar la presentación de capas.

HTML: HyperText Markup Language. Lenguaje de marcado con que se estructuran los documentos web.

HTTP: HyperText Transfer Protocol. Protocolo de aplicación usado para servir y consumir recursos web.

IDE: Integrated Development Environment. Entorno de desarrollo integrado.

JIT: Just-In-Time. Compilación que el motor de JavaScript realiza en tiempo de ejecución sobre rutas de código frecuentes para producir variantes nativas más rápidas que la interpretación inicial.

JSON: JavaScript Object Notation. Formato textual ligero para serializar datos estructurados, ampliamente usado para intercambio entre componentes web.

MIT: Massachusetts Institute of Technology. Nombre de la licencia de software de código abierto permisiva que permite uso, modificación y distribución libre.

MVC: Model-View-Controller. Patrón arquitectónico que separa la lógica de negocio (Model), la presentación (View) y la coordinación entre ambas (Controller).

RAF: RequestAnimationFrame. API del navegador que permite programar actualizaciones visuales sincronizadas con el ciclo de renderizado, a aproximadamente 60 veces por segundo.

SHA: Secure Hash Algorithm. En el contexto de Git, identificador hexadecimal único que identifica un commit específico en el historial del repositorio.

SVG: Scalable Vector Graphics. Formato de gráficos vectoriales basado en XML, escalable sin pérdida de calidad e integrable directamente en el DOM.

URL: Uniform Resource Locator. Cadena que identifica y permite localizar un recurso accesible por red.

UTFSM: Universidad Técnica Federico Santa María.

Términos técnicos

AbortController: API estándar del navegador que expone una señal compartida para abortar de forma cooperativa una operación asíncrona en curso, como una petición de red o una animación.

Adaptador: patrón de diseño que envuelve una interfaz existente y expone otra distinta, permitiendo que componentes incompatibles colaboren. En el harness, cada librería evaluada se accede a través de un adaptador que normaliza su API a un contrato uniforme.

Backend: componente de una aplicación web que se ejecuta en el servidor. Gestiona la lógica de negocio, la base de datos y expone una API al frontend.

Benchmark: experimento controlado que ejecuta un conjunto fijo de operaciones sobre uno o varios sistemas y mide su comportamiento bajo métricas comparables.

Build: artefacto producido por el proceso de compilación y empaquetado de un proyecto, listo para distribución o despliegue.

Bundle: archivo o conjunto de archivos resultante del proceso de empaquetado del código fuente para su distribución o despliegue en producción.

Callback: función pasada como argumento a otra función para ser invocada en un momento posterior, típicamente al completarse una operación o al ocurrir un evento.

Camino de aumento: en algoritmos sobre grafos bipartitos, secuencia alternante de aristas emparejadas y no emparejadas que conecta dos vértices libres y cuyo recorrido aumenta en uno el tamaño del emparejamiento.

Cancelación cooperativa: política de terminación anticipada de una tarea asíncrona en la que el productor publica una señal observable y los consumidores la consultan en puntos seguros para abortar su trabajo, en lugar de ser interrumpidos forzosamente desde fuera.

Churn de DOM: volumen de mutaciones sobre el árbol del DOM inducido por una operación. Un churn alto indica que el navegador realiza más trabajo de reconciliación y repintado.

Coalescencia: agrupación de varios eventos o invocaciones contiguos en una sola operación efectiva. En el contexto del rendering, evita ejecutar trabajo redundante cuando llegan múltiples notificaciones en el mismo frame.

Codebase: cuerpo completo de código fuente que conforma un proyecto o producto de software.

Composite layer: capa dedicada que el navegador crea para ciertos elementos con el fin de aislar su renderizado del resto del documento. Animar un elemento en su propia capa no provoca repintes de los elementos circundantes.

Compositor: subsistema del navegador que combina las composite layers ya pintadas para producir el frame final que se muestra en pantalla, generalmente delegando el trabajo a la GPU.

Costo reducido: en el algoritmo Húngaro, valor que resulta de restar a cada costo original los potenciales duales asociados a su fila y columna. El algoritmo mantiene los costos reducidos no negativos y construye el emparejamiento sobre las aristas de costo reducido cero.

Drag-and-drop: interacción que permite al usuario arrastrar un elemento de la interfaz con el puntero y soltarlo en otro lugar.

Emparejamiento: en teoría de grafos, subconjunto de aristas de un grafo en el que ningún vértice aparece en más de una arista. En un grafo bipartito asocia a lo sumo un elemento de un lado con un elemento del otro.

Fallback: comportamiento o ruta alternativa que se activa cuando la opción preferida no está disponible en el entorno de ejecución.

Feature: funcionalidad concreta que un producto de software ofrece a sus usuarios.

First contentful paint: instante en que el navegador pinta el primer contenido visible de la página tras la carga; métrica habitual de tiempo de respuesta percibido por el usuario.

Flag: argumento o bandera que altera el comportamiento de un programa o del navegador en tiempo de invocación.

Fork: copia independiente de un repositorio de código abierto que se mantiene y evoluciona de forma separada al original. Permite introducir cambios sin alterar el proyecto upstream.

Frontend: componente de una aplicación web que se ejecuta en el navegador del usuario. Gestiona la interfaz visual y la interacción.

Grafo bipartito: grafo cuyos vértices se particionan en dos conjuntos disjuntos y cuyas aristas conectan exclusivamente vértices de conjuntos distintos. El grafo bipartito completo es aquel en el que existe una arista entre cada par de vértices de lados opuestos.

Harness: entorno de pruebas o medición que orquesta la ejecución automatizada de benchmarks o tests, registrando métricas bajo condiciones controladas y reproducibles.

Headless: modo de ejecución de un navegador sin interfaz gráfica visible, controlado programáticamente. Es habitual en pruebas automatizadas que no requieren interacción humana.

Heap: región de memoria gestionada por el motor de JavaScript donde se almacenan los objetos creados en tiempo de ejecución; su tamaño usado y total son indicadores de la huella de memoria del programa.

Hook: punto de extensión que una librería expone para que código externo inyecte comportamiento sin modificar el núcleo.

Algoritmo Húngaro: método polinomial para resolver el problema de asignación lineal sobre un grafo bipartito completo, también conocido como algoritmo de Kuhn-Munkres. Devuelve un emparejamiento de costo agregado mínimo en complejidad cúbica en el tamaño del problema.

Hot path: ruta de código que se ejecuta con alta frecuencia durante la operación normal de un programa. Las optimizaciones en el hot path tienen mayor impacto en el rendimiento global.

Layout thrashing: degradación de rendimiento causada por ciclos repetidos de lectura y escritura del DOM que obligan al navegador a recalcular el layout de la página en cada iteración.

Layer promotion: mecanismo del navegador que mueve un elemento a su propia composite layer, generalmente activado por propiedades CSS como `will-change: transform`.

Microkernel: patrón arquitectónico que define un núcleo mínimo de funcionalidades y permite extender el sistema mediante plugins registrables, sin modificar el core.

MutationObserver: API del navegador que notifica al código cuando se producen mutaciones sobre el árbol del DOM, como nodos añadidos, eliminados o cambios en atributos.

npm: gestor de paquetes por defecto del ecosistema de Node.js, usado para declarar dependencias y exponer puntos de entrada de un paquete a través del archivo `package.json`.

Object pooling: patrón de diseño que reutiliza instancias de objetos en lugar de crearlas y destruirlas repetidamente, reduciendo la presión sobre el garbage collector en hot paths.

Observer: patrón de diseño en el que un objeto (sujeto) mantiene una lista de dependientes (observadores) y los notifica automáticamente cuando su estado cambia. Separa el emisor de eventos de sus consumidores sin acoplamiento directo.

p95: percentil 95 de una distribución de medidas; el valor por debajo del cual cae el 95% de las observaciones. Útil para caracterizar el comportamiento en los casos peores frecuentes sin contaminarse con los extremos raros.

performance.memory: propiedad no estándar disponible en navegadores basados en Chromium que expone una estimación del heap usado y total del proceso de la página. Su precisión aumenta cuando el navegador se inicia con la flag que habilita información detallada de memoria.

performance.now: método estándar del navegador que devuelve un timestamp monótono de alta resolución, expresado en milisegundos con fracciones. Es la base usual para medir duraciones de operaciones en código de aplicación.

Pipeline de rendering: secuencia de etapas que el navegador ejecuta para llevar el DOM a píxeles en pantalla: cálculo de estilos, layout, paint y composición.

Plugin: módulo que se conecta al núcleo de un sistema para añadir o modificar comportamiento sin tocar el código del núcleo.

Polling: técnica que consiste en consultar periódicamente el estado de un recurso o condición en lugar de esperar una notificación asíncrona. Genera trabajo continuo incluso cuando no hay cambios.

Potenciales duales: en el algoritmo Húngaro, vectores de valores asociados a las filas y columnas de la matriz de costos cuya suma para cada par no excede el costo correspondiente. Se actualizan iterativamente para mantener la no negatividad de los costos reducidos y guían la construcción del emparejamiento óptimo.

Preload: hint del navegador, expresado como `<link rel="preload">`, que instruye a descargar un recurso crítico en paralelo con el resto de la carga de la página, antes de que el parser del HTML lo solicite.

Problema de asignación: problema combinatorio que busca emparejar elementos de dos conjuntos minimizando el costo agregado de las parejas, sujeto a que cada elemento se empareje a lo sumo una vez. Su versión sobre un grafo bipartito completo con costos arbitrarios se conoce como problema de asignación lineal.

Pull request: propuesta formal de cambios sobre un repositorio Git que solicita su revisión y posterior incorporación a una rama de destino.

Race condition: situación en la que el resultado de una operación depende del orden no determinístico en que se ejecutan múltiples tareas concurrentes. En el contexto de animaciones, puede producir estados visuales incorrectos.

Refactoring: modificación de la estructura interna del código que preserva su comportamiento observable, con el objetivo de mejorar su legibilidad, mantenibilidad o desempeño.

ResizeObserver: API del navegador que notifica al código cuando el tamaño de un elemento del DOM cambia, sin necesidad de polling ni de escuchar el evento global `window.resize`.

Runner: componente del harness que orquesta la ejecución de las suites, dispara las iteraciones, aplica el warmup y registra las métricas resultantes.

Runtime: entorno en el que un programa se ejecuta una vez compilado y cargado en memoria, distinto del entorno de compilación.

Singleton: patrón de diseño que garantiza que una clase tiene una única instancia en todo el proceso y proporciona un punto de acceso global a ella.

Speedup: razón entre el tiempo de una implementación de referencia y el de una implementación bajo evaluación sobre la misma carga. Valores mayores que uno indican aceleración respecto a la referencia.

Sprite: imagen o archivo que agrupa múltiples gráficos (en este caso, las piezas de ajedrez) en un solo recurso. Reduce el número de peticiones de red y simplifica la gestión de recursos visuales.

Strategy: patrón de diseño que define una familia de algoritmos, los encapsula en clases separadas e intercambiables, y permite seleccionar el algoritmo en tiempo de ejecución sin modificar el código que lo usa.

Sub-path exports: mecanismo del campo `exports` en `package.json` que permite exponer rutas específicas de un paquete npm como puntos de entrada independientes, facilitando el tree-shaking y la importación granular de módulos.

Suite: agrupación de casos relacionados dentro de un benchmark o de una batería de pruebas; cada suite cubre un aspecto o escenario específico del sistema bajo medición.

Throttling: limitación deliberada de un recurso por debajo de su capacidad máxima. En procesadores describe la reducción de frecuencia por temperatura o energía; en redes, la moderación del ancho de banda.

Tipado gradual: estrategia de incorporación de tipos estáticos en un lenguaje dinámico que permite coexistencia de código tipado y no tipado en el mismo proyecto.

Token de cancelación: objeto compartido entre un productor y los consumidores de una operación asíncrona que permite al productor señalar, mediante una bandera observable, que la operación debe abortarse cooperativamente.

Tradeoff: compromiso de ingeniería en el que ganar una propiedad implica sacrificar parcialmente otra; la decisión consiste en elegir qué se prioriza.

Tree-shaking: proceso de eliminación de código no utilizado durante el empaquetado. Requiere que los módulos exporten sus funcionalidades de forma granular para que el empaquetador identifique qué puede omitir.

Upstream: repositorio original del que se deriva un fork. Los cambios del upstream pueden incorporarse al fork mediante merges, pero no ocurren automáticamente.

Vecino más próximo: heurística que asigna a cada elemento de un conjunto el elemento más cercano de otro conjunto bajo una métrica dada. Es rápida (lineal en una pasada local) pero no garantiza un emparejamiento globalmente óptimo, a diferencia del algoritmo Húngaro.

Viewport: región visible del navegador en la que se renderiza el contenido de la página, medida en píxeles lógicos.

Warmup: fase inicial de un benchmark cuyas corridas se descartan del agregado estadístico, con el fin de amortizar el costo del JIT y de las primeras reservas de memoria antes de medir el régimen estable.

Web Animations API: API del navegador que permite crear y controlar animaciones directamente desde JavaScript mediante `element.animate()`, con soporte para cancelación, composición y control de estado.

ÍNDICE DE CONTENIDOS

RESUMEN	iii
ABSTRACT	iii
GLOSARIO	iv
ÍNDICE DE FIGURAS	xii
ÍNDICE DE TABLAS	xii
CAPÍTULO 1: INTRODUCCIÓN	1
1.1 OBJETIVOS	2
1.2 ALCANCE	2
1.3 ESTRUCTURA DEL DOCUMENTO	3
CAPÍTULO 2: DEFINICIÓN DEL PROBLEMA	5
2.1 CONTEXTO Y ESCENARIOS DE USO	5
2.2 ACTORES	6
2.3 LIMITACIONES DE LA VERSIÓN ORIGINAL	6
2.4 ALTERNATIVAS EVALUADAS	7
2.5 REQUERIMIENTOS	8
2.6 CRITERIOS DE ÉXITO	8
2.7 ALCANCE Y DELIMITACIONES	9
CAPÍTULO 3: MARCO CONCEPTUAL	10
3.1 ARQUITECTURA MVC Y SISTEMA DE PLUGINS	10
3.2 TIPADO GRADUAL EN JAVASCRIPT	10
3.3 PIPELINE DE RENDERIZADO DEL NAVEGADOR	11
3.4 OBJECT POOLING Y COSTO DEL GARBAGE COLLECTOR	12
3.5 PROBLEMA DE ASIGNACIÓN Y ALGORITMO HÚNGARO	12
3.6 LIBRERÍAS COMPARADAS	13
CAPÍTULO 4: DISEÑO E IMPLEMENTACIÓN	15
4.1 DESCRIPCIÓN DEL CASO	15
4.2 MIGRACIÓN A TYPESCRIPT	15
4.3 ARQUITECTURA MVC Y SISTEMA DE EXTENSIONES	16
4.4 ANIMATIONMANAGER	17
4.5 RENDERING SVG RESPONSIVO	19
4.5.1 OPTIMIZACIONES DE LAYOUT Y REFLOW	19
4.5.2 CAPAS LAZY Y PRECARGA DE SPRITES	19
4.6 POOL DE PIEZAS ARRASTRADAS	20
4.7 MÁQUINA DE ESTADOS DE INPUT	21
4.8 MORPHTRANSITION	22

4.8.1	PROBLEMA DE ASIGNACIÓN APLICADO AL TABLERO	23
4.8.2	ALGORITMO HÚNGARO	23
CAPÍTULO 5: EXPERIMENTACIÓN		25
5.1	HERRAMIENTAS Y ENTORNO	25
5.2	ESTRATEGIA Y MEDICIÓN	25
5.3	EJECUCIÓN	26
5.4	PARAMETRIZACIÓN	26
5.5	SUITES Y MÉTRICAS	27
5.6	CRITERIOS DE EVALUACIÓN	28
CAPÍTULO 6: RESULTADOS		29
6.1	ENTORNO DE MEDICIÓN	29
6.2	PROTOCOLO DE EJECUCIÓN	30
6.3	SUITERENDER	30
6.4	SUISESET-POSITION	31
6.5	SUITEMOVE-INSTANT	31
6.6	SUITEBULK-MOVES	31
6.7	SUITEORIENTATION	32
6.8	SUITERESIZE	32
6.9	SUITEDESTROY-CREATE	33
6.10	SPEEDUP DE CM-CHESSBOARD-2 FRENTE A LA VERSIÓN ORIGINAL	33
6.11	MUTACIONES DEL DOM POR SUITE	34
6.12	DISCUSIÓN	34
6.12.1	LECTURA CONJUNTA DE LAS SUITES	34
6.12.2	ATRIBUCIÓN A DECISIONES TÉCNICAS	35
6.12.3	COMPARATIVA CUALITATIVA CON LAS ALTERNATIVAS	36
6.13	LIMITACIONES DE LA EVALUACIÓN	36
CAPÍTULO 7: CONCLUSIONES		38
7.1	SÍNTESIS DE RESULTADOS	38
7.2	CONTRIBUCIONES	39
7.3	LIMITACIONES DEL TRABAJO	39
7.4	TRABAJO FUTURO	40
ANEXOS		41
7.5	ANEXO A:PROTOCOLO DE MEDICIÓN	41
7.5.1	ESTRUCTURA DEL HARNESS	41
7.5.2	EJECUCIÓN	41
7.5.3	PARAMETRIZACIÓN DEL RUNNER	41
7.6	ANEXO B:ENTORNO DE MEDICIÓN	43
7.7	ANEXO C:MAPA DE DECISIONES TÉCNICAS	43
REFERENCIAS BIBLIOGRÁFICAS		45

ÍNDICE DE FIGURAS

Figura 1	Tablero renderizado por cm-chessboard-2 en una posición de ejemplo.	5
Figura 2	Arquitectura de cm-chessboard-2: el núcleo MVC expone ocho puntos de extensión a los que se suscriben los plugins.	16
Figura 3	Ejemplo de transición de MorphTransition entre dos posiciones arbitrarias: cada pieza de la posición origen se anima hacia su contraparte en la destino según el emparejamiento óptimo (algoritmo Húngaro).	22

ÍNDICE DE TABLAS

Tabla 1	Características de las librerías de tableros de ajedrez web evaluadas.	14
Tabla 2	Matriz de costos para un subgrupo de tres peones blancos en a2, b2, c2 con destinos a3, b3, c3. Cada celda es el cuadrado de la distancia entre la casilla origen y la casilla destino.	24
Tabla 3	Entorno bajo el que se ejecutó el harness de benchmark.	29
Tabla 4	Tiempo promedio por iteración en la suite render.	30
Tabla 5	Tiempo promedio por iteración en la suite set-position.	31
Tabla 6	Tiempo promedio por iteración en la suite move-instant.	31
Tabla 7	Tiempo total promedio del replay en la suite bulk-moves.	32
Tabla 8	Tiempo promedio por iteración en la suite orientation.	32
Tabla 9	Tiempo promedio por iteración en la suite resize.	33
Tabla 10	Tiempo promedio por iteración en la suite destroy-create.	33
Tabla 11	Speedup promedio de cm-chessboard-2 frente al componente original por suite.	34
Tabla 12	Mutaciones promedio del DOM acumuladas por suite y adaptador.	34
Tabla 13	Parámetros del runner del harness.	42
Tabla 14	Entorno completo de medición.	43
Tabla 15	Decisiones técnicas de cm-chessboard-2 y dónde se discuten en la memoria. .	44

CAPÍTULO 1

INTRODUCCIÓN

ChessMindAI (ChessMindAI, 2026) es una plataforma de enseñanza de ajedrez cuyo frontend está desarrollado en Vue 3 y cuyo backend corre sobre Django. El componente central de la interfaz es el tablero interactivo: la pieza de software con la que el estudiante resuelve puzzles, reproduce partidas y navega posiciones. Para ese rol, el frontend utilizaba `cm-chessboard` (Haack, 2024), una librería de código abierto mantenida por Stefan Haack bajo licencia MIT, que renderiza tableros SVG en el navegador con soporte para drag-and-drop y animaciones de movimiento.

A medida que ChessMindAI incorporó features educativas más exigentes (secuencias de puzzles encadenados, deshacer y rehacer movidas, transiciones visuales entre posiciones arbitrarias, capas de anotación sobre el tablero) la versión original comenzó a mostrar tres limitaciones concretas.

La primera limitación es de rendimiento. El motor de animaciones de la versión original serializa los cambios de posición: cada nueva solicitud espera a que termine la animación anterior. En secuencias rápidas de puzzles o de undo/redo, esa serialización produce latencia acumulada visible: el tablero arrastra animaciones obsoletas antes de mostrar la posición pedida. La ausencia de composite layer promotion implicaba además que cada frame de animación disparaba repintes completos del tablero.

La segunda limitación se origina en la ausencia de tipos estáticos. El código base estaba escrito en JavaScript puro; integrar la librería en un proyecto TypeScript obligaba a ignorar errores de tipos en los puntos de contacto, reducía la asistencia del IDE y dificultaba el refactoring. Esta ausencia de tipos era una deuda técnica que crecía con el tamaño del codebase de ChessMindAI.

La tercera limitación es de extensibilidad. El sistema de extensiones de la versión original, con ocho puntos de extensión, cubre casos generales como el resaltado de casillas o la promoción de peón, pero varias features que ChessMindAI necesita (las transiciones entre posiciones arbitrarias, la reducción de reservas de memoria durante el drag) exceden ese contrato y exigen intervenir en el motor de animación y en el subsistema de input, no a nivel de un plugin.

Ante estas limitaciones, se evaluaron alternativas. Chessground (Lichess, 2024), el tablero de Lichess, ofrece buen rendimiento pero se distribuye bajo licencia GPL-3.0, que obliga a publicar el código fuente completo de cualquier proyecto que lo integre; incompatible con ChessMindAI. `chessboard.js` (Oakman, 2019), el componente más veterano del ecosistema, carece de TypeScript y no tiene mantenimiento activo. Contribuir los cambios requeridos al proyecto original de `cm-chessboard` tampoco resultaba viable: las modificaciones necesarias eran demasiado específicas del contexto educativo de ChessMindAI y comprometían la generalidad de la librería original.

La decisión adoptada fue crear cm-chessboard-2, un fork de cm-chessboard con control total sobre la evolución del componente. Este trabajo documenta ese fork: las decisiones de diseño que lo conforman, su implementación y los resultados obtenidos con un benchmark comparado frente al componente original y las principales alternativas.

1.1. OBJETIVOS

El objetivo general del trabajo es mejorar y extender cm-chessboard para ChessMindAI mediante una migración completa a TypeScript, un rediseño arquitectónico bajo el patrón MVC con sistema de extensiones, un conjunto de optimizaciones de animación y rendering medidas con benchmark comparado, y la documentación de las decisiones técnicas que conforman el fork.

Los objetivos específicos son:

- OE1. Migrar el código base de JavaScript a TypeScript preservando la API pública de la librería, con anotaciones de tipos sobre la API pública y sin pérdidas de funcionalidad respecto a la versión original.
- OE2. Rediseñar la arquitectura interna del componente bajo el patrón MVC con un sistema de extensiones, de modo que features de ChessMindAI puedan agregarse sin modificar el núcleo.
- OE3. Implementar optimizaciones de animación y rendering SVG que eliminen la latencia acumulada de la cola FIFO de la versión original, reduzcan repintes mediante composite layer promotion y mejoren el comportamiento bajo redimensionamiento y secuencias rápidas de movimientos.
- OE4. Construir un harness de benchmark reproducible que compare cm-chessboard-2 con la versión original, Chessground y chessboard.js en métricas de tiempo de operación, estabilidad de FPS y uso de memoria, ejecutable en Google Chrome de escritorio.
- OE5. Aplicar las optimizaciones complementarias de cm-chessboard-2 (pool de piezas arrastradas, máquina de estados de input, transición morfológica entre posiciones) y evaluar su impacto cualitativo o cuantitativo según aplique.

1.2. ALCANCE

El trabajo abarca siete familias principales de cambios técnicos en cm-chessboard-2:

- Migración completa a TypeScript.
- Arquitectura MVC y sistema de plugins.
- Rediseño del motor de animaciones.
- Optimizaciones de rendering SVG.
- Pool de piezas arrastradas.
- Máquina de estados de input.

- Harness de benchmark comparativo.

A esas familias se suma una sección breve sobre una extensión que anima transiciones visuales entre dos posiciones arbitrarias del tablero, sin asumir que entre ellas exista una jugada legal; se documenta como contribución funcional sin medición cuantitativa propia. Dos cambios de menor alcance (sub-path exports en package . json y un refactor de limpieza del código heredado) se mencionan donde son relevantes pero no constituyen secciones propias. El trabajo cubre también el benchmark comparado, ejecutado en Google Chrome de escritorio, y el análisis del historial de desarrollo y código fuente.

El componente está orientado a navegadores de escritorio. El funcionamiento en navegadores móviles se obtiene por consecuencia de funcionar bien en escritorio, pero no es objetivo explícito de optimización en este trabajo; su evaluación cuantitativa se propone como trabajo futuro.

Quedan fuera del alcance de este trabajo y se proponen como trabajo futuro los siguientes aspectos:

- Validación con usuarios: los resultados del benchmark son métricas técnicas de rendimiento, no medidas de experiencia de usuario. Un estudio A/B queda propuesto como trabajo futuro.
- Evaluación cuantitativa en motores y dispositivos móviles: las mediciones se realizaron únicamente en Google Chrome de escritorio; replicar el harness en otros motores o en dispositivos móviles queda como trabajo futuro.

1.3. ESTRUCTURA DEL DOCUMENTO

El capítulo 2, Definición del problema, caracteriza con detalle las limitaciones de la versión original para ChessMindAI, el contexto en que opera el componente, los actores involucrados y los requerimientos y restricciones que guiaron el trabajo.

El capítulo 3, Marco conceptual, presenta los conceptos técnicos que sustentan las decisiones de cm-chessboard-2 (arquitectura MVC y sistema de plugins, tipado gradual, pipeline de renderizado del navegador, object pooling) y compara las librerías de tableros existentes.

El capítulo 4, Diseño e implementación, describe cada una de las familias de cambios principales de cm-chessboard-2 con referencia al código fuente.

El capítulo 5, Experimentación, describe el harness de benchmark construido para evaluar el componente: herramientas y entorno, estrategia de medición, parametrización, suites y métricas, y criterios de evaluación.

El capítulo 6, Resultados, presenta las mediciones del benchmark y la comparativa cuantitativa frente al componente original y a las alternativas evaluadas.

El capítulo 7, Conclusiones, sintetiza los resultados, lista las contribuciones, discute las limitaciones del trabajo y propone líneas de trabajo futuro.

CAPÍTULO 2

DEFINICIÓN DEL PROBLEMA

Este capítulo caracteriza con detalle el problema que motiva la creación de cm-chessboard-2: la inadecuación del componente cm-chessboard en su versión original para los escenarios educativos de ChessMindAI. La introducción presentó las tres limitaciones principales en términos generales; aquí se especifica el contexto de uso, los actores afectados, las limitaciones técnicas concretas, las alternativas evaluadas y los requerimientos y criterios de éxito que se derivan.

2.1. CONTEXTO Y ESCENARIOS DE USO

ChessMindAI es una plataforma de enseñanza de ajedrez cuyo frontend está implementado en Vue 3 y cuyo backend corre sobre Django. El tablero es la pieza central del frontend (Figura 1): a través de él, el estudiante resuelve puzzles, reproduce partidas y navega posiciones.

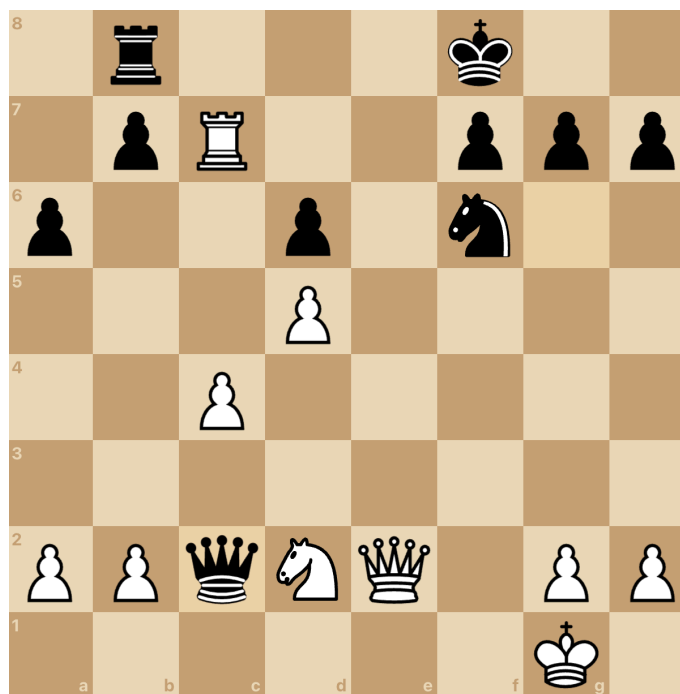


Figura 1: Tablero renderizado por cm-chessboard-2 en una posición de ejemplo.

Tres escenarios condicionan los requerimientos sobre el componente:

- Secuencias de puzzles encadenados. Tras resolver un puzzle, el sistema presenta el siguiente de inmediato. Cada transición implica varias movidas animadas ejecutadas en sucesión rápida. La fluidez percibida del enlace entre puzzles es parte de la experiencia educativa: una pausa entre puzzle y puzzle rompe el ritmo de la sesión.

- Navegación con deshacer y rehacer. Durante el análisis de una partida, el estudiante avanza y retrocede entre posiciones. Cada paso puede generar varias movidas animadas. Cuando el usuario pulsa repetidamente la flecha de avance, el tablero recibe solicitudes de cambio de posición más rápido de lo que puede animar cada movimiento.
- Transiciones entre posiciones arbitrarias. Algunos ejercicios presentan dos posiciones distantes (por ejemplo, una posición inicial y la posición final de una secuencia de ejemplo) y requieren una transición visual que ayude al estudiante a entender qué piezas se movieron y a dónde. Esta operación no corresponde a una sola movida de ajedrez: implica calcular un emparejamiento entre piezas presentes en ambas posiciones y animarlas en paralelo.

2.2. ACTORES

Tres grupos de actores se ven afectados por la elección del componente de tablero:

- Estudiantes de ChessMindAI. Son los usuarios finales. Perciben la calidad del componente como suavidad de animación, ausencia de saltos visuales y respuesta inmediata a sus acciones. No tienen visibilidad de la implementación.
- Equipo de desarrollo de ChessMindAI. Mantiene el frontend en TypeScript y añade funcionalidades educativas que dependen del tablero. La calidad de los tipos del componente y la extensibilidad de su API determinan el costo de incorporar nuevas features.
- Mantenedor del proyecto original. Stefan Haack mantiene cm-chessboard como librería de propósito general. Su ritmo de evolución y sus criterios de aceptación de cambios siguen prioridades distintas a las de ChessMindAI; cambios específicos del contexto educativo no son siempre apropiados para incorporarse al proyecto original.

2.3. LIMITACIONES DE LA VERSIÓN ORIGINAL

cm-chessboard cumple en su versión original su rol como tablero de ajedrez web genérico, pero presenta tres limitaciones concretas para los escenarios descritos:

1. **Latencia acumulada por encolado FIFO de animaciones.** La versión original encola las animaciones en una cola FIFO: cada nueva solicitud de cambio de posición se serializa hasta que la animación anterior termina. En los escenarios de uso educativo descritos (secuencias rápidas de puzzles, navegación con flechas, undo y redo continuos), esta política produce latencia acumulada visible. El tablero arrastra animaciones obsoletas antes de mostrar la posición que el usuario pidió, porque no hay forma de cancelar una animación en curso para superponer la siguiente.

2. **Ausencia de tipos estáticos.** El código original está escrito en JavaScript puro, sin anotaciones de tipos. La ausencia de tipos no es la limitación en sí, pero es la causa de un conjunto de restricciones concretas al integrar la librería en un codebase TypeScript: hay que aceptar errores de tipo del compilador, mantener archivos de declaración (.d.ts) escritos a mano que pueden desincronizarse del código real, o ignorar la asistencia del IDE en los puntos de contacto con la librería. El análisis empírico de Gao et al. (Gao et al., 2017) mostró que el 15% de los bugs en proyectos JavaScript populares son detectables estáticamente con TypeScript. La integración sin tipos renuncia a esa red de seguridad para una parte considerable del frontend.
3. **Extensibilidad para casos de uso educativos.** La versión original provee un sistema de extensiones con ocho puntos de extensión, suficiente para casos generales (resaltado de casillas, promoción de peón, marcadores). Sin embargo, varias features que ChessMindAI necesita exceden ese contrato: las transiciones entre posiciones arbitrarias requieren coordinar varias animaciones simultáneas a nivel del motor de animación, no a nivel de un plugin; la reducción de reservas de memoria durante el drag requiere intervenir en el subsistema de input. Estas features no caben dentro de los puntos de extensión existentes sin modificar el núcleo del componente.

2.4. ALTERNATIVAS EVALUADAS

Cuatro alternativas se evaluaron antes de optar por el fork:

- Contribuir los cambios al proyecto original. Las modificaciones requeridas son específicas del contexto educativo y del codebase TypeScript de ChessMindAI; comprometerían la generalidad de la librería original y exigirían rondas de discusión con el mantenedor cuyos plazos no son compatibles con el desarrollo del producto.
- Adoptar Chessground. El tablero de Lichess (Lichess, 2024) es rápido y maduro, pero se distribuye bajo licencia GPL-3.0. Esta licencia obliga a publicar el código fuente completo de cualquier proyecto que la integre, incompatible con el modelo de licenciamiento de ChessMindAI.
- Adoptar chessboard.js. El componente veterano del ecosistema (Oakman, 2019) carece de TypeScript, depende de jQuery y no tiene mantenimiento activo desde 2019. Adoptar una librería abandonada habría sustituido una limitación por otra mayor.
- Reescribir desde cero. Un tablero nuevo daría libertad total de diseño pero exigiría rehacer un volumen significativo de funcionalidad ya resuelta por la versión original (renderizado SVG, sistema de coordenadas, manejo de FEN, conjunto de piezas vectoriales). El costo de oportunidad es alto y la deuda inicial considerable.

cm-chessboard-2, el fork de cm-chessboard, combina ventajas de las alternativas anteriores: hereda una base funcional madura, preserva la licencia MIT y permite intervenir el código en cualquier punto necesario sin condicionarse al ritmo del proyecto original.

2.5. REQUERIMIENTOS

Los requerimientos del componente derivan de las limitaciones identificadas y del contexto de uso:

- R1. Preservar la API pública. El frontend de ChessMindAI ya consume la API original. La migración a cm-chessboard-2 debe ser transparente para el código cliente: los mismos métodos, con la misma semántica.
- R2. Tipado estático. El componente debe estar escrito en TypeScript, con tipos que cubran la API pública.
- R3. Animaciones cancelables. El motor de animación debe permitir que una nueva solicitud de cambio de posición interrumpa la animación previa y comience la nueva, eliminando la latencia acumulada de la cola FIFO.
- R4. Composite layers durante la animación. Las piezas en movimiento deben promoverse a su propia capa de compositing del navegador para evitar repintes innecesarios del resto del tablero.
- R5. Soporte para transiciones entre posiciones arbitrarias. El componente debe ofrecer un mecanismo para animar el paso entre dos posiciones que no se conectan por una sola movida de ajedrez.
- R6. Reducción de presión sobre el GC durante interacciones. El drag y otras interacciones continuas no deben generar reservas de memoria por evento que aumenten la presión sobre el recolector de basura.
- R7. Licencia MIT. cm-chessboard-2 debe preservar la licencia original para no introducir nuevas restricciones sobre ChessMindAI.

2.6. CRITERIOS DE ÉXITO

El éxito del trabajo se evalúa en dos dimensiones:

- Cuantitativos: comparativa de cm-chessboard-2 frente al componente original, Chessground y chessboard.js en un harness de benchmark reproducible. Las métricas a contrastar son tiempo de operación de animaciones y transiciones, estabilidad de FPS bajo cargas sostenidas, conteo de mutaciones del DOM y delta de huella de memoria por escenario.
- Cualitativos: capacidad demostrada de añadir extensiones nuevas (capas de anotación, transición entre posiciones) sin modificar el núcleo, preservación de la suite de ejemplos original y migración del código cliente del frontend de ChessMindAI con cambios mínimos.

2.7. ALCANCE Y DELIMITACIONES

Dentro del alcance del trabajo:

- Las siete familias de cambios técnicos descritas en la introducción.
- Una extensión para transiciones visuales entre dos posiciones arbitrarias del tablero, sin asumir una jugada legal entre ellas.
- El harness de benchmark comparativo y los resultados obtenidos en Google Chrome de escritorio.

El componente está orientado a navegadores de escritorio. El funcionamiento en navegadores móviles se obtiene por consecuencia de funcionar bien en escritorio, pero no es objetivo de optimización en este trabajo; su evaluación cuantitativa se propone como trabajo futuro.

Fuera del alcance de este trabajo:

- Estudio de percepción con usuarios. Las métricas son técnicas, no de experiencia de usuario. Un estudio A/B con estudiantes queda propuesto como trabajo futuro.
- Evaluación cuantitativa multi-motor y móvil. La medición se realiza únicamente en Google Chrome de escritorio. Replicar el harness en otros motores o en dispositivos móviles queda como trabajo futuro.
- Reemplazo del modelo de datos del tablero. El sistema de coordenadas, el parsing de FEN y la API de piezas se preservan de la versión original; el trabajo se concentra en arquitectura, animación, rendering, input y herramientas de evaluación.

CAPÍTULO 3

MARCO CONCEPTUAL

Este capítulo presenta los conceptos técnicos que fundamentan las decisiones de diseño de cm-chessboard-2. Cada sección corresponde a una familia de problemas que el trabajo aborda: la organización arquitectónica del código, el sistema de tipos, el rendimiento de animaciones en el navegador y la gestión de memoria durante la interacción. El capítulo cierra con una comparación de las librerías de tableros existentes que motivó la elección de cm-chessboard como base del fork.

3.1. ARQUITECTURA MVC Y SISTEMA DE PLUGINS

El patrón Model-View-Controller (MVC) organiza una aplicación en tres componentes con responsabilidades distintas. El Model gestiona el estado de la aplicación, la View presenta ese estado al usuario y el Controller coordina las interacciones entre ambos. Propuesto por Reenskaug en 1979 para el entorno Smalltalk en Xerox PARC (Reenskaug, 1979), MVC se convirtió en el patrón de referencia para interfaces de usuario interactivas. cm-chessboard adopta una variante simplificada de este patrón: Model y View tienen clases dedicadas, mientras que el rol de Controller queda distribuido entre la clase principal del componente y subsistemas especializados como el motor de animaciones y el de entrada.

El patrón Microkernel, documentado por Buschmann et al. (Buschmann et al., 1996), extiende esta separación definiendo un núcleo mínimo de funcionalidades estables rodeado de plugins que añaden comportamientos sin modificar el core. La comunicación entre el núcleo y los plugins se realiza a través de puntos de extensión predefinidos: el core invoca callbacks registrados por los plugins en momentos específicos del ciclo de vida de la aplicación. El resultado es un sistema donde el núcleo no necesita conocer los plugins que lo extienden.

cm-chessboard implementa esta arquitectura con ocho puntos de extensión (positionChanged, boardChanged, moveInput, animation, entre otros) donde los plugins registran callbacks. cm-chessboard-2 hereda esta estructura y agrega nuevas extensiones sin modificar el core. Los patrones Observer y Strategy (Gamma et al., 1994) aparecen como mecanismos auxiliares: Observer para notificar a los plugins de cambios de estado, Strategy para intercambiar comportamientos de entrada (drag vs. click).

3.2. TIPADO GRADUAL EN JAVASCRIPT

TypeScript es un superconjunto de JavaScript que agrega anotaciones de tipos estáticos compiladas a JavaScript puro. Se clasifica como un sistema de tipado gradual: permite

mezclar código con tipos explícitos y código sin anotar en el mismo proyecto, adoptando los tipos de forma incremental (Siek & Taha, 2006). Esta propiedad es clave para migraciones de codebases existentes, donde reescribir todo el código con tipos completos de golpe no es viable.

La motivación empírica para la migración está documentada por Gao et al. (Gao et al., 2017): en un análisis de 400 repositorios JavaScript populares de GitHub, el 15% de los bugs reportados eran detectables estáticamente con TypeScript. Para un codebase en crecimiento activo como el de ChessMindAI, ese margen es significativo.

La migración de cm-chessboard-2 adopta una configuración no estricta del compilador: `strict`, `strictNullChecks`, `strictPropertyInitialization` y `noImplicitAny` quedan deshabilitados. Esta postura permisiva reduce los errores de compilación durante la migración incremental al admitir patrones del código JavaScript original que no anotan explícitamente la ausencia de valor ni los tipos de cada parámetro. El tradeoff es una cobertura de tipos incompleta: el IDE mejora y muchos errores se detectan en compilación, pero los relacionados con nulos o argumentos sin anotar no se capturan estáticamente.

3.3. PIPELINE DE RENDERIZADO DEL NAVEGADOR

El navegador convierte el DOM y los estilos CSS en píxeles en pantalla mediante un pipeline de tres fases (Lewis & Irish, 2018):

1. Layout (reflow): calcula la geometría de cada elemento (posición, tamaño). Se activa cuando cambia el tamaño o la posición de un elemento.
2. Paint: rasteriza el contenido visual. Se activa cuando cambian propiedades visuales como color o borde.
3. Composite: combina las capas rasterizadas y las envía a la GPU.

Las animaciones más eficientes operan únicamente en la fase de composite, evitando layout y paint en cada frame. Propiedades como `transform` y `opacity` son compositables: el navegador puede animarlas en la GPU sin recalcular geometría ni repintar. La propiedad CSS `will-change: transform` instruye al navegador para que promueva un elemento a su propia composite layer (layer promotion) antes de que comience la animación, eliminando el costo del primer frame (MDN Web Docs, 2025).

La Web Animations API (W3C, 2023) permite crear y controlar animaciones desde JavaScript mediante `element.animate()`, con acceso completo al estado de cada animación (pausar, cancelar, completar) y soporte para composición de efectos. A diferencia de las transiciones CSS, la Web Animations API permite cancelar una animación en curso y conocer su progreso exacto, lo que resulta esencial para los escenarios de cancelación del `AnimationManager`.

`ResizeObserver` (W3C, 2021) notifica al código cuando el tamaño de un elemento cambia, sin necesidad de polling. Sin embargo, puede disparar múltiples callbacks en un mismo frame durante operaciones de redimensionado continuo. Combinar `ResizeObserver`

con `requestAnimationFrame` coalesce esos callbacks a una actualización por frame, manteniendo la consistencia visual sin trabajo innecesario.

3.4. OBJECT POOLING Y COSTO DEL GARBAGE COLLECTOR

JavaScript gestiona la memoria automáticamente mediante un garbage collector (GC). El GC libera objetos que ya no tienen referencias activas, pero la recolección tiene un costo no determinístico: puede introducir pausas de varios milisegundos en momentos arbitrarios, visibles como caídas de FPS durante interacciones continuas.

Blackburn et al. (Blackburn et al., 2004) documentaron que el impacto real del GC depende principalmente de la tasa de reserva de memoria: si el código crea y descarta muchos objetos en poco tiempo, la presión sobre el GC aumenta proporcionalmente, independientemente de la eficiencia del algoritmo de recolección. En un hot path a 60 fps, cada reserva de memoria innecesaria acorta el tiempo hasta la siguiente pausa del GC.

El patrón Object Pool aborda este problema pre-ubicando un conjunto fijo de objetos y reutilizándolos en lugar de crear nuevos en cada operación. El costo de reserva de memoria se paga una sola vez al iniciar la aplicación. En el contexto del drag-and-drop de piezas de ajedrez, cada evento de puntero puede generar reservas de memoria si el elemento SVG de la pieza arrastrada se crea y destruye en cada ciclo. El `DraggablePiecePool` de `cm-chessboard-2` implementa un pool mínimo de un único elemento SVG reutilizable, dimensionado para el caso de uso habitual en el que solo una pieza se arrastra a la vez; en la situación excepcional de dos tableros con drag simultáneo, el pool emite un elemento temporal adicional que se destruye al liberarse.

3.5. PROBLEMA DE ASIGNACIÓN Y ALGORITMO HÚNGARO

Cuando una operación visual requiere mover múltiples piezas en paralelo desde una posición origen a una posición destino sin asumir una jugada legal entre ellas, el problema de decidir qué pieza origen se anima hacia qué pieza destino corresponde a un problema clásico de optimización combinatoria: el problema de asignación lineal sobre un grafo bipartito completo. Esta formulación es la base teórica del algoritmo de emparejamiento que utiliza `cm-chessboard-2` para las transiciones entre posiciones arbitrarias, descritas en el capítulo de Diseño e implementación.

Sea un conjunto finito de orígenes S y un conjunto finito de destinos D , con una función de costo $c : S \times D \rightarrow \mathbb{R}^+$ que asigna a cada par (s, d) un costo no negativo. El problema de asignación lineal busca un emparejamiento μ entre orígenes y destinos que minimice la suma agregada de costos. Si $|S| = |D|$ el emparejamiento es total y biyectivo. Si los cardinales difieren, el problema se reduce al caso cuadrado completando con filas o

columnas ficticias de costo neutro: los elementos emparejados con ficticios quedan sin asignación real.

El algoritmo Húngaro de Kuhn (Kuhn, 1955) fue el primer método combinatorio en tiempo polinomial para resolver el problema, en complejidad $O(n^4)$. Munkres (Munkres, 1957) refinó el algoritmo a $O(n^3)$ sobre matrices cuadradas de tamaño $n \times n$, con extensiones que cuando se preserva la precondition $n \leq m$ alcanzan $O(n^2m)$. La idea central es construir potenciales duales sobre filas y columnas que mantienen invariante la no negatividad de los costos reducidos, y aumentar iterativamente un emparejamiento parcial recorriendo caminos alternantes sobre las aristas de costo reducido cero hasta cubrir todos los elementos del conjunto menor. El algoritmo entrega un emparejamiento de costo agregado mínimo demostrablemente óptimo.

3.6. LIBRERÍAS COMPARADAS

Al momento de decidir el fork, las principales alternativas para implementar un tablero de ajedrez web eran tres.

cm-chessboard (Haack, 2024) (upstream) es la base de cm-chessboard-2. Renderiza en SVG, tiene un sistema de extensiones con ocho puntos de extensión, está escrito en JavaScript y se distribuye bajo licencia MIT. Su limitación para ChessMindAI era la ausencia de tipos estáticos y la latencia acumulada del motor de animaciones, que encola las animaciones y obliga al usuario a esperar a que cada transición termine antes de aplicar la siguiente en secuencias rápidas de movidas.

Chessground (Lichess, 2024) es el tablero de Lichess, probado en producción en uno de los sitios de ajedrez más visitados del mundo. Su rendimiento es competitivo y su API es estable, pero se distribuye bajo licencia GPL-3.0: cualquier proyecto que lo integre debe publicar su código fuente completo bajo la misma licencia, lo que lo hace incompatible con ChessMindAI.

chessboard.js (Oakman, 2019) fue durante años la referencia del ecosistema. Su API es simple y bien documentada, pero depende de jQuery, carece de TypeScript y no tiene mantenimiento activo desde 2019.

La Tabla 1 sintetiza las características de las cuatro librerías evaluadas: la licencia, el modo de renderizado, el soporte de TypeScript, las dependencias externas, el sistema de plugins, la cancelación de animaciones, el morph entre posiciones arbitrarias y el estado de mantenimiento.

Característica	cm-chessboard-2	cm-chessboard	Chessground	chessboard.js
Licencia	MIT	MIT	GPL-3.0	MIT
Renderizado	SVG	SVG	SVG	DOM
TypeScript	Sí	No	Sí	No
Dependencias externas	Ninguna	Ninguna	Ninguna	jQuery
Sistema de plugins	Sí	Sí	Limitado	No
Animaciones cancelables	Sí	No	Parcial	No
Morph entre posiciones	Sí	No	Sí	No
Mantenimiento activo	Sí	Sí	Sí	No

Tabla 1: Características de las librerías de tableros de ajedrez web evaluadas.

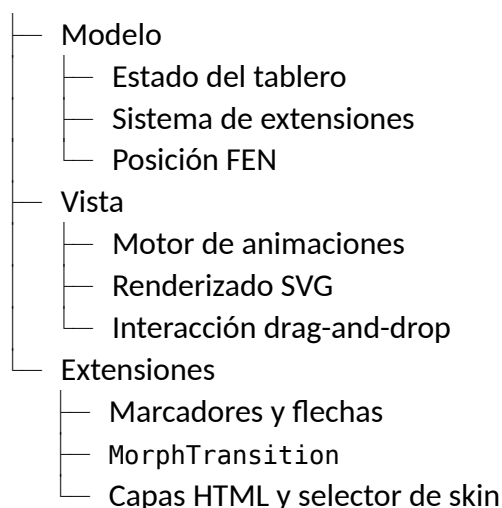
CAPÍTULO 4

DISEÑO E IMPLEMENTACIÓN

Este capítulo describe las decisiones técnicas de `cm-chessboard-2`, el fork de `cm-chessboard` integrado en ChessMindAI. `cm-chessboard-2` parte de la versión 8.x del componente original (Haack, 2024) y se mantiene en el repositorio de ChessMindAI bajo licencia MIT. Las decisiones se presentan agrupadas por dimensión: el sistema de tipos, el núcleo y su sistema de extensiones, el motor de animaciones, el pipeline de rendering SVG, dos optimizaciones de interacción (pool de piezas arrastradas y máquina de estados de input) y la extensión `MorphTransition` para transiciones entre posiciones arbitrarias.

4.1. DESCRIPCIÓN DEL CASO

`cm-chessboard-2` conserva la organización del componente original y agrega extensiones propias. Su arquitectura se divide en tres capas, como muestra el Listado 1.



Listado 1: Arquitectura de `cm-chessboard-2` por responsabilidad.

`cm-chessboard-2` introduce cambios en cuatro dimensiones: el sistema de tipos (migración a TypeScript), la arquitectura (extensiones adicionales), el motor de animaciones (rediseño del `AnimationManager`) y el rendering (optimizaciones de SVG y responsive).

4.2. MIGRACIÓN A TYPESCRIPT

La versión original está escrita en JavaScript puro. `cm-chessboard-2` migra todo el código fuente a TypeScript, generando declaraciones de tipos para que los consumidores del paquete obtengan asistencia del IDE en sus propios proyectos.

El compilador se configura con ES2020 como target. cm-chessboard-2 desactiva los checks estrictos (strict, strictNullChecks, strictPropertyInitialization, noImplicitAny) para que la migración acepte patrones heredados del JavaScript original sin reescribirlos, en línea con el enfoque de tipado gradual (Siek & Taha, 2006) y con el tradeoff documentado en el marco conceptual.

4.3. ARQUITECTURA MVC Y SISTEMA DE EXTENSIONES

cm-chessboard-2 hereda la arquitectura MVC del componente original y amplía su sistema de plugins. El núcleo define ocho puntos de extensión a los que los plugins suscriben callbacks: positionChanged, boardChanged, moveInputToggled, moveInput, beforeRedrawBoard, afterRedrawBoard, animation y destroy.

Cada plugin registra callbacks en los puntos que le interesan. El núcleo invoca esos callbacks en el momento correspondiente sin saber qué plugins están registrados. Esta separación sigue el patrón Microkernel (Buschmann et al., 1996): el core es estable y los plugins son independientes entre sí. El patrón Observer (Gamma et al., 1994) describe el mecanismo de notificación subyacente. La Figura 2 resume esta organización: el núcleo MVC y los puntos de extensión a los que se suscriben los plugins.

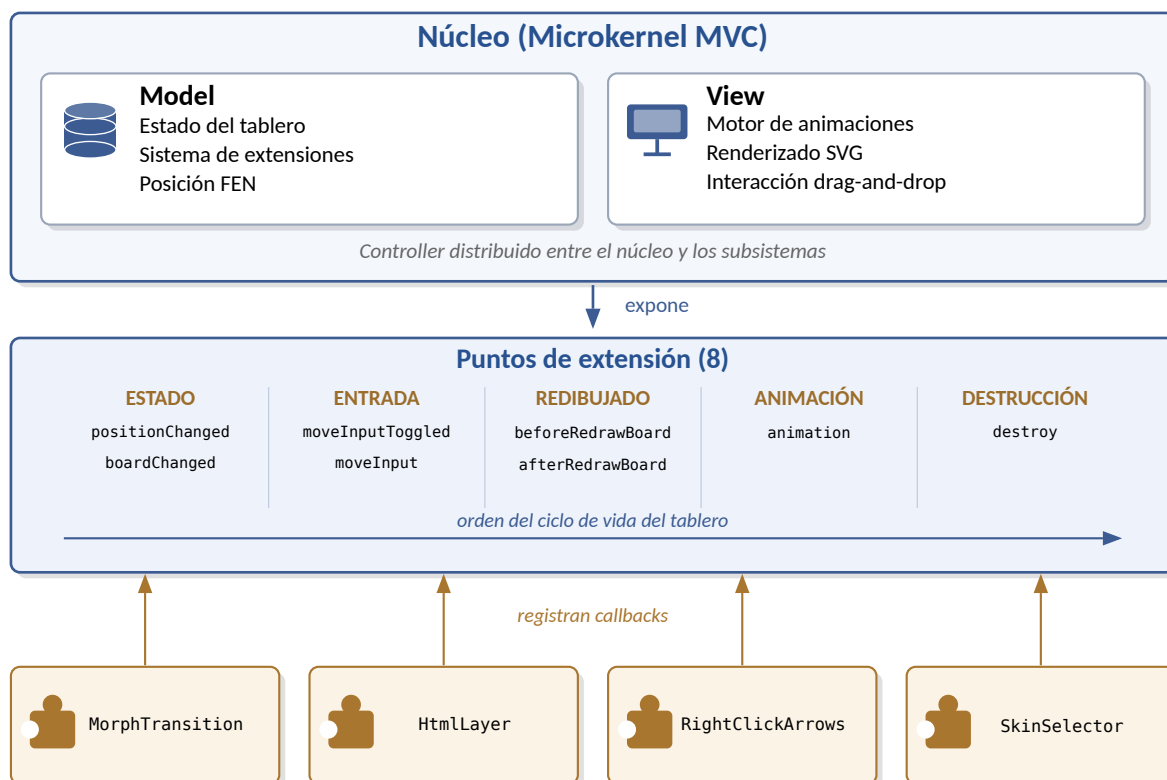


Figura 2: Arquitectura de cm-chessboard-2: el núcleo MVC expone ocho puntos de extensión a los que se suscriben los plugins.

cm-chessboard-2 agrega varias extensiones que el componente original no incluye: `MorphTransition` para transiciones entre posiciones arbitrarias, `HtmlLayer` para superponer contenido HTML sobre el tablero, `RightClickArrows` para flechas de análisis y `SkinSelector` para cambiar la apariencia en tiempo de ejecución.

4.4. ANIMATIONMANAGER

Como se anticipó en la introducción, el `AnimationManager` del componente original encola los cambios de posición y acumula latencia en secuencias rápidas. cm-chessboard-2 sustituye esa política de espera por una de cancelación cooperativa, sostenida por tres mecanismos que se describen a continuación.

Token de cancelación. Cada animación recibe un token al iniciarse. Todos los callbacks de esa animación verifican la validez del token antes de ejecutarse. Si llega una nueva solicitud de cambio de posición, el token de la animación anterior se invalida. El Algoritmo 1 resume el patrón.

```
1 procedure SetPosition(newFen)
2   Invalidar token actual
3   token ← nuevo Token
4   for each cambio in calcularCambios(newFen) do
5     if token.esValido() then
6       | aplicarCambio(cambio, token)
7     end if
8   end for
9 end procedure
```

Algoritmo 1: Implementación de `SetPosition(newFen)`. Parámetros: posición destino en notación FEN. Objetivo: aplicar la nueva posición invalidando la animación previa para evitar latencia acumulada. Retorno: ninguno (efectos sobre el DOM mediante callbacks asíncronos).

Cuando el usuario avanza al siguiente puzzle antes de que la animación anterior termine, el token se invalida y los callbacks pendientes retornan sin modificar el DOM. El resultado visual es un corte limpio hacia la nueva posición, sin artefactos. El patrón es análogo al `AbortController` del estándar web.

Web Animations API y composite layers. La versión original usa transiciones CSS. cm-chessboard-2 migra a la Web Animations API (W3C, 2023), que permite controlar cada animación individualmente: pausar, cancelar y conocer su progreso exacto. Esto habilita el token de cancelación: al invalidar un token, se cancela cada animación activa asociada.

Para aislar el costo de repintado, cm-chessboard-2 instruye al navegador para que promueva cada pieza SVG a su propia composite layer mediante la propiedad CSS `will-change: transform` (MDN Web Docs, 2025). Cuando una pieza se anima, el navegador transforma su capa en la GPU sin repintar el resto del tablero. El costo es memoria adicional por cada capa promovida; con 32 piezas como máximo, el impacto es despreciable frente a la ganancia en fluidez (Lewis & Irish, 2018).

Detección de cambios y secuenciación. Para determinar qué piezas se mueven, aparecen o desaparecen entre dos posiciones, el `AnimationManager` implementa una detección en $O(n)$: agrupa las piezas por tipo y asigna cada pieza de la posición origen a la más cercana del mismo tipo en la posición destino mediante vecino más próximo. El resultado es un conjunto de cambios tipados: `move`, `appear` y `disappear`.

Un error de secuenciación en implementaciones naive ocurre cuando un `appear` en una casilla destino se procesa antes que el `move` que tenía esa casilla como origen: el `appear` sobrescribe la referencia al elemento visual, dejando el `move` sin nada que animar. cm-chessboard-2 corrige este problema garantizando que todos los `move` se procesen antes que los `appear`; los `disappear` se acumulan al inicio y se ejecutan tras completarse las dos pasadas anteriores. El Algoritmo 2 materializa este orden.

```
1 procedure AplicarCambios(cambios, token)
2   for each cambio where cambio.tipo = MOVER do
3     | animarMovimiento(cambio, token)
4   end for
5   for each cambio where cambio.tipo = APARECER do
6     | animarAparicion(cambio, token)
7   end for
8   for each cambio where cambio.tipo = DESAPARECER do
9     | eliminarPieza(cambio)
10  end for
11 end procedure
```

Algoritmo 2: Implementación de `AplicarCambios(cambios, token)`. Parámetros: lista de cambios tipados entre dos posiciones y token de la animación en curso. Objetivo: aplicar los cambios en tres pasadas (mover, aparecer y, al final, desaparecer) para evitar que un `appear` sobrescriba la referencia visual de un `move` pendiente. Retorno: ninguno.

Al completar cada animación, las métricas de posición se recalculan con los valores actuales del DOM, garantizando que las piezas aterricen en la posición correcta incluso si el tablero fue redimensionado durante la animación.

4.5. RENDERING SVG RESPONSIVO

El componente de vista gestiona el renderizado SVG y la adaptación a cambios de tamaño del contenedor. La versión original recalcula las dimensiones del tablero en JavaScript en cada redimensionado. cm-chessboard-2 aplica varias optimizaciones para reducir el trabajo por frame y evitar layout thrashing.

4.5.1. OPTIMIZACIONES DE LAYOUT Y REFLOW

El contenedor del tablero mantiene su proporción cuadrada mediante la propiedad CSS nativa `aspect-ratio`, eliminando el frame de lag donde el ancho se actualiza por CSS pero la altura espera al siguiente ciclo de JavaScript.

Para detectar redimensionados, cm-chessboard-2 prioriza `ResizeObserver` (W3C, 2021) sobre el evento global `window.resize`, al que solo recurre como fallback en entornos donde `ResizeObserver` no está disponible. Durante operaciones de redimensionado continuo, `ResizeObserver` puede disparar múltiples veces por frame. Un `requestAnimationFrame` intermedio coalesce esos callbacks a una sola actualización, según se describe en el Algoritmo 3.

```
1 on elementoObservado cambia de tamaño do  
2   | Cancelar frame pendiente si existe  
3   | Programar nuevo frame: ejecutar handleResize()  
4 end on
```

Algoritmo 3: Implementación del handler del `ResizeObserver`. Parámetros: evento implícito disparado por el observador al cambiar el tamaño del contenedor. Objetivo: coalescer múltiples notificaciones del observador en una única ejecución por frame del navegador mediante `requestAnimationFrame`. Retorno: ninguno.

Las métricas de geometría del tablero se leen del DOM una sola vez por redimensionado y se guardan en caché. Los cálculos de posición subsiguientes usan esos valores cacheados en lugar de leer el DOM en cada frame, evitando layout thrashing.

4.5.2. CAPAS LAZY Y PRECARGA DE SPRITES

Las capas de extensión opcionales se crean únicamente la primera vez que se accede a ellas. Un tablero sin marcadores no agrega ningún elemento al DOM, como ilustra el Algoritmo 4.

```
1 function capaMarcadores()
2   if capa no inicializada then
3     capa ← crearGrupoSVG()
4     insertarAntesQueCapaPiezas(capa)
5   end if
6   return capa
7 end function
```

Algoritmo 4: Implementación de `capaMarcadores()`. Parámetros: ninguno. Objetivo: devolver el grupo SVG de la capa de marcadores, creándolo bajo demanda la primera vez que se accede e insertándolo en el orden DOM correcto. Retorno: el elemento SVG correspondiente a la capa de marcadores.

Para los sprites de las piezas, `cm-chessboard-2` inyecta un `<link rel="preload">` (W3C, 2022) tan pronto como se crea el tablero. Un conjunto interno evita duplicados cuando se instancian varios tableros. El `preload` instruye al navegador para descargar el sprite en paralelo con el resto de la carga de la página, eliminando el flash de piezas ausentes en el primer render.

4.6. POOL DE PIEZAS ARRASTRADAS

Durante el drag-and-drop, el componente original crea y destruye un elemento visual en cada operación. A 60 eventos de puntero por segundo, esa reserva de memoria aumenta la presión sobre el garbage collector (Blackburn et al., 2004). `cm-chessboard-2` implementa un pool mínimo con patrón Singleton (Gamma et al., 1994): una sola instancia compartida entre todos los tableros de la página. El Algoritmo 5 describe las dos operaciones del pool.

```
1 function Adquirir()
2   if pieza en uso then
3     | return elemento temporal nuevo
4   end if
5   if pieza no existe then
6     | pieza ← crearElementoSVG()
7   end if
8   marcar como en uso
9   return pieza
10 end function
11 function Liberar(elemento)
12   if elemento es la pieza del pool then
13     | limpiar contenido y transformación
14     | marcar como disponible
15   else
16     | destruir elemento temporal
17   end if
18 end function
```

Algoritmo 5: Implementación de Adquirir() y Liberar(elemento) del pool de pieza arrastrada. Adquirir() no recibe parámetros; objetivo: entregar el elemento SVG reutilizable del pool, creándolo perezosamente en la primera llamada o devolviendo un elemento temporal cuando el pool ya está ocupado; retorna el elemento listo para arrastrar. Liberar(elemento) recibe el SVG devuelto previamente por Adquirir(); objetivo: limpiar y marcar como disponible la pieza del pool, o eliminar del DOM el elemento temporal; no retorna valor.

El elemento SVG se crea la primera vez y se reutiliza en drags posteriores. Liberar limpia el contenido y el transform sin eliminar el elemento del DOM. Solo en el caso extremo de dos tableros con drag simultáneo se crea un elemento temporal adicional, que sí se destruye al completarse.

4.7. MÁQUINA DE ESTADOS DE INPUT

La interacción del usuario con las piezas (click, drag, cancelación) se gestiona mediante una máquina de estados finitos con ocho estados. La versión original maneja estos eventos con condicionales dispersos; cm-chessboard-2 centraliza el control de flujo.

Los eventos de entrada se clasifican en cinco tipos de cancelación explícita: segundo click sobre la misma pieza, click derecho, puntero fuera del tablero, pieza arrastrada de vuelta a su casilla origen, y selección de otra pieza antes de completar el movimiento. Cada razón

de cancelación se comunica al plugin de input registrado, que puede mostrar feedback visual diferenciado. En ChessMindAI, esto permite distinguir entre un movimiento ilegal y un cambio de intención del usuario.

4.8. MORPHTRANSITION

cm-chessboard-2 incluye una extensión visual, MorphTransition, que el componente original no proporciona. Su propósito es animar la transición entre dos posiciones arbitrarias sin asumir que entre ellas existe una jugada legal, lo que resulta útil para secuencias de demostración y recorridos estéticos entre ejercicios.

MorphTransition se integra como extensión del sistema de plugins descrito en la sección de arquitectura y agrega al tablero un método público `transitionTo(fen, opts)`. La diferencia con el `setPosition()` estándar es la semántica del emparejamiento. `setPosition()` solo anima las piezas cuya casilla cambia entre el FEN de origen y el de destino: las piezas que permanecen en la misma casilla son invisibles al algoritmo. `transitionTo()` toma en cambio todas las piezas presentes en origen y destino y resuelve un emparejamiento óptimo entre ellas. La Figura 3 ilustra una transición entre dos posiciones arbitrarias y el emparejamiento de piezas que la origina.

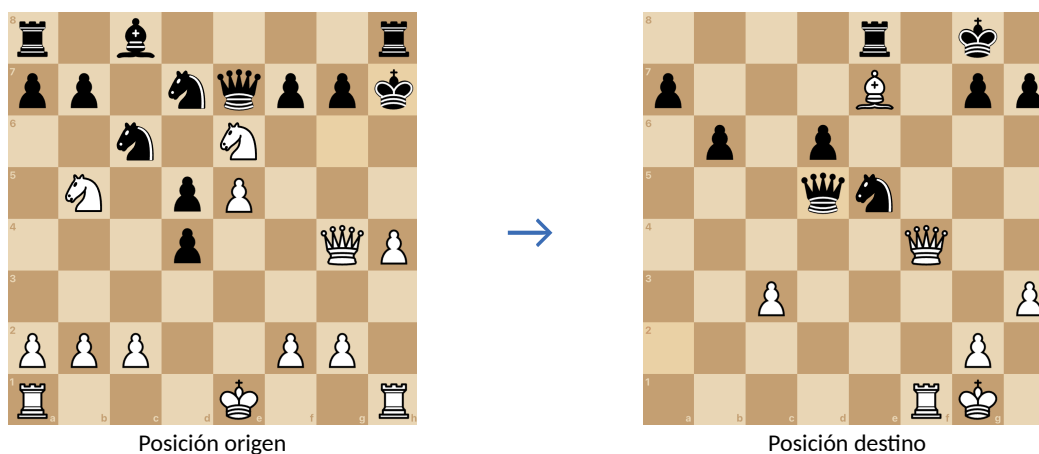


Figura 3: Ejemplo de transición de MorphTransition entre dos posiciones arbitrarias: cada pieza de la posición origen se anima hacia su contraparte en la destino según el emparejamiento óptimo (algoritmo Húngaro).

El problema visual que `transitionTo()` resuelve es: dado un conjunto de piezas en la posición origen y un conjunto en la posición destino, decidir qué pieza origen se anima hacia qué pieza destino de modo que el movimiento conjunto sea visualmente coherente y se minimice el desplazamiento total. Esta formulación se reduce al problema de asignación lineal presentado en el marco conceptual mediante dos observaciones. Primero, solo tiene sentido emparejar piezas equivalentes: un alfil blanco no puede sustituirse por un peón negro durante la transición. La extensión descompone entonces el problema global en subproblemas independientes, uno por cada combinación de color y tipo. Segundo, la

función de costo natural sobre cada subproblema es una medida geométrica de cuánto debe desplazarse cada pieza en el tablero. La extensión utiliza el cuadrado de la distancia euclidiana entre las casillas origen y destino dentro del tablero de 64 escaques. La asignación óptima sobre cada subproblema se obtiene aplicando el algoritmo Húngaro descrito en el marco conceptual, y se materializa como una secuencia de eventos *move*, *appear* y *disappear* que el motor de animación procesa con las garantías de cancelación cooperativa introducidas en la sección del motor de animaciones.

4.8.1. PROBLEMA DE ASIGNACIÓN APLICADO AL TABLERO

Sea un grupo de piezas idénticas en color y tipo, con un conjunto de casillas origen S y un conjunto de casillas destino D dentro del tablero de 64 escaques. Se busca un emparejamiento parcial $\mu : S' \rightarrow D'$ con $S' \subseteq S$ y $D' \subseteq D$ que minimice la distancia agregada

$$\sum_{s \in S'} c(s, \mu(s))$$

donde $c(s, d)$ es el cuadrado de la distancia euclidiana entre las casillas s y d del tablero. El cardinal del emparejamiento es $\min(|S|, |D|)$: las casillas de S no emparejadas producen *disappear*, las casillas de D no emparejadas producen *appear*, y las pareadas con $s \neq d$ producen *move*. Las pareadas con $s = d$ no generan evento.

La descomposición por grupo permite resolver el problema global como una colección de subproblemas independientes sin pérdida de optimalidad, ya que el emparejamiento solo tiene sentido entre piezas equivalentes (un alfil blanco no puede ocupar el lugar de un peón negro). Cada subproblema es una instancia del problema de asignación lineal sobre un grafo bipartito completo, cuya solución óptima existe y se puede calcular en tiempo polinomial.

4.8.2. ALGORITMO HÚNGARO

cm-chessboard-2 resuelve cada subproblema con el algoritmo Húngaro de Kuhn (Kuhn, 1955), en la variante refinada por Munkres (Munkres, 1957), con complejidad $O(n^2m)$ donde $n = |S| \leq |D| = m$. Cuando el grupo es rectangular en el otro sentido ($|S| > |D|$), se transpone el problema para preservar la precondition $n \leq m$ y las casillas de la dimensión mayor que queden sin asignación se reinterpretan como *disappear*. La implementación utiliza potenciales duales sobre filas y columnas y caminos de aumento de costo reducido, devolviendo una asignación de costo agregado mínimo dentro de cada grupo.

La aplicación concreta al fork procede en tres pasos. Primero, para cada subgrupo color-tipo se construye una matriz de costos rectangular C de tamaño $|S| \times |D|$, como la que ilustra la Tabla 2, donde C_{ij} es el cuadrado de la distancia euclidiana entre la casilla s_i y la casilla d_j

medida sobre el plano del tablero. Usar el cuadrado de la distancia (en lugar de la distancia) penaliza desproporcionadamente los emparejamientos largos: el algoritmo prefiere repartir varios movimientos cortos antes que uno solo desproporcionadamente largo, lo que se percibe como una animación más natural. Segundo, se invoca el algoritmo sobre C y se obtiene una asignación μ de costo mínimo. Tercero, la asignación se traduce a eventos: cada par $(s, \mu(s))$ con $s \neq \mu(s)$ produce un move; los pares con $s = \mu(s)$ no generan evento (la pieza permanece en su casilla); las casillas origen no asignadas producen disappear; las casillas destino no asignadas producen appear.

Costo	Destino a3	Destino b3	Destino c3
Origen a2	1	2	5
Origen b2	2	1	2
Origen c2	5	2	1

Tabla 2: Matriz de costos para un subgrupo de tres peones blancos en a2, b2, c2 con destinos a3, b3, c3. Cada celda es el cuadrado de la distancia entre la casilla origen y la casilla destino.

A modo de ilustración, la Tabla 2 describe un subgrupo simple con tres peones blancos en casillas $S = \{a2, b2, c2\}$ y destinos $D = \{a3, b3, c3\}$. La matriz 3×3 contiene los cuadrados de las distancias entre cada origen y cada destino. La asignación óptima $\mu(a2) = a3$, $\mu(b2) = b3$, $\mu(c2) = c3$ tiene costo agregado 3; cualquier otra permutación produce un costo mayor (por ejemplo, intercambiar los emparejamientos extremos eleva el costo a 11). En este caso los tres emparejamientos resultan en tres eventos move paralelos hacia la fila adyacente, traducción visualmente equivalente a un movimiento de los peones una casilla hacia adelante.

CAPÍTULO 5

EXPERIMENTACIÓN

Este capítulo describe el entorno, la estrategia y los criterios bajo los que se evaluó cm-chessboard-2. El harness de benchmark fue la última pieza del trabajo: se construyó después de estabilizar las decisiones técnicas del componente, lo que permitió parametrizarlo en función de las decisiones reales y no de hipótesis previas.

5.1. HERRAMIENTAS Y ENTORNO

El componente y el harness se construyeron con las siguientes herramientas:

- Lenguaje: TypeScript, con el compilador tsc configurado para emitir módulos ES2020.
- Gestor de paquetes: npm. cm-chessboard-2 no incorpora dependencias de runtime, preservando la política original de cero dependencias externas en producción; las únicas dependencias del paquete son herramientas de desarrollo (compilador TypeScript, framework de pruebas).
- Control de versiones: git, con cm-chessboard-2 mantenido como rama de desarrollo paralela al proyecto original para facilitar la incorporación selectiva de cambios aguas arriba cuando son relevantes.
- Navegador: Google Chrome de escritorio, tanto para desarrollo como para la ejecución del harness.
- Sistema operativo de medición: Linux de escritorio.

El detalle del equipo (CPU, GPU, memoria), del sistema operativo, del navegador y de las flags utilizadas durante la corrida reportada en el capítulo de Resultados se documenta en la Tabla 14 del Anexo B.

5.2. ESTRATEGIA Y MEDICIÓN

La evaluación cuantitativa se apoya en un harness dedicado que compara cm-chessboard-2 con tres alternativas representativas (cm-chessboard original, Chessground y chessboard.js) bajo la misma operación, el mismo entorno y la misma corrida. Cada librería se accede a través de un adaptador que expone una interfaz uniforme (setPosition, movePiece, setOrientation, resize, destroy); cada suite ejecuta su escenario a través de esos adaptadores y un runner común recoge las métricas.

Las directrices de Georges et al. (Georges et al., 2007) sobre evaluación rigurosa de rendimiento y la advertencia de Mytkowicz et al. (Mytkowicz et al., 2009) sobre distorsiones causadas por el entorno de medición guiaron la elección de los parámetros (warmup, iteraciones, estadísticos reportados). La lista de «benchmarking crimes» de Heiser (Heiser,

2010) se empleó como checklist informal de errores a evitar (entorno no documentado, medición de un solo trial, comparación entre versiones de hardware distintas, etc.).

El harness no usa un driver headless externo (no requiere puppeteer ni playwright). La ejecución se inicia desde la propia página del runner en el navegador, lo que simplifica la reproducibilidad a costa de la ejecución completamente desatendida.

5.3. EJECUCIÓN

El protocolo para reproducir una corrida es el siguiente:

1. Servir el directorio del harness con un servidor HTTP local (`bunx serve`).
2. Iniciar Google Chrome con las flags requeridas (`--enable-precise-memory-info` y `--js-flags="--expose-gc"`) para habilitar `performance.memory` con valores no aproximados y exponer el hook opcional de recolección de basura.
3. Abrir la URL del runner principal en el navegador y verificar que las cuatro librerías queden listadas como disponibles.
4. Configurar el número de iteraciones (200 en la corrida reportada) y seleccionar las suites a ejecutar; pulsar el control de inicio.
5. Esperar a que el runner reporte los estadísticos al terminar y exportar el resultado a un archivo JSON.

El bucle externo recorre las suites; el bucle interno recorre los cuatro adaptadores (`cm-chessboard-2`, `cm-chessboard`, `Chessground`, `chessboard.js`) en orden fijo. No hay recarga de la página entre adaptadores, lo que mantiene un único contexto de JavaScript para toda la corrida.

5.4. PARAMETRIZACIÓN

Los parámetros del runner están pensados para reducir el ruido de las primeras iteraciones y producir estadísticos comparables entre adaptadores:

- Warmup: las primeras 3 corridas de cada escenario se descartan, para amortizar el costo del JIT y las primeras reservas de memoria del navegador.
- Iteraciones de medición: 200 corridas por (suite, adaptador) tras el warmup, configurables desde la UI del runner. La corrida reportada combina 200 iteraciones por cada una de las ocho suites comparables y por cada uno de los cuatro adaptadores, totalizando $200 \times 8 \times 4 = 6.400$ medidas.
- Estadísticos reportados: media, mediana, mínimo, máximo, percentil 95 y desviación estándar de los tiempos observados.
- Recolector de basura: el runner expone un hook opcional para forzar recolecciones entre iteraciones cuando el navegador se inicia con la flag correspondiente, reduciendo la varianza inducida por pausas no deterministas del GC (Blackburn et al., 2004).

- Presupuesto de frame: el harness considera 8,3 ms como umbral por frame para el monitor de medición (120 Hz); los frames cuyo delta supera ese presupuesto se contabilizan aparte.

5.5. SUITES Y MÉTRICAS

El harness define once suites independientes, cada una orientada a un aspecto distinto del comportamiento del componente:

- `render`: construcción inicial del componente (creación del nodo SVG, montaje del set inicial de piezas, primer paint).
- `set-position` y `set-position-anim`: aplicación de una nueva posición sin y con animación.
- `move-instant` y `move-animated`: movimiento individual sin y con animación.
- `bulk-moves`: replay de la partida de la Ópera (Morphy vs. Duque de Brunswick y Conde Isouard, París 1858) en 33 medias jugadas aplicadas en sucesión rápida.
- `orientation`: inversión de la orientación del tablero.
- `resize`: cambio de las dimensiones del contenedor y refresco de las coordenadas.
- `destroy-create`: ciclo completo de destrucción y recreación del componente.
- `memory` y `sustained-memory`: estimación de la huella de memoria mediante `performance.memory`, en un escenario puntual y en uno sostenido.

Tres suites quedan excluidas de la comparativa numérica del capítulo de Resultados. Las dos variantes con animación (`set-position-anim`, `move-animated`) se excluyen porque las cuatro librerías definen la duración y el modelo de animación de manera estructuralmente distinta, lo que hace que los tiempos por iteración no sean comparables entre ellas. La suite `memory` también queda fuera de la comparativa de tiempos porque reporta una unidad distinta (kilobytes de heap usado en lugar de milisegundos), por lo que sus resultados se presentan aparte y no se agregan a la comparativa de tiempos del capítulo de Resultados. Quedan así ocho suites comparables. El efecto del rediseño del motor de animación se observa indirectamente en `bulk-moves`, donde las llamadas a `setPosition()` se acumulan más rápido que el tiempo de animar una sola jugada.

El runner soporta cuatro tipos de métrica:

- Tiempo de operación: se mide con `performance.now()` el costo de cada iteración del escenario; el runner agrega los estadísticos descritos arriba.
- Estabilidad de FPS: un colector basado en `requestAnimationFrame` registra los deltas entre callbacks consecutivos durante una operación y deriva estadísticos de FPS (mediana, percentil 95) junto con el conteo de frames cuyo delta excede el presupuesto del monitor de medición.

- Mutaciones del DOM: mediante `MutationObserver` se contabilizan las mutaciones inducidas por cada operación (nodos añadidos, eliminados, cambios de atributos y de texto) y el delta en la cuenta total de nodos.
- Huella de memoria: en navegadores que exponen `performance.memory`, se reporta el heap usado y el heap total antes y después de cada escenario; la precisión aumenta cuando el navegador se inicia con flags que habilitan información detallada de memoria.

5.6. CRITERIOS DE EVALUACIÓN

El éxito del trabajo se evalúa contra los objetivos específicos enunciados en la introducción. La correspondencia entre cada objetivo y los criterios concretos es la siguiente:

- OE1 (TypeScript): cobertura de tipos sobre la API pública del componente, preservación de la suite de ejemplos original e integración en el codebase TypeScript de ChessMindAI sin necesidad de archivos `.d.ts` externos.
- OE2 (arquitectura): capacidad demostrada de añadir extensiones nuevas (capas de anotación, transición morfológica entre posiciones) sin modificar el núcleo del componente.
- OE3 (animación y rendering): comparativa cuantitativa frente al componente original en tiempo de operación de animaciones, estabilidad de FPS bajo secuencias sostenidas y huella de memoria, reportada por el harness sobre las suites correspondientes.
- OE4 (harness): reproducibilidad del benchmark a partir de un protocolo documentado, versiones fijadas de las librerías evaluadas y posibilidad de obtener resultados equivalentes ejecutando las mismas suites en el mismo entorno.
- OE5 (optimizaciones complementarias): descripción técnica de cada optimización (pool de piezas, máquina de estados de input, transición morfológica) y, donde la suite del harness lo permita, número de reservas de memoria evitadas o reducción medida.

CAPÍTULO 6

RESULTADOS

Este capítulo presenta los resultados obtenidos al ejecutar el harness de benchmark descrito en el capítulo de Experimentación. El objetivo es contrastar cuantitativamente cm-chessboard-2 con el componente original, Chessground y chessboard.js en las suites que el harness implementa, y discutir qué decisiones técnicas del componente se reflejan en cada resultado.

6.1. ENTORNO DE MEDICIÓN

Las mediciones de este capítulo se ejecutaron bajo las condiciones que describe la Tabla 3.

Parámetro	Valor
Equipo	ASUS Zenbook 14 UX3405CA
Procesador	Intel Core Ultra 7 255H, 16 núcleos físicos (navigator.hardwareConcurrency reporta 6 al navegador)
GPU	Intel Arc Pro 130T/140T (integrada)
Memoria física	16 GB (navigator.deviceMemory reporta 8 GB, limitado por privacidad)
Sistema operativo	Fedora Linux 43 (Workstation Edition), x86_64
Kernel	Linux 7.0.4-100.fc43.x86_64
Servidor de ventanas	Wayland bajo GNOME 49.6 (Mutter)
Monitor	Panel interno 14", 2880×1800, 120 Hz
Viewport del navegador	1.728 × 967 píxeles lógicos, DPR 1,67
Navegador	Google Chrome 148.0.0.0 (Linux x86_64)
Flags del navegador	--enable-precise-memory-info, --js-flags="--expose-gc"
Perfil de energía	Máquina enchufada, sin throttling térmico, sin autosuspend, sin aplicaciones de carga en paralelo
Versión original cm-chessboard	8.5.0
Versión Chessground	9.1.1
Versión chessboard.js	@chrisoakman/chessboardjs@1.0.0 (con jQuery 3.7.1)

Tabla 3: Entorno bajo el que se ejecutó el harness de benchmark.

6.2. PROTOCOLO DE EJECUCIÓN

El harness incluye once suites: render, set-position, set-position-anim, move-instant, move-animated, bulk-moves, orientation, resize, destroy-create, memory y sustained-memory. Para esta corrida se ejecutaron las ocho suites comparables (las dos variantes con animación, set-position-anim y move-animated, quedaron excluidas: las cuatro librerías definen la animación de manera estructuralmente distinta y los tiempos por iteración no son una medida comparable entre ellas; la suite memory también queda excluida por reportar una unidad distinta a los milisegundos). El bucle externo recorre las suites; el bucle interno recorre los cuatro adaptadores (cm-chessboard-2, cm-chessboard original, Chessground, chessboard.js) en orden fijo. Con 200 iteraciones por (suite, adaptador) y 3 iteraciones de warmup descartadas, el total combinatorio de la corrida fue $200 \times 8 \times 4 = 6.400$ invocaciones de medición. La duración de animación configurada para las suites con animación (excluidas de la comparativa) fue de 25 ms.

El cuello de botella que motiva el rediseño del AnimationManager no aparece en una animación aislada, sino en la composición y la cancelación bajo secuencias rápidas, escenario que captura la suite bulk-moves, cuyos resultados reporta la Tabla 7.

6.3. SUITE RENDER

Mide el costo de construir el componente desde cero (creación del nodo SVG, montaje del set inicial de piezas, primer paint). Es la métrica más cercana al «first contentful paint» que percibe un usuario al abrir una página.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	2,28	2,00	3,90
cm-chessboard	1,99	1,70	4,00
Chessground	1,04	1,00	1,50
chessboard.js	2,40	2,00	3,80

Tabla 4: Tiempo promedio por iteración en la suite render.

La Tabla 4 muestra tiempos comparables entre cm-chessboard-2 (2,28 ms) y el componente original (1,99 ms), con una diferencia absoluta de 0,3 ms dentro del rango de ruido de la medición. Chessground encabeza el grupo con 1,04 ms y chessboard.js queda a la par con cm-chessboard-2 (2,40 ms).

6.4. SUITE SET-POSITION

Mide el costo de aplicar una nueva posición al tablero sin animación (`setPosition(fen, false)`). Aísla el costo de actualización del DOM del costo de la animación.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	1,18	1,20	1,40
cm-chessboard	17,24	16,70	24,50
Chessground	< 0,01	< 0,01	< 0,01
chessboard.js	5,99	5,60	10,50

Tabla 5: Tiempo promedio por iteración en la suite set-position.

La Tabla 5 reporta una reducción de 17,24 ms a 1,18 ms al pasar de cm-chessboard al fork, situando a cm-chessboard-2 en el mismo orden de magnitud que chessboard.js (5,99 ms). Chessground domina con tiempos por debajo del piso de medición.

6.5. SUITE MOVE-INSTANT

Mide el costo de un movimiento individual sin animación (`movePiece(from, to, false)`). Complementa a set-position con una operación más localizada.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	8,94	8,60	9,60
cm-chessboard	50,54	50,00	51,10
Chessground	1,05	1,00	1,20
chessboard.js	9,86	7,70	15,30

Tabla 6: Tiempo promedio por iteración en la suite move-instant.

La Tabla 6 muestra a cm-chessboard-2 con 8,94 ms frente a 50,54 ms del componente original, y un comportamiento similar al de chessboard.js (9,86 ms). Chessground se mantiene como el más rápido con 1,05 ms.

6.6. SUITE BULK-MOVES

Ejecuta el replay de la partida de la Ópera (Morphy vs. Duque de Brunswick y Conde Isouard, París 1858), 33 medias jugadas aplicadas en sucesión. Es el escenario principal de carga sostenida: las llamadas a `setPosition()` se acumulan más rápido que el tiempo de animar una sola jugada, lo que ejercita la cancelación cooperativa del motor de animación. La suite reporta el tiempo total del replay por iteración.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	49,33	50,15	55,60
cm-chessboard	537,25	533,40	566,80
Chessground	1,33	1,20	1,80
chessboard.js	453,67	413,85	966,00

Tabla 7: Tiempo total promedio del replay en la suite bulk-moves.

La Tabla 7 reporta la caída más fuerte respecto al componente original: cm-chessboard-2 reduce el tiempo total del replay de 537, 25 ms a 49, 33 ms. chessboard.js queda en 453, 67 ms con alta variabilidad (p95 de 966 ms), mientras que Chessground se mantiene por debajo de los 2 ms.

6.7. SUITE ORIENTATION

Mide el costo de invertir la orientación del tablero (cambiar el color que se muestra abajo). Ejercita el recálculo de coordenadas y el reposicionamiento de todas las piezas presentes.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	4,49	4,40	5,20
cm-chessboard	33,67	33,30	33,80
Chessground	1,19	1,10	1,30
chessboard.js	2,29	1,95	2,80

Tabla 8: Tiempo promedio por iteración en la suite orientation.

La Tabla 8 muestra que cm-chessboard-2 baja de 33,67 ms a 4,49 ms respecto al componente original. chessboard.js (2, 29 ms) y Chessground (1, 19 ms) son más rápidos, pero ambos resuelven la inversión sin recalcular coordenadas pieza por pieza.

6.8. SUITE RESIZE

Mide el costo de cambiar las dimensiones del contenedor del tablero y refrescar las coordenadas en consecuencia. Ejercita el pipeline de rendering SVG, el uso de ResizeObserver y la coalescencia con requestAnimationFrame.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	1,78	1,70	2,00
cm-chessboard	5,47	4,90	5,80
Chessground	1,29	1,30	1,40
chessboard.js	3,02	2,90	3,40

Tabla 9: Tiempo promedio por iteración en la suite resize.

La Tabla 9 muestra a cm-chessboard-2 en 1,78 ms frente a 5,47 ms del componente original. El valor queda cercano al de Chessground (1,29 ms) y mejora a chessboard.js (3,02 ms).

6.9. SUITE DESTROY-CREATE

Mide el ciclo completo de destrucción y recreación del componente. Refleja el costo de montar y desmontar el tablero, escenario típico cuando se navega entre vistas que crean y destruyen instancias.

Adaptador	Media (ms)	Mediana (ms)	p95 (ms)
cm-chessboard-2	2,31	2,30	3,10
cm-chessboard	2,83	2,80	2,90
Chessground	1,78	1,40	1,60
chessboard.js	3,76	3,70	4,10

Tabla 10: Tiempo promedio por iteración en la suite destroy-create.

La Tabla 10 reporta tiempos cercanos entre cm-chessboard-2 (2,31 ms) y cm-chessboard (2,83 ms). Chessground (1,78 ms) es el más rápido y chessboard.js (3,76 ms) el más lento del grupo.

6.10. SPEEDUP DE CM-CHESSBOARD-2 FRENTE A LA VERSIÓN ORIGINAL

La Tabla 11 consolida la mejora de tiempo medio de cm-chessboard-2 respecto al componente original para las siete suites de tiempo reportadas. Valores mayores que 1,00 × indican que cm-chessboard-2 es más rápido; valores menores indican regresión.

Suite	cm-chessboard-2 media (ms)	Original media (ms)	Speedup
render	2,28	1,99	0,87 ×
set-position	1,18	17,24	14,66 ×
move-instant	8,94	50,54	5,65 ×
bulk-moves	49,33	537,25	10,89 ×
orientation	4,49	33,67	7,49 ×
resize	1,78	5,47	3,07 ×
destroy-create	2,31	2,83	1,22 ×

Tabla 11: Speedup promedio de cm-chessboard-2 frente al componente original por suite.

6.11. MUTACIONES DEL DOM POR SUITE

Las optimizaciones de rendering de cm-chessboard-2 (composite layers, recálculo coalescido con requestAnimationFrame) buscan reducir el trabajo del navegador durante las operaciones. Un proxy técnico de ese objetivo es la cantidad de mutaciones del DOM que cada operación induce. La Tabla 12 reporta el conteo total de mutaciones (nodos añadidos, removidos, atributos modificados y cambios de texto) acumulado sobre las 200 iteraciones de cada suite, por adaptador.

Suite	cm- -chessboard-2	Original	Chessground	chessboard.js
render	43.848	75.515	9.540	9.135
set-position	33.594	38.869	46	3.671
move-instant	137.025	174.579	9.540	9.744
bulk-moves	939.687	974.399	9.540	12.789
orientation	39.395	147.345	9.384	9.789
resize	40.014	96.125	9.386	10.197
destroy-create	43.848	75.515	9.540	9.135

Tabla 12: Mutaciones promedio del DOM acumuladas por suite y adaptador.

6.12. DISCUSIÓN

6.12.1. LECTURA CONJUNTA DE LAS SUITES

cm-chessboard-2 mejora al componente original en seis de las siete suites de tiempo reportadas; la única excepción es render. La mejora más fuerte aparece en set-

position (14,66 ×) y en bulk-moves (10,89 ×); le siguen orientation (7,49 ×) y move-instant (5,65 ×); resize y destroy-create muestran mejoras menores (3,07 × y 1,22 × respectivamente). En render cm-chessboard-2 es 0,87 × respecto al componente original, una diferencia absoluta de 0,3 ms que cae dentro del ruido de medición (las desviaciones estándar de ambos adaptadores son varias veces mayores que la diferencia observada).

El patrón es coherente: las mejoras se concentran en operaciones que ejercitan el motor de animación y el pipeline de actualización del DOM (set-position, bulk-moves, orientation), mientras que en operaciones puntuales y de construcción inicial (render, destroy-create) la ventaja es marginal. Esto refleja que las inversiones técnicas de cm-chessboard-2 se concentran en el hot path de uso educativo (transiciones rápidas, navegación entre posiciones) y no en el camino de inicialización, que ya era satisfactorio en el componente original.

La Tabla 12 refuerza la lectura: cm-chessboard-2 reduce el churn de DOM respecto al componente original en todas las suites donde mejora el tiempo. La reducción más notoria es orientation (39.395 mutaciones vs 147.345, una disminución del 73%), seguida por resize (58%) y render (42%). En bulk-moves la reducción es modesta (4%) porque el escenario aplica 33 medias jugadas y la carga absoluta de mutaciones es inevitablemente alta.

6.12.2. ATRIBUCIÓN A DECISIONES TÉCNICAS

Las suites donde cm-chessboard-2 muestra las mejoras más fuertes corresponden a las familias de decisiones del capítulo de Diseño e implementación:

- La aceleración de bulk-moves (10,89 ×) se atribuye al rediseño del AnimationManager con cancelación cooperativa: la cola FIFO de la versión original obliga a esperar a que cada movida termine antes de animar la siguiente, mientras que cm-chessboard-2 cancela animaciones en curso y solapa la siguiente, reduciendo el tiempo total del replay de 537 ms a 49 ms.
- La aceleración de set-position (14,66 ×) y la reducción de mutaciones del DOM en esa suite se atribuyen a las optimizaciones del pipeline de rendering SVG: actualización incremental de las piezas en lugar de recreación del nodo, uso de transform en propiedades animables y cache de cálculos de coordenadas.
- La aceleración de orientation (7,49 ×) y su reducción del 73% en mutaciones se atribuyen al recálculo de coordenadas con coalescencia por requestAnimationFrame: en lugar de regenerar el contenido del tablero al invertir la orientación, cm-chessboard-2 actualiza atributos transform de los elementos existentes.
- La aceleración de resize (3,07 ×) se atribuye a la combinación de ResizeObserver con requestAnimationFrame que coalesce eventos de redimensionado y al uso de aspect-ratio en CSS para evitar recálculos en JavaScript.

La mejora en `destroy-create` es modesta ($1,22 \times$) y la regresión en `render` indica que la migración a TypeScript y la arquitectura ampliada introducen un costo de construcción inicial dentro del ruido. Estas operaciones no son hot path en el caso de uso educativo.

6.12.3. COMPARATIVA CUALITATIVA CON LAS ALTERNATIVAS

Chessground domina las tablas de tiempo (Tablas 4 a 10): es entre 1,6 y 4.310 veces más rápido que el componente original en todas las suites. Esto refleja la madurez de su implementación y un modelo de DOM minimalista visible en la Tabla 12 (donde Chessground se mantiene en torno a 9.540 mutaciones para la mayoría de las suites, con un valor excepcional de 46 en `set-position`). Sin embargo, su licencia GPL-3.0 lo descarta como componente para ChessMindAI, y carece de `MorphTransition` para transiciones entre posiciones arbitrarias.

`chessboard.js` muestra desempeño irregular: en `bulk-moves` su tiempo mediano es 413,85 ms pero su p95 alcanza 966 ms, con una desviación estándar de 303 ms; comparado con `cm-chessboard-2` (p95 de 55,60 ms, desviación estándar de 9,59 ms), su comportamiento bajo carga es considerablemente menos predecible. A esto se suma su dependencia de jQuery y la ausencia de tipos estáticos.

`cm-chessboard-2` ofrece un balance: es entre 3 y 14 veces más rápido que el componente original en operaciones intensivas, mantiene mayor predictibilidad que `chessboard.js` en escenarios de carga, preserva la licencia MIT, conserva la API pública de la versión original y agrega `MorphTransition` y un tipado completo. Las cifras absolutas en milisegundos lo sitúan en el orden de magnitud de Chessground para varias operaciones (`set-position`: 1,18 ms vs $< 0,01$ ms; `resize`: 1,78 ms vs 1,29 ms; `destroy-create`: 2,31 ms vs 1,78 ms), manteniendo licencia compatible con el producto.

6.13. LIMITACIONES DE LA EVALUACIÓN

La evaluación presentada tiene limitaciones que conviene reconocer al interpretar los resultados:

- Un solo motor de renderizado. Las mediciones se realizaron únicamente en Google Chrome de escritorio. Gecko (Firefox) y WebKit (Safari) implementan el pipeline de rendering, el compositor y el GC con estrategias distintas, y los números pueden diferir. Extender la evaluación a esos motores queda como trabajo futuro.
- Sin medición cuantitativa en dispositivos móviles. El componente funciona en navegadores móviles como consecuencia de funcionar bien en escritorio, pero la evaluación cuantitativa de ese escenario no es objetivo de este trabajo y no se realizó; replicar el harness en dispositivos móviles queda como trabajo futuro.

- Orden de ejecución secuencial. El harness ejecuta los adaptadores en orden fijo (cm-chessboard-2, original, Chessground, chessboard.js) dentro de cada suite. La deriva térmica o de carga del sistema entre los primeros y últimos adaptadores puede introducir sesgo; el control sobre el entorno (máquina enchufada, sin throttling, sin aplicaciones pesadas en paralelo) lo limita pero no lo elimina por completo.
- Regresión menor en render dentro del ruido de medición. En la suite render cm-chessboard-2 resulta $0,87 \times$ respecto al componente original (2,28 ms vs 1,99 ms). La diferencia absoluta es de 0,3 ms; las desviaciones estándar de ambos adaptadores (1,36 ms y 1,11 ms) son varias veces mayores que esa diferencia, por lo que el resultado es consistente con ruido de medición. La construcción inicial del componente original ya era satisfactoria para el caso de uso, lo que reduce el interés práctico de esta cifra.
- Comparativa a nivel de componente. El harness mide cada librería en aislamiento, no integrada en la aplicación completa de ChessMindAI. La interacción con el resto del frontend (Vue 3, estado global, capas superiores) puede modificar el comportamiento observado; medir el componente integrado en la aplicación completa queda como trabajo futuro.
- Suites de animación excluidas de la comparativa numérica. Las suites set-position-anim y move-animated no se reportan en este capítulo porque las librerías definen la animación de forma estructuralmente distinta; los tiempos no son comparables. El efecto del rediseño del motor de animación se observa en la suite bulk-moves (secuencias rápidas que obligan a cancelar animaciones en curso) y en las suites de redimensionamiento y orientación.
- Cobertura parcial de la captura de frames. El colector basado en requestAnimationFrame integrado al harness para registrar tiempos entre frames produjo datos comparables únicamente para el componente original. Para cm-chessboard-2, Chessground y chessboard.js el colector no capturó ningún frame, debido a que sus operaciones se completan en tiempos demasiado cortos para sostener una ventana de muestreo o se ejecutan en el compositor del navegador fuera del hilo principal. Por esta razón, no se reporta tabla de FPS en este capítulo; ajustar la instrumentación para producir datos comparables queda como trabajo futuro.
- Huella de memoria no reportada en esta corrida. La suite memory apoyada en performance.memory no produjo valores útiles para cm-chessboard-2 y componente original (todos los valores son cero) y entregó datos ruidosos para Chessground y chessboard.js. Aislar el efecto del DraggablePiecePool con una métrica cuantitativa requiere reinstrumentar la suite con el flag --enable-precise-memory-info activo durante toda la corrida y un protocolo de medición de heap delta dedicado; se deja como trabajo futuro.
- No se evaluó la percepción del usuario. Las métricas son técnicas. La pregunta de cuánto contribuye cada mejora a la experiencia del estudiante requiere un estudio A/B con usuarios, que se propone como trabajo futuro.

CAPÍTULO 7

CONCLUSIONES

Este capítulo cierra la memoria sintetizando el cumplimiento de los objetivos planteados en la introducción, listando las contribuciones del trabajo y discutiendo las limitaciones encontradas. Cierra con líneas de trabajo futuro que extenderían naturalmente lo aquí desarrollado.

7.1. SÍNTESIS DE RESULTADOS

A continuación se retoma cada objetivo específico y se discute el grado de cumplimiento alcanzado. Los números concretos que respaldan cada afirmación se encuentran en el capítulo de Resultados.

- OE1, migración a TypeScript. La totalidad del código de cm-chessboard-2 está escrita en TypeScript. La API pública del componente expone tipos, y la integración en el codebase de ChessMindAI se realiza sin archivos de declaración externos. El tradeoff aceptado es la configuración permisiva del compilador, que deja fuera del análisis estático los errores relacionados con valores nulos. Esa decisión está documentada y es revisable en iteraciones futuras.
- OE2, arquitectura MVC con sistema de extensiones. cm-chessboard-2 preserva la variante simplificada del patrón MVC del componente original y, sobre los ocho puntos de extensión heredados, agrega las extensiones que requieren las features de ChessMindAI. La capacidad de añadir extensiones (capas de anotación, transición morfológica entre posiciones) sin modificar el núcleo se demostró durante el propio desarrollo del componente.
- OE3, optimizaciones de animación y rendering. El rediseño del AnimationManager reemplaza la cola FIFO del componente original por cancelación cooperativa basada en tokens, eliminando la latencia acumulada en secuencias rápidas. La promoción de piezas a su propia composite layer durante la animación reduce los repintes del resto del tablero. El pipeline de rendering SVG incorpora coalescencia de ResizeObserver con requestAnimationFrame para evitar trabajo redundante bajo redimensionamiento. La comparativa cuantitativa frente al componente original confirma el impacto del rediseño: cm-chessboard-2 resulta entre $3 \times$ y $14 \times$ más rápido en las suites de operaciones intensivas, con $10,89 \times$ de speedup en el escenario de carga sostenida del replay de la partida de la Ópera. Las cifras completas y la atribución por suite se reportan en el capítulo de Resultados.
- OE4, harness de benchmark. El harness se construyó como un conjunto de páginas HTML servibles localmente, con adaptadores para cm-chessboard-2, el componente original, Chessground y chessboard.js, y un runner que aplica fases de warmup, iteraciones

de medición y reporte de estadísticos. La parametrización sigue recomendaciones de la literatura de benchmarking riguroso. La reproducibilidad se sostiene por la documentación del protocolo y la fijación de versiones de las librerías evaluadas.

- OE5, optimizaciones complementarias. `DraggablePiecePool` reduce las reservas de memoria inducidas por el drag. La máquina de estados de input formaliza las transiciones entre los modos de interacción (idle, hover, drag, drop) que el componente original manejaba con condicionales dispersos. La extensión `MorphTransition` ofrece transiciones entre posiciones arbitrarias, operación que el componente original no proporciona.

7.2. CONTRIBUCIONES

El trabajo deja tres contribuciones principales:

- Artefacto: `cm-chessboard-2`, un fork de `cm-chessboard` tipado, con motor de animación rediseñado, pipeline de rendering SVG optimizado, pool de piezas para drag, máquina de estados de input y extensión de transiciones entre posiciones, bajo licencia MIT preservada del componente original y en uso real en el frontend de ChessMindAI.
- Harness de benchmark: un conjunto reproducible de suites HTML con adaptadores para las principales librerías de tableros web. El harness puede reutilizarse para comparar otras librerías o nuevas versiones de `cm-chessboard-2` sin modificar el runner.
- Documentación arquitectónica: la memoria reúne en un solo documento las decisiones de diseño, su justificación técnica y la comparativa empírica contra alternativas, sirviendo de base para futuras iteraciones del componente y para integradores que quieran entender cómo opera.

7.3. LIMITACIONES DEL TRABAJO

Algunas limitaciones se asumieron por alcance y otras emergieron durante la ejecución. Las primeras quedaron explícitas en la definición del problema (estudio con usuarios y evaluación cuantitativa en motores y dispositivos móviles, ambos propuestos como trabajo futuro); las segundas se reconocen aquí:

- Configuración no estricta del compilador TypeScript. La migración optó por una configuración permisiva que dejó fuera del análisis estático varios patrones de código. Endurecer la configuración requeriría una segunda iteración de refactoring con costo no trivial.
- Cobertura del harness en un solo motor de rendering. La evaluación se ejecutó en Google Chrome de escritorio. Replicar el harness en Gecko y WebKit requiere ajustes de adaptación (algunas APIs como `performance.memory` solo están disponibles en Chromium) pero es viable.

- Instrumentación parcial de fluidez y memoria. El colector de frames basado en `requestAnimationFrame` capturó datos comparables solo para el componente original; la suite de huella de memoria no produjo valores útiles en la corrida reportada. Ambas limitaciones se discuten en el capítulo de Resultados y quedan abiertas como tareas de instrumentación.
- Atribución a optimizaciones individuales. Las optimizaciones se introdujeron por familias y no como cambios individuales aislados. Aislar el efecto de cada decisión exigiría un protocolo de bisección con compilados intermedios del componente, que no se implementó.

7.4. TRABAJO FUTURO

Seis líneas extienden naturalmente el trabajo:

1. Instrumentación de fluidez y memoria. Ajustar el colector de tiempos entre frames para que capture datos comparables en los cuatro adaptadores (por ejemplo, prolongando la ventana de muestreo o instrumentando el hilo del compositor) y reinstrumentar la suite de huella de memoria con un protocolo dedicado que aisle el efecto del `DraggablePiecePool`. Ambas métricas existen en la infraestructura pero no produjeron datos comparables en la corrida reportada.
2. Estudio de percepción con usuarios. Diseñar un experimento A/B con estudiantes de ChessMindAI para evaluar si las mejoras técnicas se traducen en una mejor experiencia educativa. Requiere protocolo ético y tamaño muestral adecuado.
3. Evaluación multi-motor y móvil. Adaptar el harness a Gecko y WebKit (incluyendo dispositivos móviles) y reportar la comparativa. El componente ya funciona en estos entornos como consecuencia de funcionar bien en Google Chrome de escritorio, pero la medición sistemática no se realizó.
4. Endurecimiento del tipado. Migrar la configuración del compilador a `strict: true` por etapas, comenzando por `strictNullChecks`. Cada etapa exige refactoring de los puntos del código que dependen implícitamente de valores ausentes.
5. Extensión del repertorio de animaciones. La extensión `MorphTransition` aborda las transiciones entre posiciones arbitrarias con el algoritmo Húngaro de asignación óptima; explorar heurísticas alternativas más ligeras o ajustes en la función de costo podría reducir el tiempo de cálculo en escenarios con muchas piezas presentes.
6. Contribución selectiva al proyecto original. Las optimizaciones más genéricas (cancelación cooperativa, composite layers, coalescencia de `ResizeObserver`) podrían proponerse al proyecto original como pull requests acotados. La migración a TypeScript y el sistema de extensiones específico de ChessMindAI no son apropiados para incorporar aguas arriba, pero el subconjunto de optimizaciones de rendering sí lo es.

ANEXOS

Los anexos reúnen material complementario al cuerpo principal de la memoria: protocolo de medición, especificación del entorno y mapa de las decisiones técnicas de cm-chessboard-2.

7.5. ANEXO A: PROTOCOLO DE MEDICIÓN

Este anexo detalla los pasos necesarios para reproducir el benchmark reportado en el capítulo de Resultados.

7.5.1. ESTRUCTURA DEL HARNESS

El harness consiste en un conjunto de páginas HTML, cada una correspondiente a una suite, y un runner JavaScript común. Cada suite carga las cuatro librerías evaluadas mediante adaptadores que exponen una interfaz uniforme (`setPosition`, `movePiece`, `setOrientation`, `resize`, `destroy`) y permite ejecutar el escenario sobre cada una de ellas.

7.5.2. EJECUCIÓN

Los pasos para reproducir las mediciones son:

1. Servir el directorio del harness con un servidor HTTP local:
`bunx serve -p 3000`
2. Iniciar Google Chrome con las flags requeridas:
`google-chrome --enable-precise-memory-info --js-flags="--expose-gc"`
3. Abrir la URL del runner principal en el navegador (`http://localhost:3000/...`) y verificar que las cuatro librerías queden listadas como disponibles.
4. Configurar el número de iteraciones (200 en la corrida reportada) y seleccionar las suites deseadas; pulsar el control de inicio.
5. Esperar a que el runner reporte los estadísticos al terminar y exportar el resultado al archivo JSON.

7.5.3. PARAMETRIZACIÓN DEL RUNNER

La Tabla 13 lista los parámetros del runner usados en las mediciones reportadas.

Parámetro	Valor
Corridas de warmup descartadas	3 (fijo en el runner)
Iteraciones de medición por (suite, adaptador)	200 (configurable desde la UI)
Suites del harness	11 totales; 8 comparables reportadas (excluidas las dos variantes con animación y la suite de huella de memoria, esta última por usar una unidad distinta de los milisegundos)
Adaptadores comparados	4 (cm-chessboard-2, cm-chessboard original, Chessground, chessboard.js)
Total combinatorio de la corrida reportada	$200 \times 8 \times 4 = 6.400$ medidas
Estadísticos reportados	media, mediana, mínimo, máximo, p95, desv. estándar
Hint de recolección de basura entre iteraciones	opcional, activable con <code>--js-flags="--expose-gc"</code>
Tipos de métrica reportados en este documento	tiempo de operación (media, mediana, p95) y mutaciones del DOM (totales por suite). El colector de tiempos entre frames y la suite de huella de memoria del harness se instrumentaron pero no produjeron datos comparables en esta corrida; ver limitaciones del capítulo de Resultados
Formato de exportación	JSON estructurado por corrida (results[] con suiteId, adapter, unit, times[], stats, mutations, frames)
Orden de ejecución	secuencial: bucle externo por suite, bucle interno por adaptador, sin recarga entre adaptadores
Presupuesto de frame del monitor de medición	8,3 ms (120 Hz)

Tabla 13: Parámetros del runner del harness.

7.6. ANEXO B: ENTORNO DE MEDICIÓN

La Tabla 14 detalla el equipo, el sistema operativo, el navegador y las flags utilizadas durante la corrida reportada.

Parámetro	Valor
Equipo	ASUS Zenbook 14 UX3405CA, 2024
Procesador	Intel Core Ultra 7 255H, 16 núcleos físicos (navigator.hardwareConcurrency reporta 6)
Frecuencia turbo	5,10 GHz
GPU	Intel Arc Pro 130T/140T (integrada, 2,25 GHz)
Memoria física	16 GB (navigator.deviceMemory reporta 8 GB, limitado por privacidad)
Swap	8 GB
Almacenamiento	SSD NVMe, sistema de archivos btrfs
Sistema operativo	Fedora Linux 43 (Workstation Edition), x86_64
Kernel	Linux 7.0.4-100.fc43.x86_64
Entorno gráfico	GNOME 49.6 sobre Wayland (compositor Mutter)
Monitor	Panel interno (SDC419D), 14", resolución 2880×1800, refresco 120 Hz
Viewport del navegador	1.728 × 967 píxeles lógicos, DPR 1,67
Navegador	Google Chrome 148.0.0.0 (Linux x86_64)
User-Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/148.0.0.0 Safari/537.36
Flags del navegador	--enable-precise-memory-info, --js-flags="--expose-gc"
Modo del sistema	Máquina enchufada a la red, screen saver y autosuspend desactivados, sin throttling térmico forzado, sin aplicaciones pesadas en paralelo durante la corrida
Locale	en_US.UTF-8
Fecha y hora de la corrida	13 de mayo de 2026, 19:20:09 a 19:24:15 UTC

Tabla 14: Entorno completo de medición.

7.7. ANEXO C: MAPA DE DECISIONES TÉCNICAS

La Tabla 15 presenta una vista compacta de las decisiones técnicas de cm-chessboard-2, agrupadas por familia, para facilitar la navegación del documento principal. Cada decisión se discute con detalle en el capítulo de Diseño e implementación.

ID	Decisión	Capítulo de tratamiento
D1	Migración completa a TypeScript	Diseño e implementación
D2	Arquitectura MVC con sistema de extensiones	Diseño e implementación
D3	Rediseño del AnimationManager (cancelación cooperativa, composite layers, recálculo bajo redimensionamiento)	Diseño e implementación
D4	Optimizaciones del pipeline de rendering SVG	Diseño e implementación
D5	DraggablePiecePool para drag de piezas	Diseño e implementación
D6	Máquina de estados de input	Diseño e implementación
D7	Harness de benchmark	Experimentación y Resultados
D8	Sub-path exports en package.json	Diseño e implementación
D9	Refactor de limpieza de código heredado	Diseño e implementación
D10	Extensión MorphTransition para transiciones entre posiciones	Diseño e implementación

Tabla 15: Decisiones técnicas de cm-chessboard-2 y dónde se discuten en la memoria.

REFERENCIAS BIBLIOGRÁFICAS

- Blackburn, S. M., Cheng, P., & McKinley, K. S. (2004). Myths and realities: the performance impact of garbage collection. *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '04)*, 25-36. <https://doi.org/10.1145/1005686.1005693>
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons.
- ChessMindAI. (2026,). *ChessMindAI*.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Gao, Z., Bird, C., & Barr, E. T. (2017). To type or not to type: quantifying detectable bugs in JavaScript. *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*, 758-769. <https://doi.org/10.1109/ICSE.2017.75>
- Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '07)*, 57-76. <https://doi.org/10.1145/1297027.1297033>
- Haack, S. (2024,). *cm-chessboard*. GitHub.
- Heiser, G. (2010,). *Systems Benchmarking Crimes*. UNSW Sydney.
- Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 83-97. <https://doi.org/10.1002/nav.3800020109>
- Lewis, P., & Irish, P. (2018,). *Animations and performance*. web.dev (Google).
- Lichess. (2024,). *Chessground*. GitHub.
- MDN Web Docs. (2025,). *CSS will-change*. Mozilla.
- Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1), 32-38. <https://doi.org/10.1137/0105003>
- Mytkowicz, T., Diwan, A., Hauswirth, M., & Sweeney, P. F. (2009). Producing wrong data without doing anything obviously wrong!. *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, 265-276. <https://doi.org/10.1145/1508244.1508275>
- Oakman, C. (2019,). *chessboard.js*. chessboardjs.com.
- Reenskaug, T. (1979). *Thing-Model-View-Editor: an example from a planning system* [Technical report].
- Siek, J. G., & Taha, W. (2006). Gradual typing for functional languages. *Proceedings of the Scheme and Functional Programming Workshop*, 81-92.

W3C. (2021,). *Resize Observer*. World Wide Web Consortium.

W3C. (2022,). *Preload*. World Wide Web Consortium.

W3C. (2023,). *Web Animations Level 1*. World Wide Web Consortium.