INSTITUTO POLITÉCNICO NACIONAL
"ESCUELA SUPERIOR DE CÓMPUTO"

SISTEMAS DISTRIBUIDOS

Práctica 8:

"PWA"

Alumno: **Miguel Ángel Vázquez Cisneros**

Profesor: **Carreto Arellano Chadwick**

Grupo: **7CM1**

Índice

Antecedentes

Antes de la aparición de las Progressive Web Apps (PWA), el desarrollo de aplicaciones se dividía principalmente en dos mundos separados: las aplicaciones web y las aplicaciones nativas.

Aplicaciones web tradicionales

Las aplicaciones web tradicionales, construidas con tecnologías como HTML, CSS y JavaScript, funcionaban únicamente dentro del navegador y requerían conexión constante a internet para cargar recursos, interactuar con el servidor o simplemente visualizar contenido. Aunque eran más fáciles de actualizar y mantener (porque los cambios se hacían desde el servidor), presentaban importantes limitaciones:

- No funcionaban sin conexión.
- No se podían instalar en dispositivos móviles como una app.
- No tenían acceso a características del dispositivo como notificaciones, almacenamiento local avanzado o sensores.
- No ofrecían una experiencia inmersiva o fluida comparable con las apps nativas.

Además, los navegadores no contaban con las capacidades modernas para manejar el almacenamiento inteligente de recursos, el acceso controlado a funciones del sistema o la ejecución de scripts en segundo plano.

Aplicaciones nativas

Por otro lado, las aplicaciones nativas eran desarrolladas para sistemas operativos específicos (como Android o iOS), utilizando lenguajes como Java/Kotlin para Android y Swift/Objective-C para iOS. Estas aplicaciones tenían ventajas claras:

- Funcionaban sin conexión.
- Accedían al hardware del dispositivo.
- Se podían instalar desde tiendas como Google Play o App Store.
- Ofrecían una experiencia fluida y rica para el usuario.

Sin embargo, presentaban desventajas clave para pequeños desarrolladores o proyectos simples:

- Altos costos de desarrollo y mantenimiento (una versión por plataforma).
- Curvas de aprendizaje pronunciadas.
- Procesos de publicación largos y estrictos en las tiendas de apps.

Planteamiento del Problema

En el desarrollo de videojuegos sencillos para la web, especialmente como parte de prácticas académicas o prototipos personales, es común enfrentarse a limitaciones relacionadas con la conectividad, el rendimiento y la experiencia del usuario. Las aplicaciones web tradicionales requieren una conexión constante a internet y dependen de los recursos del navegador en cada carga. Esto afecta la fluidez del juego, impide su uso en modo offline y limita la capacidad de acceso en dispositivos móviles.

Por otro lado, aunque las aplicaciones nativas ofrecen mejor rendimiento y acceso a funcionalidades del dispositivo, desarrollarlas implica mayor complejidad, costos elevados, y la necesidad de dominar tecnologías específicas para cada plataforma (Android, iOS, etc.), lo que no siempre es viable para proyectos simples o educativos.

En este contexto, surge la necesidad de una solución intermedia que combine la facilidad de desarrollo web con algunas capacidades propias de las aplicaciones nativas, como el almacenamiento local, el uso sin conexión y la instalación en dispositivos. Es decir, se requiere una forma de convertir una aplicación web en una experiencia más robusta, fluida y accesible, sin salir del entorno JavaScript ni depender de herramientas costosas.

Este problema es especialmente relevante en el contexto académico, donde los recursos son limitados, pero se busca lograr una experiencia de usuario más realista y completa, incluso en simulaciones sencillas como juegos de tablero o movimiento en cuadrícula. Por ello, es necesario encontrar una estrategia de desarrollo que permita:

- Ejecutar el juego sin conexión.
- Acelerar tiempos de carga.
- Permitir su instalación como si fuera una app.
- Ofrecer una experiencia uniforme en distintos dispositivos.

Propuesta de Solución

Para resolver las limitaciones de conectividad, accesibilidad y experiencia de usuario que presentan las aplicaciones web tradicionales en el desarrollo de videojuegos simples, se propone implementar una Progressive Web App (PWA) utilizando tecnologías modernas del ecosistema web, específicamente React, Vite y un Service Worker personalizado.

La idea central es transformar una simulación básica de videojuego, en este caso, una cuadrícula interactiva donde el jugador se mueve, ataca enemigos y gestiona su estado, en una aplicación web instalable, rápida y funcional sin conexión a internet, todo sin abandonar el desarrollo en JavaScript.

La solución se compone de los siguientes elementos:

- **Uso de React con Vite**: React permite una gestión eficiente del estado del juego y una interfaz dinámica y responsiva. Vite se utiliza como herramienta de construcción rápida (bundler) para optimizar tiempos de carga y experiencia de desarrollo.

- **Implementación de un Service Worker**: Un archivo JavaScript que intercepta las peticiones HTTP y gestiona la caché de archivos esenciales (HTML, JS, CSS, íconos, etc.). Esto permite que el juego funcione aun cuando el dispositivo está sin conexión.

- **Manifest Web App**: Se define un archivo manifest.json que permite instalar la aplicación como si fuera una app nativa en dispositivos móviles o de escritorio, proporcionando un ícono, nombre, color de fondo y comportamiento de pantalla completa.

- **Interacción mediante teclado y estado persistente**: La interfaz del juego permite moverse con teclas, atacar casillas y visualizar el estado del jugador. Todo esto se realiza de forma fluida y sin necesidad de recargar la página.

Esta solución no solo mejora la experiencia del usuario final, sino que también introduce al desarrollador en prácticas modernas de desarrollo web. Además, permite que aplicaciones simples como esta puedan funcionar como prototipos multiplataforma, accesibles desde cualquier navegador moderno, instalables en el dispositivo, y resistentes a la pérdida de conexión.

Materiales y Métodos Empleados

Para llevar a cabo esta práctica, se emplearon herramientas modernas del desarrollo web enfocadas en la creación de aplicaciones reactivas, ligeras y progresivas. A continuación, se detallan los materiales (software y recursos utilizados) y los métodos (cómo se implementaron las herramientas):

4.1 Materiales

- **React**
  Biblioteca de JavaScript para construir interfaces de usuario dinámicas y basadas en componentes reutilizables. Se usó para gestionar el estado del jugador, renderizar la cuadrícula y controlar la interacción mediante teclado.

- **Vite**
  Herramienta de construcción rápida (bundler) que permite iniciar y compilar aplicaciones React de forma muy eficiente, con recarga en tiempo real y tiempos de espera mínimos.

- **JavaScript (ES6+)**
  Lenguaje de programación utilizado tanto para la lógica del juego como para la creación del Service Worker.

- **Service Worker API**
  Tecnología nativa del navegador que permite interceptar y manejar solicitudes de red, habilitando funciones offline, almacenamiento en caché y manejo avanzado de recursos.

- **Web App Manifest (manifest.json)**
  Archivo de configuración que describe cómo debe comportarse la aplicación cuando se instala en un dispositivo. Define nombre, íconos, colores y orientación.

- **Tailwind CSS (opcional)**
  Framework de estilos utilizado para facilitar el diseño responsivo de la cuadrícula del juego y los elementos visuales.

- **Navegador web compatible con PWA**
  (Google Chrome, Microsoft Edge, Firefox, etc.). Necesario para probar características como instalación de la app, caché y funcionamiento sin conexión.

- **Editor de código**
  (Visual Studio Code). Usado para desarrollar y estructurar el código fuente de la aplicación.

4.2 Métodos

1. **Inicialización del proyecto con Vite + React**
   Se creó un proyecto base con el comando npm create vite@latest, seleccionando React como plantilla. Se configuró el entorno de desarrollo con los comandos npm install y npm run dev.

2. **Diseño de la lógica del juego**
   Se implementó una cuadrícula 10x10 renderizada dinámicamente mediante map(). El jugador es representado por un objeto con coordenadas (x, y), vida y nivel. Se implementaron controles mediante teclado (W, A, S, D, Enter y E) usando un useEffect para capturar los eventos.

3. **Definición de enemigos y mecánica de ataque**
   Se posicionaron enemigos en ciertas casillas fijas. Al atacar (presionar Enter), el sistema verifica si hay un enemigo en la casilla adyacente. Si lo hay, se reduce la vida del jugador y se incrementa su nivel.

4. **Implementación del Service Worker**
   Se creó un archivo service-worker.js personalizado con tres eventos principales:

   - install: para guardar en caché los archivos esenciales del juego.
   - activate: para eliminar versiones anteriores del caché.
   - fetch: para interceptar solicitudes y responder con recursos cacheados o de red.

5. **Creación del manifest.json**
   Se configuró el archivo de manifiesto para permitir la instalación de la aplicación. Se especificaron íconos, nombre, color de fondo y modo de visualización (standalone).

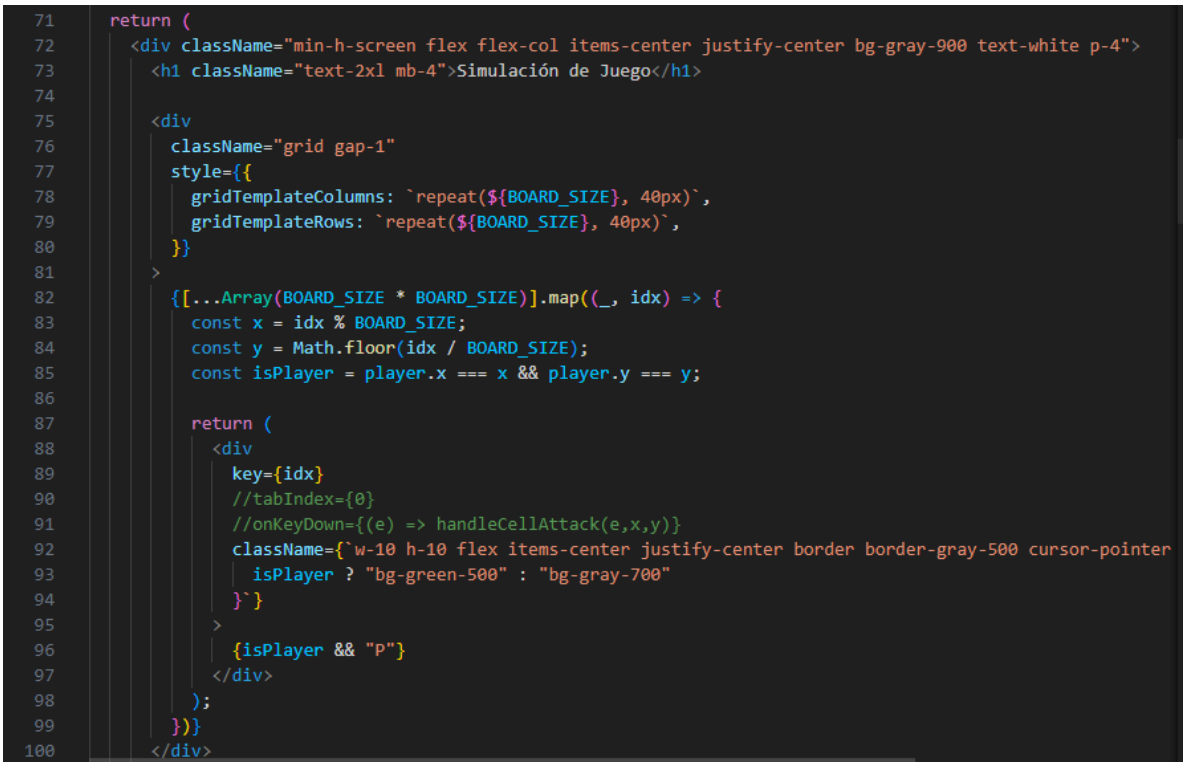
6. **Prueba de instalación y uso offline**
   Se ejecutó la aplicación en un navegador compatible, se realizó la instalación en escritorio/móvil y se probó el funcionamiento sin conexión a internet.

Desarrollo de la Solución

El desarrollo de esta práctica se enfocó en crear una aplicación web interactiva y progresiva, simulando un videojuego simple con elementos básicos como movimiento, ataque y evolución del personaje. A continuación, se describe cómo se construyó paso a paso:

5.1 Estructura del juego en React

Se diseñó una cuadrícula de 10x10 utilizando HTML dinámico dentro de un componente React. Cada celda de la cuadrícula es renderizada mediante un bucle, y se determina visualmente si contiene al jugador con base en sus coordenadas.



```
return (
  <div className="min-h-screen flex flex-col items-center justify-center bg-gray-900 text-white p-4">
    <h1 className="text-2xl mb-4">Simulación de Juego</h1>

    <div
      className="grid gap-1"
      style={{
        gridTemplateColumns: `repeat(${BOARD_SIZE}, 40px)`,
        gridTemplateRows: `repeat(${BOARD_SIZE}, 40px)`,
      }}
    >
      {[...Array(BOARD_SIZE * BOARD_SIZE)].map((_, idx) => {
        const x = idx % BOARD_SIZE;
        const y = Math.floor(idx / BOARD_SIZE);
        const isPlayer = player.x === x && player.y === y;

        return (
          <div
            key={idx}
            //tabIndex={0}
            //onKeyDown={(e) => handleCellAttack(e,x,y)}
            className={`w-10 h-10 flex items-center justify-center border border-gray-500 cursor-pointer
              isPlayer ? "bg-green-500" : "bg-gray-700"
            }`}
          >
            {isPlayer && "P"}
          </div>
        );
      })}
    </div>
```

El estado del jugador (x, y, vida, nivel) se administra usando useState. Además, se usa useEffect para capturar eventos de teclado (W, A, S, D) que permiten mover al jugador por la cuadrícula, y Enter para atacar.

El componente principal App.jsx gestiona:

- Movimiento del jugador dentro de los límites del tablero.
- Visualización condicional del estado del jugador (E para mostrar u ocultar).

- Lógica de ataque: si el jugador ataca una casilla que contiene un enemigo, pierde 20 puntos de vida y sube de nivel. Se muestra una alerta con el resultado del ataque.

```
// Movement handler
useEffect(() => {
  const handleKeyDown = (e) => {
    let { x, y, vida, nivel } = player;
    switch (e.key.toLowerCase()) {
      case "w":
        if (y > 0) y--;
        break;
      case "s":
        if (y < BOARD_SIZE - 1) y++;
        break;
      case "a":
        if (x > 0) x--;
        break;
      case "d":
        if (x < BOARD_SIZE - 1) x++;
        break;
      case "enter":
        for(let [xEnemy,yEnemy] of enemies){
          console.log(`xEnemy: ${xEnemy}, yEnemy: ${yEnemy}`)
          console.log(`xPlayer: ${x+1}, yPlayer: ${y+1}`)
          if((x+1)==xEnemy && (y+1)==yEnemy){
            isInEnemy = true;
            vida-=20;
            nivel++;
            alert(`¡Atacaste la casilla (${y+1}, ${x+1}). Has derrotado al enemigo.
              Tus nuevos status son: vida=${player.vida-20}, nivel=${player.nivel+1}`);
            break;
          }
        }
```

```
        if(!isInEnemy){
          alert(`¡Atacaste la casilla (${y+1}, ${x+1})!`);
        }

        break;
      case "e":
        setStatusVisible((prev) => !prev);
        return;
      default:
        return;
    }
    setPlayer({ x, y, vida, nivel });
  };

  window.addEventListener("keydown", handleKeyDown);
  return () => window.removeEventListener("keydown", handleKeyDown);
}, [player]);
```

5.2 Enemigos y combate

Los enemigos están definidos como coordenadas fijas en un arreglo. Cuando el jugador presiona Enter, se compara su posición con la de los enemigos para determinar si hay un enfrentamiento.

La condición del ataque se ejecuta dentro del handleKeyDown, y se actualiza el estado del jugador según el resultado. La lógica incluye alertas y mensajes informativos para retroalimentar al usuario.

5.3 Apariencia visual con Tailwind CSS

Se utilizó Tailwind CSS para aplicar estilos rápidos y responsivos:

- Celdas con bordes y colores diferenciados (verde para el jugador, gris para el fondo).
- Diseño centralizado y responsivo de la cuadrícula.
- Estilo adicional para mostrar los estatus del jugador en una tarjeta flotante.

```
@tailwind base;
@tailwind components;
@tailwind utilities;

:root {
  font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

a {
  font-weight: 500;
  color: #646cff;
  text-decoration: inherit;
}
a:hover {
  color: #535bf2;
}

```

5.4 Configuración del Service Worker

Para convertir la aplicación en una PWA funcional sin conexión, se creó un service-worker.js con tres eventos clave:

- install: guarda en caché los archivos estáticos necesarios (index.html, JavaScript, íconos, etc.).
- activate: elimina cachés antiguos si se detecta una nueva versión.
- fetch: intercepta las solicitudes; si el recurso está en caché, lo devuelve desde ahí, y si no, intenta obtenerlo de la red. Si falla, responde con index.html.

```
const CACHE_NAME = "game-cache-v1";
const FILES_TO_CACHE = [
  "/",
  "/index.html",
  "/manifest.json",
  "/favicon.ico",
  "/icons/icon-192x192.png",
  "/icons/icon-512x512.png",
  "/src/index.css",
  "/src/main.jsx",
  "/src/App.jsx"
];

//  1. Instalar el Service Worker y guardar archivos en caché
self.addEventListener("install", (event) => {
  console.log("[ServiceWorker] Install");
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      console.log("[ServiceWorker] Pre-caching offline page");
      return cache.addAll(FILES_TO_CACHE);
    })
  );
  self.skipWaiting();
});

```

Esto permite que el juego cargue correctamente aun sin conexión, después de su primera visita.

5.5 Archivo de manifiesto (manifest.json)

Se agregó un archivo manifest.json que describe los metadatos de la aplicación. Este incluye:

- Nombre corto y completo.

- Íconos en varios tamaños.
- Colores temáticos.
- Modo de inicio (standalone), que permite que se vea como una app nativa.

```
{
    "name": "Simulador de Videojuego",
    "short_name": "Simulador",
    "start_url": ".",
    "display": "standalone",
    "background_color": "#ffffff",
    "theme_color": "#000000",
    "icons": [
      {
        "src": "/icon-192x192.png",
        "sizes": "192x192",
        "type": "image/png"
      },
      {
        "src": "/icon-512x512.png",
        "sizes": "512x512",
        "type": "image/png"
      }
    ]
}
```

Esto habilita la opción de instalación desde el navegador (botón "Instalar app") en dispositivos compatibles.

5.6 Pruebas

Se probaron las siguientes funcionalidades:

- Movimiento correcto del jugador por el tablero.
- Ataque a casillas vacías y con enemigos.
- Actualización de nivel y vida tras el combate.
- Visualización de estatus (E).
- Instalación de la app como PWA.

Resultados

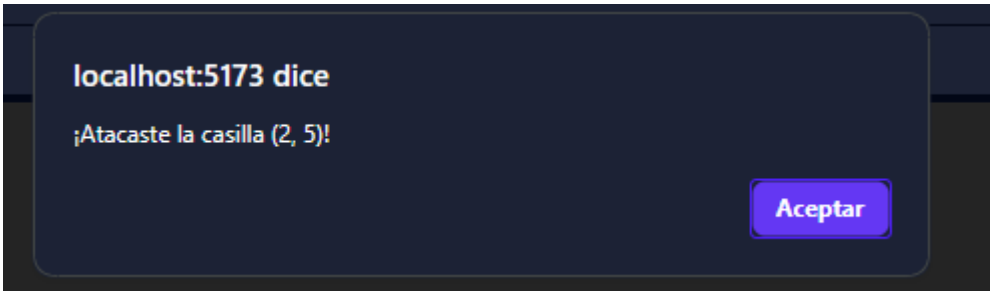
Como se esta usando Vite, se tiene que correr el proyecto con el comando “npm run dev” asegurándose que se encuentra en del directorio donde se encuentran los archivos de Vite.

```
C:\Users\Asher\Documents\ESCOM\Sistemas Distribuidos\Practica8\videojuego-pwa>npm run dev
"C:\Users\Asher\" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.

> videojuego-pwa@0.0.0 dev
> vite


  VITE v6.2.5  ready in 840 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

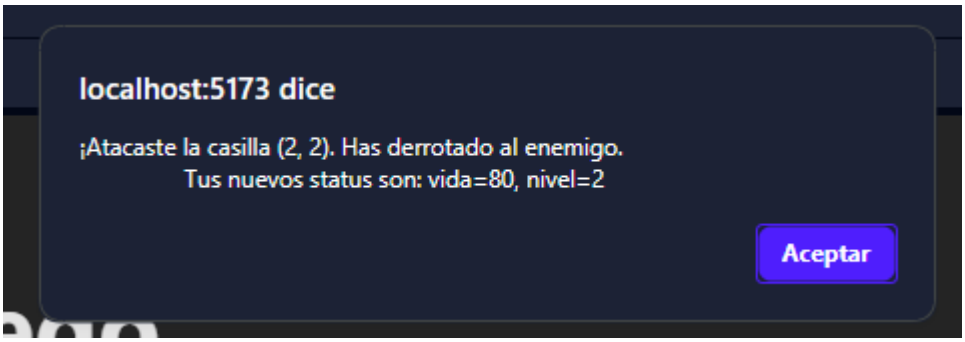
Una vez que se ha iniciado el servidor solo tenemos que acceder a él a través del navegador (de preferencia Chrome, ya que otros navegadores como Opera no permiten descargar la app).

La interfaz principal es una cuadrícula en la que el jugador se puede mover con las teclas “WASD”, como se muestra en la siguiente imagen.

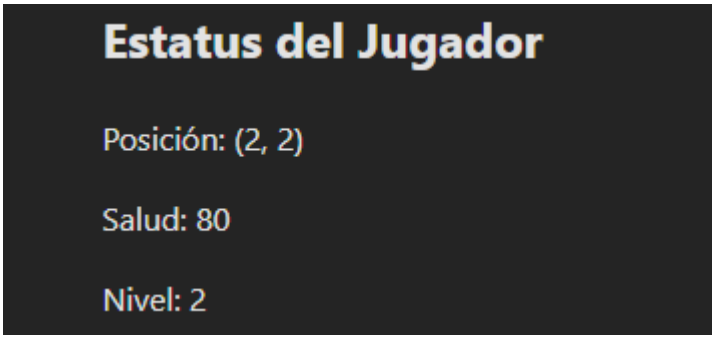
La letra P es el lugar en el que actualmente se encuentra el jugador. Si se presiona “enter”, entonces el jugador procede a atacar la casilla en la que se encuentre y nos aparecerá un mensaje como se muestra a continuación:



Si la casilla que se ataca tiene un enemigo, aparece el siguiente mensaje en su lugar:



Para visualizar tus estadísticas se presiona la tecla “e”, y estas aparecerán debajo de la cuadricula. Luego se puede volver a presionar “e” para ocultarlas de nuevo.



Finalmente, si es que lo deseamos, podemos instalar el programa a través del ícono que, generalmente, se encuentra en la esquina superior derecha de la pantalla,

Conclusiones

El desarrollo de esta práctica representó una oportunidad significativa para aplicar conceptos fundamentales de las Progressive Web Apps (PWA) y el desarrollo moderno con React y Vite en un contexto interactivo y motivador como lo es una simulación de videojuego. A lo largo del proyecto, se integraron diversas tecnologías que no solo mejoraron la experiencia del usuario, sino que también demostraron cómo es posible construir aplicaciones web más resistentes, funcionales y cercanas a las aplicaciones nativas.

Uno de los principales logros fue la implementación de la lógica del juego usando React, aprovechando su capacidad para gestionar estados y eventos de forma eficiente. Se simuló un entorno de juego sencillo, pero completo, que permite al usuario moverse dentro de una cuadrícula, atacar enemigos, visualizar su progreso (vida y nivel) y recibir retroalimentación inmediata. Este diseño sirvió como un ejemplo práctico de cómo estructurar interactividad con componentes reutilizables y estados controlados.

Otro aspecto clave fue la integración del Service Worker, lo que permitió transformar una simple SPA (Single Page Application) en una verdadera PWA. Esto implicó la creación de una estrategia de caché personalizada para almacenar recursos críticos, lo que aseguró que la aplicación funcionara incluso en condiciones de desconexión. La capacidad de cargar sin conexión después del primer uso representa una gran ventaja, especialmente para usuarios con conexiones inestables o en zonas con poca cobertura.

También se abordó la importancia del manifest.json, necesario para ofrecer una experiencia más coherente en dispositivos móviles, permitiendo la instalación del juego como si se tratara de una aplicación independiente, con íconos, nombre personalizado y modo pantalla completa. Esta funcionalidad amplía considerablemente el alcance de la aplicación y mejora la percepción del usuario sobre su profesionalismo y utilidad.

Durante el desarrollo se utilizaron herramientas modernas como **Vite**, que demostró ser una alternativa rápida y eficiente para compilar y ejecutar el entorno de desarrollo. Su compatibilidad con módulos de ES y recarga instantánea favoreció un ciclo de desarrollo fluido. Por otro lado, **Tailwind CSS** facilitó la estilización rápida del proyecto sin necesidad de escribir CSS tradicional desde cero, permitiendo prototipar y ajustar estilos con velocidad y precisión.

En términos de aprendizaje, esta práctica permitió entender no solo los beneficios técnicos de las PWA, sino también su impacto en la experiencia del usuario final. Se reafirmó que una aplicación web moderna puede ir más allá de una simple página: puede ser instalable, confiable, rápida y funcional incluso sin internet.