OPERATING SYSTEM PROJECT

PARTH KHANDELWAL

23BDS0043

**Clock Algorithm Page Replacement Visualizer**

**Index.html**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Clock Algorithm Page Replacement Visualizer</title>

    <!-- Link to the main stylesheet -->

    <link rel="stylesheet" href="style.css">

</head>

<body>

    <!-- Full-width header at the top of the page -->

    <header>

        <h1>Clock Algorithm Page Replacement Visualizer</h1>

    </header>

    <!-- Main container for the two-column layout -->

    <main id="main-container">

        <!-- ===== LEFT PANEL ===== -->

        <!-- This column holds the input form and the simulation controls -->

        <div class="left-panel">

            <!-- Section for user inputs -->

            <section id="input-section">

                <h2>Input Parameters</h2>

                <form id="input-form">

                    <label for="num-frames">Number of Frames (1-10):</label>

                    <input type="number" id="num-frames" min="1" max="10" value="3"><br>

                    <label for="reference-string">Reference String (comma-separated):</label>

                    <input type="text" id="reference-string" placeholder="e.g., 1,2,3,2,4,1,3,5,4,1"><br>

                    <button type="submit">Start Simulation</button>

                </form>

            </section>

            <!-- Section for simulation controls (play, pause, etc.) -->

            <!-- It's hidden by default and shown with JavaScript after simulation starts -->

            <section id="controls-section" style="display: none;">

                <h2>Simulation Controls</h2>

                <button id="play-btn">Play</button>

                <button id="pause-btn">Pause</button>

                <button id="step-forward-btn">Step Forward</button>

                <button id="step-backward-btn">Step Backward</button>

                <!-- Animation speed slider -->

                <div class="slider-control">

                    <label for="speed-slider">Animation Speed:</label>

                    <input type="range" id="speed-slider" min="100" max="2000" value="1000" step="100">

                </div>

                <!-- Timeline slider for scrubbing through steps -->

                <div id="timeline-slider-container">

                    <label for="timeline-slider">Timeline:</label>

                    <input type="range" id="timeline-slider" min="0" max="0" value="0">

                    <span id="timeline-label">Step: 0</span>

                </div>

                <!-- Export buttons -->

                <button id="export-screenshot">Export Screenshot</button>

                <button id="export-trace">Export Execution Trace</button>

            </section>

        </div>

        <!-- ===== RIGHT PANEL ===== -->

        <!-- This column holds the visualization and statistics -->

        <div class="right-panel">

            <section id="visualization-section">

                <h2>Visualization</h2>

                <!-- The canvas element where JavaScript will draw the animation -->

                <canvas id="animation-canvas"></canvas>

                <!-- The display box for all statistics -->

                <div id="stats-display">

                    <h3>Execution Statistics</h3>

                    <p id="page-faults">Page Faults: 0</p>

                    <p id="page-hits">Page Hits: 0</p>

                    <p id="miss-ratio">Miss Ratio: 0%</p>

                    <p id="hit-ratio">Hit Ratio: 0%</p>

                </div>

            </section>

        </div>

    </main>

    <!-- JavaScript files are loaded at the end of the body for faster page load -->

    <!-- 'type="module"' allows using 'import' and 'export' in the JS files -->

    <script src="ui.js" type="module"></script>

    <script src="algorithm.js" type="module"></script>

    <script src="animation.js" type="module"></script>

    <script src="main.js" type="module"></script>

</body>

</html>

**Style.css**

/\* --- Global Styles --- \*/

/\* These styles apply to the entire page \*/

body {

    background-color: #0e0e10; /\* Dark background color \*/

    color: #e0e0e0; /\* Light grey text color for readability \*/

    font-family: 'Segoe UI', Roboto, Helvetica, Arial, sans-serif; /\* Modern, clean font stack \*/

    margin: 0;

    padding: 0;

    line-height: 1.6; /\* Standard line spacing \*/

}

h1, h2, h3 {

    color: #00c3ff; /\* Bright blue for all headings \*/

    text-shadow: 0 0 10px rgba(0, 195, 255, 0.6); /\* Adds a "glow" effect \*/

}

header {

    text-align: center;

    padding: 0.5rem; /\* Minimal padding to keep it short \*/

    background: linear-gradient(90deg, #121212, #1c1f26); /\* Subtle gradient background \*/

    border-bottom: 2px solid #00c3ff33; /\* Faint blue bottom border \*/

}

/\* Base style for the content "cards" (e.g., Input, Controls, Visualization) \*/

section {

    padding: 1rem 1.5rem;

    margin-bottom: 1rem;

    background-color: #1b1d22; /\* Slightly lighter than the body background \*/

    border-radius: 12px; /\* Rounded corners \*/

    box-shadow: 0 0 20px rgba(0, 195, 255, 0.1); /\* Subtle blue glow \*/

}

/\* --- Layout --- \*/

/\* This container establishes the main 2-column layout using flexbox \*/

#main-container {

    display: flex;

    flex-wrap: wrap; /\* Allows columns to stack on small screens \*/

    gap: 1.5rem; /\* Space between the left and right panels \*/

    padding: 1rem;

    align-items: flex-start; /\* Aligns columns to the top \*/

}

/\* Left column (for inputs and controls) \*/

.left-panel {

    flex: 1; /\* Takes 1 part of the available space \*/

    min-width: 300px; /\* Ensures it doesn't get too small \*/

}

/\* Right column (for visualization and stats) \*/

.right-panel {

    flex: 2.5; /\* Takes 2.5 parts, making it wider \*/

    min-width: 500px;

}

/\* --- Input Form --- \*/

#input-section label {

    display: block; /\* Makes labels appear on their own line \*/

    margin-top: 0.8rem;

}

#input-section input[type="number"],

#input-section input[type="text"] {

    width: 95%; /\* Makes text boxes almost full-width \*/

    padding: 0.5rem;

    border-radius: 6px;

    border: 1px solid #333;

    margin-top: 0.4rem;

    background-color: #26282d; /\* Dark input background \*/

    color: #fff;

    font-size: 1rem;

}

#input-section button {

    margin-top: 1.5rem;

    background: linear-gradient(45deg, #0078ff, #00c3ff); /\* Gradient for the main button \*/

    color: #fff;

    border: none;

    padding: 0.7rem 1.2rem;

    border-radius: 8px;

    font-size: 1rem;

    cursor: pointer;

    transition: 0.2s ease-in-out; /\* Smooth hover transition \*/

}

#input-section button:hover {

    transform: scale(1.05); /\* Slightly enlarge on hover \*/

    box-shadow: 0 0 10px #00c3ff;

}

/\* --- Simulation Controls --- \*/

#controls-section button {

    background: #22252a; /\* Dark button style \*/

    color: #00c3ff; /\* Blue text \*/

    border: 1px solid #00c3ff44; /\* Faint blue border \*/

    border-radius: 8px;

    padding: 0.6rem 1rem;

    margin: 0.3rem;

    cursor: pointer;

    transition: all 0.2s;

}

#controls-section button:hover {

    background: #00c3ff22; /\* Light blue glow on hover \*/

    box-shadow: 0 0 10px #00c3ff44;

}

.slider-control, #timeline-slider-container {

    margin-top: 1rem;

    display: flex;

    flex-direction: column; /\* Stack label on top of slider \*/

}

label[for="speed-slider"], label[for="timeline-slider"] {

    margin-bottom: 0.5rem;

}

#controls-section input[type="range"] {

    width: 100%;

    accent-color: #00c3ff; /\* Makes the slider thumb blue \*/

}

#timeline-label {

    margin-top: 0.3rem;

    color: #bbb; /\* Grey text for the step counter \*/

    text-align: center;

}

/\* --- Visualization Canvas --- \*/

#visualization-section {

    text-align: center;

}

#animation-canvas {

    display: block;

    width: 100%; /\* Canvas scales to fit its container \*/

    height: 360px; /\* Fixed height to help with layout alignment \*/

    margin: 1rem auto;

    border: 2px solid #00c3ff33;

    border-radius: 12px;

    background: radial-gradient(circle at top left, #14161b, #0d0f12); /\* Dark gradient background \*/

}

/\* --- Statistics Display --- \*/

#stats-display {

    background-color: #15171b;

    border-radius: 10px;

    padding: 1rem;

    margin-top: 1.5rem;

    box-shadow: 0 0 10px #00c3ff22;

    text-align: left; /\* Aligns text to the left inside the box \*/

}

#stats-display h3 {

    margin-top: 0;

}

#stats-display p {

    margin: 0.4rem 0;

    font-size: 1rem;

    color: #d6d6d6;

}

/\* Specific colors for different stats \*/

#page-faults { color: #ff5f5f; } /\* Red for faults \*/

#page-hits { color: #7cf57c; } /\* Green for hits \*/

#miss-ratio { color: #ff5f5f; } /\* Red for miss ratio \*/

#hit-ratio { color: #ffd95f; } /\* Yellow for hit ratio \*/

**Main.js**

/\*\*

 \* main.js

 \* \* This is the "brain" of the application.

 \* It connects all the other modules (ui, algorithm, animation)

 \* and sets up all the event listeners for the page.

 \*/

// --- Module Imports ---

// Import the form handler from the UI module

import { handleFormSubmit } from './ui.js';

// Import the core algorithm logic

import { runClockAlgorithm } from './algorithm.js';

// Import all the control functions from the animation module

import {

    animateSimulation,

    resetAnimation,

    play,

    pause,

    stepForward,

    stepBackward,

    setSpeed,

    jumpToStep,

    exportScreenshot,

    exportTrace

} from './animation.js';

// --- Event Listeners ---

// Wait until the entire HTML document is loaded and parsed

document.addEventListener('DOMContentLoaded', () => {

    // Find the input form

    const form = document.getElementById('input-form');

    // --- Form Submission Handler ---

    // This is the main trigger for the simulation

    form.addEventListener('submit', (e) => {

        e.preventDefault(); // Stop the page from reloading on form submit

        // Get the validated user inputs from the UI module

        const params = handleFormSubmit();

        // Only proceed if the inputs were valid (not null)

        if (params) {

            // Convert the reference string (e.g., "1,2,3") into an array of numbers

            const pages = params.referenceString.split(',')

                .map(s => parseInt(s.trim()));

            // Run the algorithm with the user's parameters.

            // This 'simulationData' object contains all the steps and stats.

            const simulationData = runClockAlgorithm(pages, params.numFrames);

            // Prepare the animation module for a new simulation

            resetAnimation();

            // Send the new data to the animation module to be drawn

            animateSimulation(simulationData);

            // Make the simulation controls (play, pause, etc.) visible

            document.getElementById('controls-section').style.display = 'block';

        } else {

            // If params were invalid, 'handleFormSubmit' already showed an alert

            console.error('Parameters invalid, simulation not started.');

        }

    });

    // --- Control Button Listeners ---

    // Hook up all the control buttons to their respective functions

    // from the animation.js module.

    document.getElementById('play-btn').addEventListener('click', play);

    document.getElementById('pause-btn').addEventListener('click', pause);

    document.getElementById('step-forward-btn').addEventListener('click', stepForward);

    document.getElementById('step-backward-btn').addEventListener('click', stepBackward);

    // Sliders send their value to the handler function

    document.getElementById('speed-slider').addEventListener('input', (e) => setSpeed(e.target.value));

    document.getElementById('timeline-slider').addEventListener('input', (e) => jumpToStep(parseInt(e.target.value)));

    // Export buttons

    document.getElementById('export-screenshot').addEventListener('click', exportScreenshot);

    document.getElementById('export-trace').addEventListener('click', exportTrace);

});

**Algorithm.js**

/\*\*

 \* algorithm.js

 \* \* Contains the core logic for the Clock Page Replacement Algorithm.

 \* This file does not interact with the DOM. It only performs calculations.

 \*/

/\*\*

 \* Executes the Clock Page Replacement algorithm.

 \* @param {number[]} pages - An array of page numbers (e.g., [1, 2, 3, 2, 4]).

 \* @param {number} frameCount - The total number of available frames in memory.

 \* @returns {object} A simulation data object containing all steps and final stats.

 \*/

export function runClockAlgorithm(pages, frameCount) {

    // --- Initialization ---

    // Represents physical memory frames, -1 is 'Empty'

    const frames = Array(frameCount).fill(-1);

    // The "Reference Bit" for each frame

    const useBits = Array(frameCount).fill(0);

    // The "clock hand" that points to the next frame to inspect

    let clockPointer = 0;

    let pageFaults = 0;

    let pageHits = 0;

    // Array to store the state at every single step for animation

    const steps = [];

    // Push the initial state (all frames empty) before processing any pages

    steps.push({

        frames: [...frames], // Use spread operator for a new copy

        useBits: [...useBits],

        pointer: clockPointer,

        page: null,      // No page being processed yet

        fault: false,

        faults: 0,

        hits: 0

    });

    // --- Process Each Page in the Reference String ---

    for (const page of pages) {

        let hit = false;

        let pageFoundIndex = -1; // To track \*which\* frame was a hit

        // 1. Check for a PAGE HIT

        for (let j = 0; j < frameCount; j++) {

            if (frames[j] === page) {

                // Page is already in memory!

                useBits[j] = 1; // Set the reference bit to 1

                hit = true;

                pageHits++;

                pageFoundIndex = j;

                break; // Exit loop once page is found

            }

        }

        // 2. Handle PAGE FAULT (if 'hit' is still false)

        if (!hit) {

            pageFaults++;

            // This is the core "Clock" logic:

            // Find a frame to replace by checking the use bit.

            while (true) {

                if (useBits[clockPointer] === 0) {

                    // Found a frame to replace (Use Bit is 0)

                    frames[clockPointer] = page;

                    useBits[clockPointer] = 1; // Set use bit for the new page

                    // Move pointer to the \*next\* frame for the next inspection

                    clockPointer = (clockPointer + 1) % frameCount;

                    break; // Exit the while loop, replacement is done

                } else {

                    // This frame was referenced (Use Bit is 1)

                    // Give it a "second chance": set its bit to 0

                    useBits[clockPointer] = 0;

                    // Move pointer to the next frame to inspect

                    clockPointer = (clockPointer + 1) % frameCount;

                }

            }

        }

        // 3. Store a snapshot of the current state for animation

        // This happens after every single page, whether it was a hit or a fault

        steps.push({

            frames: [...frames],      // Copy of the frames array

            useBits: [...useBits],    // Copy of the use bits array

            pointer: clockPointer,

            page: page,               // The page we just processed

            fault: !hit,              // Boolean, true if it was a fault

            hitIndex: pageFoundIndex, // -1 if fault, or the index of the hit

            faults: pageFaults,       // Cumulative fault count

            hits: pageHits            // Cumulative hit count

        });

    }

    // --- Final Statistics Calculation ---

    const totalRequests = pages.length;

    // Calculate final ratios (as percentages, fixed to 2 decimal places)

    const hitRatio = totalRequests > 0 ? ((pageHits / totalRequests) \* 100).toFixed(2) : 0;

    const missRatio = totalRequests > 0 ? ((pageFaults / totalRequests) \* 100).toFixed(2) : 0;

    // Return the complete simulation data object

    return {

        steps,

        pageFaults,

        pageHits,

        hitRatio,

        missRatio,

        totalSteps: steps.length

    };

}

**Animation.js**

/\*\*

 \* animation.js

 \* \* This is the "rendering engine." It's responsible for

 \* drawing the simulation data onto the HTML canvas.

 \* It also manages all the animation controls (play, pause, speed, etc.).

 \*/

// --- Module-Level Variables ---

let canvas, ctx; // Our drawing surface and its 2D context

let simulationData = null; // Will hold the 'steps' array from algorithm.js

let currentStep = 0; // The index of the step we are currently viewing

let animationInterval = null; // A timer ID for the 'play' function

let isPlaying = false;

let animationSpeed = 1000; // Default 1-second delay between steps

/\*\*

 \* Initializes the animation with new data.

 \* This is called by main.js after the "Start" button is clicked.

 \* @param {object} data - The full simulation data from algorithm.js.

 \*/

export function animateSimulation(data) {

    // Get the canvas element once

    if (!canvas) {

        canvas = document.getElementById("animation-canvas");

        ctx = canvas.getContext("2d");

    }

    // Set the canvas's internal bitmap size to match its display size.

    // This is CRITICAL to prevent the drawing from being blurry or pixelated.

    canvas.width = canvas.clientWidth;

    canvas.height = canvas.clientHeight;

    // Store the data and reset the view

    simulationData = data;

    currentStep = 0;

    // Set the timeline slider's max value to the number of steps

    const timeline = document.getElementById('timeline-slider');

    timeline.max = simulationData.steps.length - 1;

    timeline.value = 0;

    drawFrame(); // Draw the very first frame (initial state)

}

/\*\*

 \* Resets the entire visualizer to a clean slate.

 \* Called before a new simulation starts.

 \*/

export function resetAnimation() {

    pause(); // Stop any ongoing animation

    simulationData = null;

    currentStep = 0;

    if (ctx) { // Clear the canvas only if it exists

        ctx.clearRect(0, 0, canvas.width, canvas.height);

    }

    // Reset all the text in the stats box

    document.getElementById("page-faults").textContent = `Page Faults: 0`;

    document.getElementById("page-hits").textContent = `Page Hits: 0`;

    document.getElementById("miss-ratio").textContent = `Miss Ratio: 0%`;

    document.getElementById("hit-ratio").textContent = `Hit Ratio: 0%`;

    document.getElementById("timeline-label").textContent = `Step: 0`;

    document.getElementById("timeline-slider").value = 0;

    document.getElementById("timeline-slider").max = 0;

}

/\*\*

 \* Starts auto-playing the animation.

 \*/

export function play() {

    // Don't do anything if already playing or if there's no data

    if (isPlaying || !simulationData) return;

    isPlaying = true;

    // If at the end, restart from the beginning

    if (currentStep >= simulationData.steps.length - 1) {

        currentStep = 0;

    }

    // Start a timer that advances the frame

    animationInterval = setInterval(() => {

        if (currentStep < simulationData.steps.length - 1) {

            currentStep++;

            drawFrame(); // Advance and draw

        } else {

            pause(); // We've reached the end, so stop

        }

    }, animationSpeed); // 'animationSpeed' controls the delay

}

/\*\*

 \* Pauses the auto-play.

 \*/

export function pause() {

    clearInterval(animationInterval); // Stop the timer

    isPlaying = false;

}

/\*\*

 \* Manually moves one step forward.

 \*/

export function stepForward() {

    // Stop if at the end

    if (!simulationData || currentStep >= simulationData.steps.length - 1) return;

    pause(); // Always pause when stepping manually

    currentStep++;

    drawFrame();

}

/\*\*

 \* Manually moves one step backward.

 \*/

export function stepBackward() {

    // Stop if at the beginning

    if (!simulationData || currentStep <= 0) return;

    pause();

    currentStep--;

    drawFrame();

}

/\*\*

 \* Jumps to a specific step from the timeline slider.

 \* @param {number} step - The step index to jump to.

 \*/

export function jumpToStep(step) {

    if (!simulationData) return;

    pause();

    currentStep = parseInt(step); // Ensure the value is a number

    drawFrame();

}

/\*\*

 \* Sets the animation speed from the speed slider.

 \* @param {number} value - The raw value from the slider (100 to 2000).

 \*/

export function setSpeed(value) {

    // The slider is "speed" (higher = faster)

    // The interval is "delay" (lower = faster)

    // So we invert the value. (2100 - 2000 = 100ms, 2100 - 100 = 2000ms)

    animationSpeed = 2100 - parseInt(value);

    // If we're currently playing, restart the interval with the new speed

    if (isPlaying) {

        pause();

        play();

    }

}

/\*\*

 \* This is the master function that draws everything on the canvas

 \* for the 'currentStep'.

 \*/

function drawFrame() {

    // Safety check, don't draw if data isn't ready

    if (!simulationData || !ctx) return;

    const state = simulationData.steps[currentStep]; // Get the data for this step

    ctx.clearRect(0, 0, canvas.width, canvas.height); // Clear the canvas

    // --- Dynamic Sizing Logic ---

    // This logic dynamically scales the frames to fit the canvas

    const numFrames = state.frames.length;

    const baseFrameWidth = 80;

    const baseGap = 40;

    const padding = 50; // Keep 50px padding on the sides

    const frameHeight = 80; // Frames are always 80px tall

    let frameWidth = baseFrameWidth;

    let gap = baseGap;

    // Calculate the total width our frames \*want\* to be

    let totalWidth = (numFrames \* frameWidth) + ((numFrames - 1) \* gap);

    // If that width is too big for the canvas, we need to scale down

    if (totalWidth > canvas.width - padding) {

        const scale = (canvas.width - padding) / totalWidth;

        frameWidth \*= scale; // Scale down the width

        gap \*= scale; // Scale down the gap

    }

    // Recalculate the centered starting X position

    totalWidth = (numFrames \* frameWidth) + ((numFrames - 1) \* gap);

    const startX = (canvas.width - totalWidth) / 2;

    // Vertically center the frames

    const startY = (canvas.height - frameHeight) / 2 + 20;

    // --- Draw Frames ---

    // Loop through each frame in the current state

    state.frames.forEach((page, index) => {

        const x = startX + index \* (frameWidth + gap);

        // Default color

        ctx.strokeStyle = "#00ccff"; // Standard blue/cyan

        // --- Color Coding Logic ---

        if (currentStep > 0 && state.fault) {

            // A fault just happened. We need to highlight the frame that was \*replaced\*.

            // The 'pointer' has already moved \*past\* the replaced frame,

            // so we look at the index \*before\* the current pointer.

            const replacedIndex = (state.pointer === 0) ? numFrames - 1 : state.pointer - 1;

            if (replacedIndex === index) {

                ctx.strokeStyle = "#ff5f5f"; // Red = Replaced

            }

        } else if (currentStep > 0 && !state.fault && state.hitIndex === index) {

            // A hit just happened. Highlight the frame that was hit.

            ctx.strokeStyle = "#7cf57c"; // Green = Hit

        }

        // Draw the frame box

        ctx.lineWidth = 3;

        ctx.fillStyle = "#1e293b";

        ctx.shadowBlur = 10;

        ctx.shadowColor = ctx.strokeStyle; // Use the (potentially modified) stroke color for the glow

        ctx.beginPath();

        ctx.roundRect(x, startY, frameWidth, frameHeight, 10); // Draw a rounded rectangle

        ctx.fill();

        ctx.stroke();

        ctx.shadowBlur = 0; // Reset shadow

        // Draw the Page Number (e.g., "5" or "-")

        ctx.fillStyle = "#ffffff";

        ctx.font = `${Math.min(24, frameWidth \* 0.4)}px Arial`; // Scale font size

        ctx.textAlign = "center";

        ctx.fillText(page === -1 ? "-" : page, x + frameWidth / 2, startY + frameHeight / 2 + 8);

        // Draw the Use Bit (e.g., "R:1")

        ctx.fillStyle = "#93c5fd";

        ctx.font = `${Math.min(16, frameWidth \* 0.25)}px monospace`; // Scale font size

        ctx.fillText(`R:${state.useBits[index]}`, x + frameWidth / 2, startY + frameHeight + 20);

    });

    // --- Draw Clock Pointer (Yellow Arrow) ---

    // The pointer points to the \*next\* frame to be inspected

    const pointerX = startX + state.pointer \* (frameWidth + gap) + frameWidth / 2;

    const pointerY = startY - 40;

    ctx.fillStyle = "#facc15";

    ctx.beginPath();

    ctx.moveTo(pointerX, pointerY);

    ctx.lineTo(pointerX - 10, pointerY - 20);

    ctx.lineTo(pointerX + 10, pointerY - 20);

    ctx.closePath();

    ctx.fill();

    // --- Draw Top Status Text (e.g., "HIT on Page 2") ---

    ctx.font = "24px Arial";

    ctx.textAlign = "center";

    if (state.page !== null) { // 'page' is null only on step 0

        ctx.fillStyle = state.fault ? "#ff5f5f" : "#7cf57c";

        ctx.fillText(`${state.fault ? 'FAULT' : 'HIT'} on Page ${state.page}`, canvas.width / 2, 50);

    } else {

        ctx.fillStyle = "#facc15";

        ctx.fillText("Initial State", canvas.width / 2, 50);

    }

    // --- Update UI Elements ---

    updateStats(state);

    updateTimeline();

}

/\*\*

 \* Updates the statistics display (faults, hits, ratios) in the HTML.

 \* @param {object} state - The current simulation step object.

 \*/

function updateStats(state) {

    // Live update of counts based on the current step's data

    document.getElementById("page-faults").textContent = `Page Faults: ${state.faults}`;

    document.getElementById("page-hits").textContent = `Page Hits: ${state.hits}`;

    // Only show final ratios at the very last step

    if (currentStep === simulationData.steps.length - 1) {

        document.getElementById("miss-ratio").textContent = `Miss Ratio: ${simulationData.missRatio}%`;

        document.getElementById("hit-ratio").textContent = `Hit Ratio: ${simulationData.hitRatio}%`;

    } else {

        // Show N/A during simulation as ratios are misleading mid-run

        document.getElementById("miss-ratio").textContent = `Miss Ratio: N/A`;

        document.getElementById("hit-ratio").textContent = `Hit Ratio: N/A`;

    }

}

/\*\*

 \* Updates the timeline slider and step label to match the current step.

 \*/

function updateTimeline() {

    document.getElementById("timeline-slider").value = currentStep;

    document.getElementById("timeline-label").textContent = `Step: ${currentStep}`;

}

/\*\*

 \* Exports the current canvas state as a PNG image.

 \*/

export function exportScreenshot() {

    if (!canvas) return; // Don't export if canvas isn't ready

    const link = document.createElement('a');

    link.download = 'clock-algorithm-snapshot.png';

    link.href = canvas.toDataURL(); // Convert canvas to Base64 image data

    link.click(); // Programmatically click the link to trigger download

}

/\*\*

 \* Exports the entire simulation trace (all steps) as a TXT file.

 \*/

export function exportTrace() {

    if (!simulationData) return;

    let traceContent = "Clock Algorithm Execution Trace\n===================================\n";

    // Build a string by looping through every step

    simulationData.steps.forEach((step, index) => {

        traceContent += `Step ${index}:\n`;

        traceContent += `  - Referencing Page: ${step.page === null ? 'N/A' : step.page}\n`;

        traceContent += `  - Result: ${index === 0 ? 'Initial State' : (step.fault ? 'Page Fault' : 'Page Hit')}\n`;

        traceContent += `  - Frames: [${step.frames.join(', ')}]\n`; // Show frame contents

        traceContent += `  - Use Bits: [${step.useBits.join(', ')}]\n`; // Show use bits

        traceContent += `  - Clock Pointer at index: ${step.pointer}\n\n`;

    });

    // Create a "Blob" (Binary Large Object) from the text string

    const blob = new Blob([traceContent], { type: 'text/plain' });

    // Create a temporary link to download the blob

    const link = document.createElement('a');

    link.download = 'clock-algorithm-trace.txt';

    link.href = URL.createObjectURL(blob);

    link.click();

    URL.revokeObjectURL(link.href); // Clean up the temporary URL

}

**Ui.js**

/\*\*

 \* ui.js

 \* \* Handles all interactions with the UI input form.

 \* Its primary job is to read and validate user input.

 \*/

/\*\*

 \* Reads and validates the simulation parameters from the input form.

 \* @returns {object|null} An object {numFrames, referenceString} if inputs are valid, or null if invalid.

 \*/

export function handleFormSubmit() {

    // Read the raw values from the input fields

    const numFrames = document.getElementById('num-frames').value;

    const referenceString = document.getElementById('reference-string').value;

    // --- Validation ---

    // 1. Check if fields are empty

    if (!numFrames || !referenceString) {

        console.error('Missing required fields');

        alert('Please fill all fields.');

        return null; // Stop execution and return null

    }

    // 2. Validate the reference string format

    const referenceArray = referenceString.split(',').map(item => {

        // Trim whitespace and convert to a number

        const num = parseInt(item.trim());

        // If conversion fails (e.g., "a" or ""), return null

        return isNaN(num) ? null : num;

    });

    // 3. Check if any part of the string was invalid

    if (referenceArray.includes(null)) {

        console.error('Invalid reference string');

        alert('Reference string must be comma-separated numbers.');

        return null; // Stop execution and return null

    }

    // If all checks pass, return the valid data

    return {

        numFrames: parseInt(numFrames),

        referenceString: referenceString // Return the original valid string

    };

}

**README**

**🕒 Clock Algorithm Page Replacement Visualizer**

**📘 Project Description**

**Clock Algorithm Page Replacement Visualizer** is a standalone, client-side web application that provides an **interactive, animated visualization** of the **Clock Page Replacement Algorithm**.  
Users can input a custom **page reference string** and **number of memory frames** to see a **step-by-step simulation** of how the algorithm manages page faults and hits.

Built entirely using **HTML, CSS, and JavaScript**, the app leverages the **HTML Canvas API** for real-time animations.  
The architecture is fully modular, separating **algorithm logic**, **canvas animation**, and **UI event handling** for clarity and maintainability.

**⚙️ Feature Overview**

**🔢 Interactive Parameter Input**

* Set the number of available memory frames.
* Enter a custom page reference string.

**🎬 Full Animation Controls**

* **Play**, **pause**, **step forward**, and **step backward** through the simulation.

**🕓 Configurable Speed**

* Adjust the animation speed with an intuitive slider.

**🎨 Real-Time Canvas Visualization**

* Displays all memory frames and their currently loaded pages.
* Shows the **Use Bit (Reference Bit)** for each frame.
* Animates the **Clock Pointer** to indicate which frame is being inspected.

**📊 Step-by-Step Timeline**

* Jump to any specific step in the simulation using the timeline slider.

**📈 Detailed Statistics**

* Track **page faults**, **page hits**, and final **hit/miss ratios**.

**📤 Export Functionality**

* **Screenshot Export:** Save a PNG image of the current visualization.
* **Execution Trace Export:** Download a .txt file logging each step, page request, fault/hit status, and memory state.

**🧰 Technology Stack**

| **Technology** | **Purpose** |
| --- | --- |
| **HTML5** | Structures the web application |
| **CSS3** | Styles the layout, controls, and visual components |
| **JavaScript (ES6+)** | Implements application logic and algorithm behavior |
| **HTML Canvas API** | Renders animations and visualizations in real time |

**🧱 Code Architecture**

The project follows a **modular architecture** for readability and scalability:

| **File** | **Description** |
| --- | --- |
| index.html | Main HTML file defining the page structure |
| style.css | Contains all styling for layout and visuals |
| main.js | Entry point connecting UI events to logic |
| ui.js | Handles input reading and validation |
| algorithm.js | Core implementation of the Clock Page Replacement Algorithm; generates step data for visualization |
| animation.js | Renders and animates each simulation step using the Canvas API |

**🚀 Getting Started**

1. Clone the repository

git clone https://github.com/your-username/clock-page-replacement-visualizer.git

**GUIDE**

**🧭 Execution Guide: Clock Algorithm Visualizer**

This one-page guide explains how to **run the application** and **understand its features**.

**⚙️ 1. Setup Instructions (Running Locally)**

This is a **standalone, client-side application** — no web server or special setup required.

**Steps:**

1. Ensure all project files are in the same folder:
2. index.html, style.css, main.js, ui.js, algorithm.js, animation.js
3. Double-click **index.html**.
4. The application will automatically open and run in your **default web browser**.

**🧩 2. User Interface Guide**

The interface is divided into **two main panels**:

**🖱️ Left Panel — Inputs & Controls**

| **Control** | **Description** |
| --- | --- |
| **Number of Frames** | Set the total available memory frames (e.g., 3). |
| **Reference String** | Enter the sequence of page requests, separated by commas (e.g., 1,2,3,2,4,1,3,5). |
| **Start Simulation** | Validates inputs and starts the simulation. |
| **Play / Pause** | Starts or stops the automatic step-by-step animation. |
| **Step Fwd / Step Back** | Move one step forward or backward manually. |
| **Animation Speed** | Adjusts delay between animation steps using a slider. |
| **Timeline** | Shows current progress; drag to jump to any point in the simulation. |
| **Screenshot** | Exports a PNG image of the current visualizer state. |
| **Export Trace** | Saves a .txt file logging the state of memory at each step. |

**📊 Right Panel — Visualization & Statistics**

| **Section** | **Description** |
| --- | --- |
| **Visualization** | The main canvas where the Clock algorithm is animated. |
| **Execution Statistics** | Displays real-time performance metrics such as page hits, faults, and ratios. |

**🎞️ 3. Animation Features & Color Coding**

**🧱 Frames (Boxes)**

* Represent physical memory frames.
* Each box shows a **Page ID**.

**🔁 Reference Bit (Use Bit)**

* Shown as **R:1** or **R:0** on each frame.
  + **R:1** → The page has been referenced (Use Bit = 1).
  + **R:0** → The page has not been recently referenced (Use Bit = 0).

**🟡 Clock Pointer**

* A **yellow arrow** indicates the next frame to inspect during a page fault.

**🧾 Status Text**

* Displays messages like **“HIT on Page X”** or **“FAULT on Page X”** above the canvas.

**🎨 Color Coding**

| **Color** | **Meaning** |
| --- | --- |
| **Green Frame** | Page **Hit** — the requested page is found in memory. |
| **Red Frame** | Page **Fault** — a page replacement occurs here. |
| **Cyan Frame** | Neutral / default state. |

**📈 Statistics**

| **Metric** | **Description** |
| --- | --- |
| **Page Faults** | Running total of page faults during simulation. |
| **Page Hits** | Running total of page hits. |
| **Miss Ratio / Hit Ratio** | Final percentages displayed after simulation ends (shown as *N/A* during execution). |

**🌐 4. Browser Requirements**

The visualizer runs on **any modern browser** that supports **HTML5**, **CSS3**, and **ES6 JavaScript**.

**✅ Recommended Browsers**

* **Google Chrome** (v80+)
* **Mozilla Firefox** (v78+)
* **Microsoft Edge** (v80+)
* **Apple Safari** (v14+)

💡 *No internet connection is required to run the application.*