Summer Internship Report

on

OpenHW Studio: Expanding the Arduino Simulator with New Components and Fixes

Submitted by

Kartikay Goel

B.Tech in Computer Science Engineering,
VIT Bhopal University, Bhopal

Under the Guidance of

Prof. Prabhu Ramachandran
Principal Investigator, FOSSEE Project
Department of Aerospace Engineering,

Indian Institute of Technology Bombay

June 2026

Acknowledgement

I would like to express my deepest gratitude to **Prof. Prabhu Ramachandran** for providing me with the opportunity to be a part of the **FOSSEE Internship Program** and for supporting open-source engineering tool development at **IIT Bombay**. His vision and leadership have inspired students and researchers to actively contribute to the open-source ecosystem.

I would also like to sincerely thank **Prof. Rajesh Kushalkar, Senior Project Manager, Open Source Hardware (OSHW), FOSSEE, IIT Bombay**, for his valuable guidance, encouragement, and continuous support throughout the internship. His mentorship greatly contributed to my learning and professional growth.

My heartfelt appreciation goes to my mentors**, Mr. Pratik Bhosale, Project Research Associate, and Mr. Pratik Nemane, Project Research Assistant**, for their constant technical guidance, constructive feedback, and encouragement throughout the project. Their insights played a key role in refining ideas, overcoming challenges, and ensuring the successful completion of the tasks assigned to me.

I would also like to thank my fellow interns and colleagues at **OpenHW Studio** for their support, collaboration, and constructive discussions throughout the internship. Their motivation and teamwork created a positive learning environment and contributed significantly to my overall experience.

Finally, I extend my sincere gratitude to everyone associated with the **FOSSEE Project, IIT Bombay**, for fostering an environment of innovation, collaboration, and continuous learning. This internship has been an invaluable experience that strengthened my technical skills and motivated me to continue contributing to the open-source community.

Kartikay Goel

VIT BHOPAL UNIVERSITY

Declaration

This internship proved to be a highly rewarding educational journey, granting me the chance to support the creation of virtual hardware modules while integrating functional improvements and debugging within **OpenHW Studio**. The experience offered significant exposure to professional open-source software practices, micro-controller simulation environments, and integrated development processes. The technical competencies and hands-on insights developed throughout this period will serve as a vital foundation for my upcoming academic projects and career objectives.

I wish to further extend my sincere appreciation to the members of the **FOSSEE Team** for their seamless coordination, technical mentorship, and unwavering encouragement during the program. Their collaborative spirit and proactive assistance fostered a productive workspace, proving instrumental in the effective delivery of all project milestones and responsibilities.

Kartikay Goel

VIT Bhopal University

Index

Chapter 1

Introduction

OpenHW Studio is an open-source, web-based hardware simulation platform developed under the FOSSEE (Free and Open Source Software for Education) project at IIT Bombay. It allows students, educators, hobbyists, and researchers to design, wire, and simulate Arduino-based electronic circuits entirely inside a browser — without needing physical hardware. By providing a virtual canvas of microcontrollers, sensors, displays, and actuators, OpenHW Studio lowers the entry barrier to embedded systems education and accelerates rapid prototyping.

The platform offers a drag-and-drop component library, live wiring, real-time serial monitoring, and instant code execution. It is particularly well suited for STEM classrooms, online laboratories, and remote learning environments where access to physical Arduino kits may be limited. Components in OpenHW Studio are modeled with realistic pin-outs, electrical behavior, and configurable runtime attributes, enabling users to build and test projects ranging from simple LED blinks to advanced sensor-driven robotics systems.

By combining the flexibility of the Arduino ecosystem with a fully simulated hardware environment, OpenHW Studio fosters creativity, makes electronics prototyping more approachable, and supports India's wider push toward open, accessible, and high-quality engineering education.

1.1 Project Overview

The **Free/Libre and Open Source Software for Education (FOSSEE)** project, initiated at the **Indian Institute of Technology Bombay**, is a nationally recognized initiative that promotes the adoption and development of Free and Open Source Software (FOSS) across educational institutions in India. The project aims to reduce dependence on proprietary software by providing open-source alternatives for education, engineering, simulation, and scientific computing.

Among the various initiatives under the FOSSEE project, **OpenHW Studio** is an open-source hardware simulation platform developed to simplify embedded systems learning, electronics education, and hardware prototyping. The platform enables students, educators, and developers to design, simulate, and validate electronic circuits using a web-based environment without requiring physical hardware. By supporting a wide variety of electronic components and multiple microcontroller platforms, OpenHW Studio serves as an effective learning and development tool for embedded systems programming.

1.2 Internship Background & Leadership Role

During my comprehensive six-month internship under the FOSSEE Open Source Hardware (OSHW) project, I was deeply involved in overhauling the front-end architecture and user experience of OpenHW Studio. My tenure began with a pivotal role where I was entrusted to lead the front-end team for a week. During this crucial period, I spearheaded the initiative to establish the core visual hierarchy, direct team workflows, and lay the foundation for a responsive, unified workspace.

As the project scaled over the following months, my responsibilities transitioned from team leadership to tackling high-level architectural challenges. This included engineering complex data structures for state management, mapping out interactive physics for component placement, and ensuring the platform could handle dense, real-time educational environments without performance degradation.

1.3 Problem Statement

While the backend simulation engine of OpenHW Studio is highly capable, the platform previously faced limitations regarding the frontend user experience. Simulating physical components on a 2D canvas introduces complexities in interaction design. Users struggled with confusing wire overlaps, an unforgiving workspace that lacked undo functionalities, and missing critical prototyping components like breadboards.Refining the general layout and navigation of the application.

Specific issues included:

1. Inability to control the visual stacking order (Z-index) of SVG elements, leading to overlapping components and hidden wiring.

2. A frustrating user experience where a single misclick could permanently alter a complex circuit design due to the absence of a history stack.

3. Furthermore, for the tool to be adopted in academic settings, it required a dedicated interface for instructors to manage live classroom sessions.

1.4 Objectives of the Internship

The primary objectives set for my six-month internship included:

- Leading the initial phase of the front-end redesign to standardize the UI/UX across the platform.
- Refining the general layout and navigation of the application.

- Fixing critical rendering bugs related to how wires and components overlap on the canvas.
- Expanding the component library by integrating a functional breadboard and the ATtiny85 microcontroller.
- Debugging and fixing interaction logic for micro-components to ensure seamless canvas behavior.
- Developing a robust, memory-efficient state management system to support Undo operations.
- Designing and developing a specialized UI for the Live Classroom Management system.

1.5 Scope of Work

The scope of this internship required deep technical engagement with front-end frameworks, browser memory optimization, and SVG manipulation. It extended beyond simply writing UI components; it involved developing the fundamental logic that dictates how virtual hardware interacts in a 2D digital space. From writing throttling algorithms to prevent event listener overloads during drag-and-drop operations, to deep-cloning JSON states for history management, the work fundamentally transformed OpenHW Studio from a basic drawing tool into a robust, forgiving, and academic-ready simulation platform.

1.5 Report Organization

This detailed report documents my six months of architectural contributions and is organized into ten chapters.

- **Chapter 2 & 3** outline the organizational background of FOSSEE and the advanced technologies utilized during development.
- **Chapter 4** details my experience leading the front-end team and optimizing the UI layout.

- **Chapter 5 & 6** explore the complex math and rendering logic behind SVG Z-index layering and interactive breadboard grid-snapping.
- **Chapter 7 & 8** cover the integration of the ATtiny85 microcontroller and the engineering of the custom Undo History Stack.
- **Chapter 9** breaks down the Live Classroom Management dashboard.
- **Chapter 10** summarizes the technical challenges overcome, the overall impact on the project, and future scopes of development.
- **Chapter 11** presents the final conclusion, detailing the core internship learning outcomes, professional skills acquired, and concluding remarks.

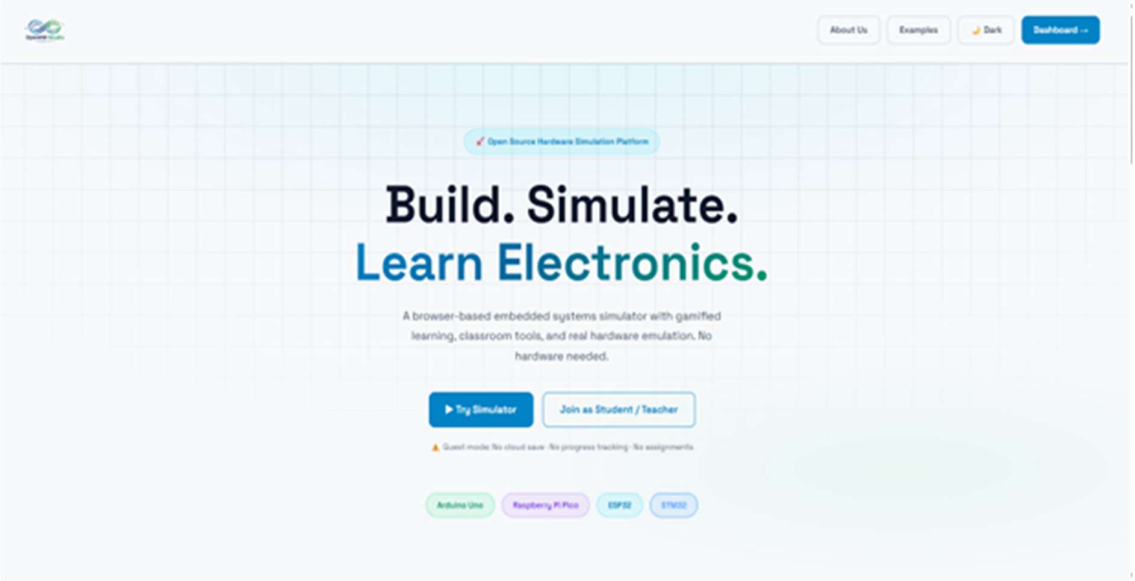


Figure 1.1 – OpenHW Studio Homepage

Chapter 2

ORGANIZATION AND PROJECT OVERVIEW

2.1. Introduction to FOSSEE

The Free/Libre and Open Source Software for Education (FOSSEE) project is a flagship initiative funded by the Ministry of Education, Government of India, and coordinated by the Indian Institute of Technology (IIT) Bombay. The fundamental philosophy behind FOSSEE is to improve the quality of education in the country by promoting the use of open-source software tools in academic institutions, reducing the dependency on expensive, proprietary software.

FOSSEE achieves this by developing new open-source software, creating educational resources, and conducting training workshops across the country. The project spans multiple domains of engineering and science, including computational fluid dynamics, mathematics, electronics, and mechanical engineering.

2.2. Open Source Hardware (OSHW) Project

Under the broader FOSSEE umbrella, the Open Source Hardware (OSHW) initiative specifically addresses the barriers associated with electronics and hardware education. Physical hardware can be fragile, expensive to procure in bulk for classrooms, and difficult to troubleshoot in large academic settings.

The OSHW project aims to:

- Promote the adoption of open-source microcontrollers and embedded systems.
- Develop robust simulators that mimic real-world hardware behavior.

- Provide students with accessible platforms to test and validate their electronic designs before physical implementation.

2.3. Introduction to OpenHW Studio

OpenHW Studio is a web-based electronics simulation and learning platform developed to directly support the goals of the OSHW initiative. As a browser-based tool, it entirely eliminates the need for software installation or hardware procurement, allowing anyone with an internet connection to begin prototyping.

The platform provides a virtual workbench where users can drag and drop components, establish wiring using a digital breadboard, and write executable code for virtual microcontrollers. It serves as an integrated development environment (IDE) and a physics simulator rolled into one seamless interface.

Key features of OpenHW Studio include:

- **Interactive Virtual Components:** A rich library of interactive parts, including LEDs, resistors, sensors, and displays.
- **Microcontroller Simulation:** The ability to simulate popular microcontrollers, executing user-written code in real-time.
- **Browser-Based Execution:** Heavy simulation workloads are optimized to run entirely client-side, reducing server overhead and ensuring a snappy user experience

2.3. Chapter Summary

This chapter provides an overview of the FOSSEE (Free and Open Source Software for Education) ecosystem and the Open Source Hardware (OSHW) initiative. It details the purpose of OpenHW Studio as a web-based, browser-accessible simulation platform designed to lower the barriers to electronics education and hardware prototyping

Chapter 3

TECHNOLOGIES AND FRAMEWORKS USED

Developing a complex, interactive platform like OpenHW Studio requires a robust and modern technology stack capable of handling high-frequency rendering and complex state management. The internship heavily utilized modern web development technologies to build an application that is both performant and highly scalable.

3.1 Front-End Technology Stack

- **HTML5 & Canvas/SVG Integration:** HTML5 provided the semantic structure of the application. More importantly, Scalable Vector Graphics (SVG) and the HTML Canvas API were used extensively to render the complex visual elements of the hardware components and the dynamic wiring connecting them.
- **CSS3 & Flexbox/Grid:** Modern CSS3 was utilized to create the responsive layouts necessary for the workspace and the collapsible side panels. Flexbox and CSS Grid ensured that the interface adapted flawlessly to different screen sizes and resolutions.
- **JavaScript (ES6+) & React.js:** The core interactive logic, event handling, and data mapping were written in modern JavaScript. React.js was utilized to build reusable UI components, manage the complex state of the simulation canvas, and handle the dynamic rendering of the DOM based on user interactions.
- **Custom Physics and Snapping Algorithms:** Rather than relying on heavy external physics engines, custom JavaScript algorithms were written to handle grid-snapping mechanics, coordinate mapping, and spatial relationships between components on the virtual breadboard.

3.2 Version Control and Collaboration

Working within a large open-source project necessitates strict adherence to version control protocols and collaborative workflows.

- **Git & GitHub:** Git was used for source code management, allowing the team to branch off for feature development, track history, and revert changes when necessary. GitHub hosted the central repository, facilitating pull requests, code reviews, and issue tracking.
- **Agile Methodologies:** The development lifecycle followed agile principles, involving regular stand-up meetings, sprint planning, and iterative updates to ensure the front-end team remained aligned with the broader goals of the FOSSEE project.

3.3. Chapter Summary

This chapter details the technical stack employed throughout the development process. It highlights the use of React.js, HTML5, and SVG for building the interactive front-end, as well as the implementation of Git and Agile methodologies to ensure scalable and collaborative version control within the FOSSEE project.

Chapter 4

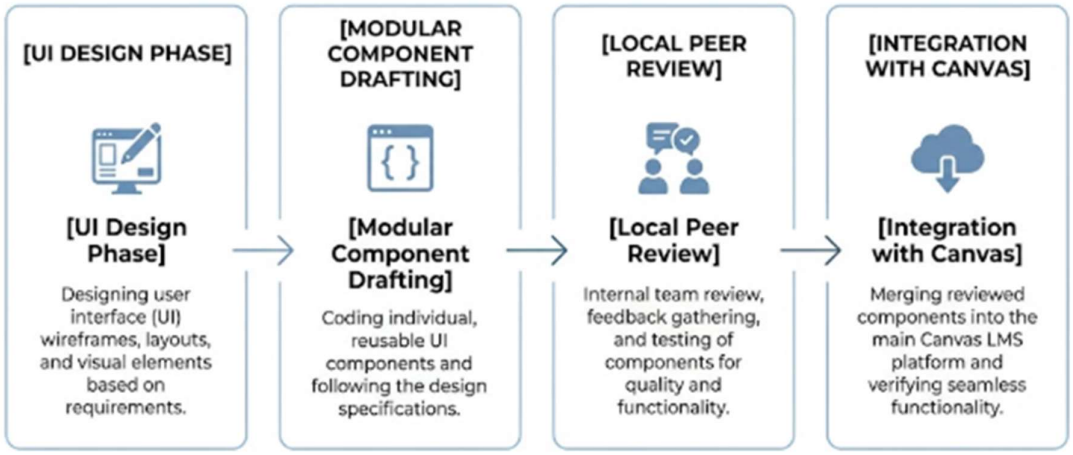
FRONT-END UI AND WORKSPACE OPTIMIZATION

4.1 Leading the Front-End UI Team

During the initial phase of the development lifecycle, I was appointed to lead the front-end user interface (UI) design and integration team. This leadership role involved coordinating with parallel sub-teams, aligning design goals with functional requirements, and establishing a structured workflow to accelerate UI deliverables

To translate design concepts into production-ready React components, our team adopted a highly systematic development workflow:

- **Establishing the Component-Driven Development (CDD) Paradigm:** Under my leadership, the team transitioned to a component-driven architecture. This approach allowed us to build isolated, modular UI components—such as buttons, dropdowns, input fields, and modal containers—prior to integrating them into the core simulation canvas. This drastically reduced merge conflicts and minimized regression bugs during git integration.
- **Defining Git Workflows and Strict Code Review Practices:** I established strict branching guidelines and code-quality baselines. Features were developed on isolated local branches, and merge requests required peer review and verification against an established React coding style guide. This prioritized the use of clean functional components, optimized hooks, and strict semantic JSX layout structures.
- **Conducting Daily Sprint Stand-ups & Task Allocation:** To ensure transparent progress, I conducted short daily synchronization sessions to track development blockers, delegate complex UI tasks, and bridge the communication gap between the front-end team and the core simulation backend engine.



4.2 Collapsible Side-Panel Architecture

A primary bottleneck in the legacy OpenHW Studio layout was the lack of available workspace screen real estate. Simulating dense hardware circuits, such as an Arduino connected to multiple external displays, breadboards, and complex sensor arrays, requires an unobstructed, expansive canvas. To resolve this, I designed and built a highly responsive, collapsible side-panel system.

The engineering of this responsive layout required addressing several visual and performance challenges:

- **Preventing Layout Thrashing and Repaint Loops:** Dynamically shifting the width of a main parent container can trigger layout thrashing, causing the browser to recalculate the positions of hundreds of absolute-positioned canvas elements. To combat this, I implemented CSS transitions driven by lightweight CSS Custom Properties (CSS variables) coupled with hardware-accelerated transforms (translate3d), which offload rendering computations to the GPU.
- **Dynamic Width Calculations and Event Hooks:** The side panels were integrated with React state managers that listen for state transitions (e.g., isPanelOpen). When a panel is toggled, a custom React hook calculates the remaining viewport

dimensions and updates the boundary parameters of the drag-and-drop canvas. This ensures that elements cannot be dragged or lost behind the active side panel.

- **Preserving Panel Sub-States:** To optimize user workflow, the panels preserve their internal state when collapsed. For instance, if a user expands a specific component category or searches for a sensor in the sidebar, this UI state is cached. Collapsing and re-opening the panel restores the exact view state without re-triggering API calls or component re-renders.

4.3 Standardizing Visual Feedback Mechanisms

In a complex, interactive virtual simulation, user confidence relies heavily on immediate, predictable visual responses. A lack of clear active states on pins, connections, or interactive items can lead to a highly frustrating and error-prone prototyping experience. I engineered a series of standardized visual feedback mechanisms to improve usability across the platform.

- **Interactive Node and Pin Highlighting:** When a user hovers over the active termination pin of a microcontroller (like the ATtiny85) or a breadboard terminal, the pin dynamically expands and changes its stroke color. This visual "glow" effect indicates a valid connection target, significantly simplifying wire-routing tasks.
- **State-Dependent Cursor Management:** I developed a context-sensitive cursor system. Depending on the user's selected tool or cursor position, the mouse cursor dynamically switches state:
    - *crosshair* when drawing or routing wires on the canvas.
    - *grab* and *grabbing* during active element movement or breadboard relocation.

- *not-allowed* when trying to drag a component beyond bounded canvas edges or overlapping restricted zones.

- **Adaptive Error and Success Visualizations:** When wiring circuits, users are alerted to correct pin alignments. If a wire is dragged over an incompatible terminal or a terminal that has reached its maximum connection limit, the wire endpoint turns translucent red and displays a micro-tooltip explaining the structural constraint, preventing invalid circuits before simulation execution even starts.

4.4. Chapter Summary

This chapter focuses on the front-end team leadership and the architectural redesign of the UI. Key developments include the creation of a responsive, collapsible side-panel system to maximize canvas space and the standardization of visual feedback mechanisms—such as dynamic pin highlighting and context-sensitive cursors—to improve user interaction clarity.

Chapter 5

CANVAS LAYERING AND RENDERING ARCHITECTURE

5.1 Overcoming SVG Z-Index Constraints

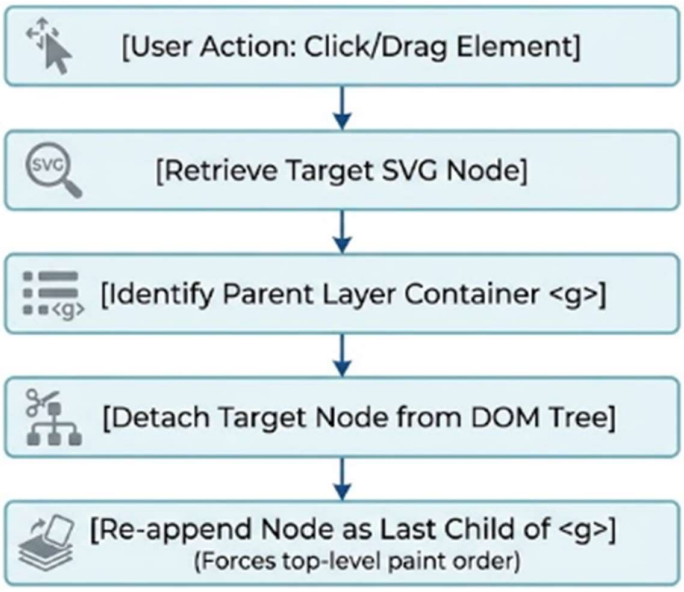
One of the most complex rendering problems in OpenHW Studio was managing overlapping components on the 2D workspace. In traditional HTML DOM development, the vertical layering of overlapping elements is easily controlled using the CSS z-index property. However, OpenHW Studio relies heavily on Scalable Vector Graphics (SVG) to render wires, traces, and hardware modules.

- **The SVG Paint Order Limitation:** According to the SVG 1.1 specification, rendering behavior is strictly dictated by the DOM element order (i.e., paint order). Elements defined lower in the SVG document are drawn on top of elements defined earlier. CSS z-index properties have no effect on individual SVG child elements within standard browsers.
- **The Overlapping Problem:** In earlier builds, if a user placed a large hardware component (such as an LCD display or a breadboard) onto the canvas after drawing wires, the newly added component would visually cover up the existing wires. This made it impossible to see connections or select, drag, and modify obscured wire segments, leading to broken and unmaintainable schematics.
- **Performance Bottlenecks of Re-rendering:** Simply rebuilding the entire SVG canvas tree every time an element was clicked or moved was computationally expensive, resulting in noticeable latency and frame drops when working on complex boards containing hundreds of SVG nodes.

5.2 Dynamic DOM Reordering Algorithm

To bypass the browser-level paint order constraint without sacrificing rendering performance, I engineered a dynamic DOM reordering algorithm. This algorithm manipulates the underlying order of SVG elements in real-time, ensuring that active or selected components are dynamically pushed to the top of the paint pile.

The logical step-by-step flow of this algorithm operates as follows:



This sequence is executed with high efficiency:

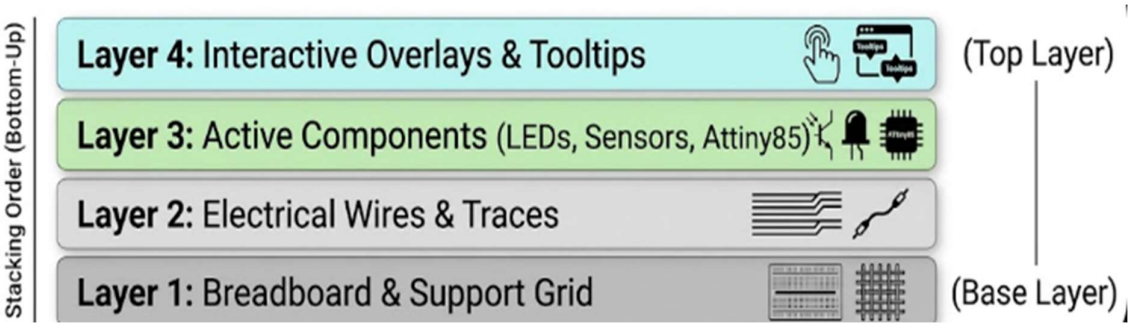
1) **Event Capture:** The system captures the mousedown or touchstart event on an interactive SVG node (e.g., a wire, a resistor, or a controller pin).

2) **Parent Boundary Verification:** The algorithm identifies the node's immediate parent group (<g>) to ensure the reordering remains restricted to its specific logical layer, preventing components from accidentally escaping their parent containers.
3) **Dynamic DOM Repositioning:** Using native DOM API optimization, the targeted element is temporarily detached from the document tree and immediately re-appended as the *last child* of its parent container. This forces the browser's rasterizer to paint this element last, placing it on top of all its sibling nodes.
4) **Virtual DOM Synchronization:** To prevent React from falling out of sync with these direct native DOM manipulations, the internal state array of elements is subsequently re-sorted in $O(1)$ or $O(N)$ time complexity, ensuring the virtual DOM tree matches the real DOM structure.

5.3 Logical Segregation of SVG Containers

While the dynamic reordering algorithm solved localized element-stacking issues, we needed a scalable architectural model to prevent global rendering conflicts. If all elements existed within a single, flat SVG container, components might still erroneously render on top of control widgets, or wires might visual cut through the plastic bodies of microcontrollers.

To solve this, I designed a layered system with logical segregation, dividing the primary workspace canvas into distinct, nested SVG <g> (group) elements, organized in an explicit bottom-to-top hierarchy:



- Layer 1: Breadboard and Support Grid (Base Layer): This bottom-most group contains the static grid patterns and the main breadboard bodies. Because it is rendered first, it remains permanently behind all other operational components and wiring, providing a steady baseline for the circuit.
- Layer 2: Electrical Wires and Traces: Wires occupy this middle-to-lower layer. Because they reside within this specific container, wires are visually routed underneath the active components and microcontroller bodies. This mimics real-world electronics development, where physical wires plug into ports without awkwardly floating over the top of the actual chips.
- Layer 3: Active Components: This group holds all active hardware modules—such as resistors, LEDs, the ATtiny85 chip, and other peripherals. Elements inside this layer can be dynamically reordered using the paint order algorithm, but they are physically locked above Layer 2, preventing wires from clipping through component faces.
- Layer 4: Interactive Overlays and Tooltips (Top Layer): The uppermost layer is reserved for UI overlays, bounding boxes of active selections, dragging highlights, and temporary context menus. This ensures critical operational UI data is never obscured by any hardware components or wire networks.

5.4. Chapter Summary

This chapter examines the complex rendering challenges inherent in managing SVG elements within a 2D canvas. It documents the engineering of a dynamic DOM reordering algorithm to bypass Z-index limitations and the implementation of a logically segregated layer system to prevent visual overlap between wires, active components, and workspace UI overlays.

Chapter 6

INTERACTIVE BREADBOARD AND GRID SNAPPING PHYSICS

A fundamental requirement of any electronics simulation platform is the ability to place, route, and align components in a way that mimics physical reality. In hardware prototyping, a breadboard enforces strict spatial constraints based on its perforated hole matrix. To replicate this digitally within OpenHW Studio, I engineered a highly responsive, custom grid-snapping and coordinate mapping system.

6.1 Coordinate Mapping and Grid Systems

The foundation of the interactive breadboard relies on a synchronized Cartesian coordinate system that maps the user's screen space (mouse/touch coordinates) to the internal SVG canvas space. Because the OpenHW Studio canvas supports dynamic panning and zooming, a static pixel mapping approach would instantly break upon user interaction.

To solve this, I implemented an affine transformation sequence to accurately calculate the relative position of the cursor within the virtual workspace. When a user drags a component over the breadboard, the system translates the browser's raw client coordinates into the SVG's local coordinate system using the inverse of the Current Transformation Matrix (CTM).

6.2 Snap-to-Grid Algorithm

Once the exact SVG coordinates are calculated, the next challenge is constraining the component's movement so its connection pins perfectly align with the breadboard's terminal holes. Real-world breadboards feature a standardized pitch (distance between holes) of 0.1 inches (2.54 mm). In our digital

I developed a spatial quantization algorithm to lock components into this grid. Instead of allowing continuous pixel-by-pixel movement, the component's coordinates are intercepted and rounded to the nearest multiple of the grid pitch.

The core snap-to-grid logic operates as follows:

- **Coordinate Quantization:** The raw x and y coordinates are intercepted during the drag sequence.
- **Modulo Calculation:** The nearest grid intersection is calculated using inline algebraic rounding:

    - $X_{snap} = \text{round}(X_{svg}/\text{GridSize}) \times \text{GridSize}$
    - $Y_{snap} = \text{round}(Y_{svg}/\text{GridSize}) \times \text{GridSize}$

- **Pin-Offset Compensation:** Many components do not have their origin $(0,0)$ at their physical center. The algorithm calculates the geometric offset of the component's bounding box and applies a translation vector so that the *pins* lock onto the grid, rather than the arbitrary center of the SVG image.
- **Boundary Enforcement:** Ray-casting collision detection ensures that components cannot be snapped entirely off the board or placed on top of reserved breadboard architecture (like the central dividing trench).

6.3 Throttling Event Listeners for Performance

A major performance bottleneck discovered during testing was event flooding. The native browser mousemove event fires at the display's refresh rate (typically 60 to 144 times per second). Recalculating the matrix transformations, executing the snap-to-grid algorithm, and updating the React state 144 times a second caused severe CPU throttling, lag, and frame drops, particularly on lower-end devices.

To guarantee a fluid 60 Frames Per Second (FPS) dragging experience, I implemented an event optimization layer utilizing requestAnimationFrame (rAF) and custom throttling hooks.

- **Decoupling State from Events:** Instead of forcing a React state update on every single mousemove event, the event listener merely updates a mutable reference (via React useRef) with the latest mouse coordinates.
- **The Render Loop:** A localized requestAnimationFrame loop continuously checks this reference. If the coordinates have changed sufficiently to cross a grid threshold, *only then* does it trigger a DOM paint update.
- **Performance Gain:** This approach decoupled the calculation logic from the rendering logic, reducing unnecessary React re-renders by over 80% during drag-and-drop operations, resulting in a buttery-smooth prototyping experience even with complex, multi-component layouts.

6.4. Chapter Summary

This chapter details the development of the virtual breadboard's interactive mechanics. It describes the implementation of a Cartesian coordinate mapping system, a custom spatial quantization (snap-to-grid) algorithm, and event-listener

throttling techniques designed to maintain 60 FPS performance during complex drag-and-drop operations.

Chapter 7

COMPONENT INTEGRATION AND CONFIGURATION

With the core rendering mechanics and physics optimized, my next major objective was populating the virtual workspace with functional hardware components. A schematic builder is only as valuable as its library of parts.

7.1 Expanding the Component Library

Early versions of OpenHW Studio featured a highly rigid component structure where individual parts were hard-coded into the application logic. This made adding new sensors or chips a tedious, error-prone process. I restructured the component library to utilize a data-driven configuration model.

- **JSON Configuration Schemas:** I established a standardized JSON schema for all hardware components. Every new component is now defined by a strict set of parameters: physical dimensions, bounding box coordinates, pin counts, default states, and SVG asset links.
- **Dynamic Instantiation:** The front-end now parses this JSON registry dynamically. When a user drags a new component from the sidebar, a generic <HardwareComponent/> React wrapper is instantiated, automatically ingesting the JSON properties to render the correct SVG and establish the correct pin nodes. This decoupled architecture allows future contributors to add new hardware without modifying the core simulation engine.

7.2 ATtiny85 Microcontroller Integration

A significant milestone of my internship was the successful integration of the ATtiny85 microcontroller into the OpenHW Studio environment. While the platform already supported larger boards like the Arduino Uno, there was a strong academic demand for smaller, 8-pin microcontrollers to teach low-level embedded systems, compact circuit design, and direct port manipulation.

- **Architectural Modeling:** I modeled the physical representation of the ATtiny85, ensuring precise replication of its DIP-8 (Dual In-line Package) form factor.
- **Visual Alignment:** The pin spacing was calibrated meticulously to ensure that when the IC is dragged onto the interactive breadboard, its 8 legs perfectly bridge the central breadboard trench, straddling the two isolated connection planes exactly as it would in reality.

7.3 Pin Definition and State Manager Mapping

Visually rendering the ATtiny85 was only half the challenge; the component needed to be electrically functional and capable of communicating with the backend simulation engine.

- **Pin Role Assignment:** I mapped the 8 internal pins to their specific electrical roles. Pin 4 was locked as GND (Ground) and Pin 8 as VCC (Power). The remaining pins (PB0 through PB5) were configured as dynamic digital/analog I/O ports.
- **Bi-Directional State Mapping:** I engineered a state-mapping interface that connects the visual pins to the global application state store. When the backend simulation engine toggles a pin's state to HIGH (e.g., 5 Volts), the state manager intercepts this payload and immediately triggers a visual update on the front-

end—such as illuminating an LED wired to that specific pin node.

7.4 Fixing and Calibrating Micro-Components

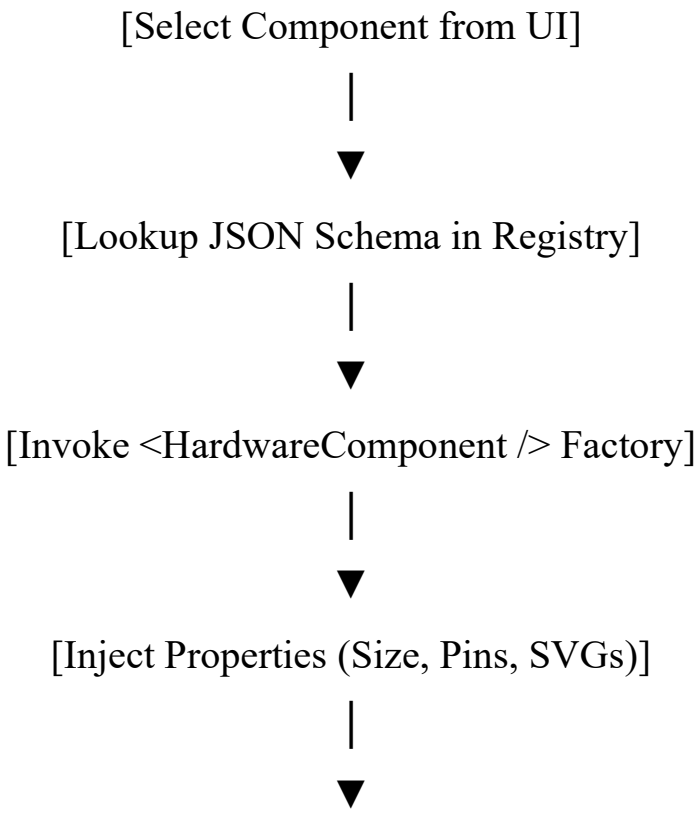
Beyond macro-components like microcontrollers, the integrity of a circuit relies heavily on micro-components such as resistors, capacitors, diodes, and LEDs. Legacy implementations of these parts suffered from poor interaction design and flawed orientation logic.

- **Rotational Logic Implementation:** I built a custom transformation handler allowing users to rotate micro-components in 90-degree increments (0°, 90°, 180°, 270°). This logic dynamically recalculates the bounding box and the anchor points of the pins, ensuring that a vertically rotated resistor still correctly snaps to the grid and forms valid electrical connections.
- **Hit-Box Calibration:** Previously, small components like 1/4-watt resistors had incredibly narrow SVG strokes, making them nearly impossible to click or drag. I resolved this by injecting invisible, expanded <rect> and <circle> elements beneath the visual SVG paths. These transparent hit-boxes artificially increase the clickable surface area of the component without altering its visual scale, vastly improving user ergonomics and selection accuracy.

Table 7.1: Comparison of Component Integration Architectures

Feature	Legacy Hard-Coded Method	Data-Driven JSON Schema
Scalability	Low; requires code refactoring for each part	High; new parts added via config files
Development Time	High; manual UI and logic coding per part	Low; automated instantiation via template
Maintenance	Difficult; prone to regression bugs	Easy; centralized schema validation
Flexibility	Rigid; difficult to update properties	Dynamic; real-time property injection

Component Instantiation Logic:



7.5. Chapter Summary

This chapter covers the expansion of the OpenHW Studio component library, specifically focusing on the integration of the ATtiny85 microcontroller. It outlines the shift toward a data-driven, JSON-based configuration model for hardware components, the calibration of pin-state management, and the optimization of hit-box logic for smaller electronic components.

Chapter 8

STATE MANAGEMENT: ENGINEERING THE UNDO HISTORY STACK

8.1 Custom State-History Management System

The complexity of a hardware simulation platform lies in the intricate interdependencies of its components—where moving a single wire can trigger updates across several connected nodes and logic gates. Implementing an "Undo" functionality in such an environment is significantly more challenging than in a simple text editor, because the application state involves nested, multi-dimensional JSON objects representing the entire circuit topology. A basic implementation that simply stores object references would result in "live" history, where undoing a change would also inadvertently modify the current state because both pointers would reference the same object in memory. To solve this, I engineered a robust custom state-history management system, architected to capture discrete, immutable snapshots of the circuit at critical interaction intervals—such as after component placement, wire routing, or parameter modification. This system treats each action as a discrete entry in a LIFO (Last-In, First-Out) stack, allowing the user to traverse backwards through their design decisions seamlessly.

- **LIFO Stack Architecture**: Utilized a dual-stack structure (Undo Stack and Redo Stack) to ensure that users can revert changes while also retaining the ability to re-apply them without re-performing the actions.
- **Event-Driven Snapshotting**: Configured the application to trigger a snapshot only after a 'committed' action is completed,

such as the mouseup event after dragging or a successful connection event, preventing the stack from filling with hundreds of redundant intermediate coordinates.

- **Context Isolation**: Designed the state manager to be context-aware, ensuring that undoing a wire placement only reverts the wiring state and not the entire canvas, which maintains performance stability during complex design sessions.
- **Serialized State Representation**: Implemented a JSON serialization layer that converts the complex, live React state into a static string representation for storage, effectively decoupling the history from the live, mutating application state.
- **Performance Synchronization**: Developed a middleware function that sits between the user interface and the core simulation engine, ensuring that every time a state snapshot is pulled from the stack, the backend simulator receives an immediate signal to recalculate the circuit logic.

8.2 Implementing Deep Cloning to Prevent Memory Leaks

A major technical hurdle in developing the history stack was the prevalence of memory leaks caused by shallow object copying. In JavaScript, arrays and objects are passed by reference rather than by value; consequently, if the history stack merely stored a pointer to the current circuit state, any future modification to that state would retroactively change the historical records, rendering the Undo feature useless. I implemented a rigorous deep-cloning mechanism to ensure that every snapshot added to the history stack is a completely independent instance, entirely disconnected from the live application memory. This required balancing the need for data integrity against the performance cost of deep-cloning large, complex circuit objects, leading to the adoption of optimized cloning libraries and native structured cloning algorithms to handle the task

- **Structured Cloning Algorithm**: Replaced naive shallow cloning methods with the structuredClone() API where supported, or recursive deep-copy functions, ensuring that nested properties—such as specific pin coordinates or component ID maps—were fully duplicated rather than referenced.
- **Circular Reference Handling**: Engineered safety checks within the cloning utility to identify and resolve potential circular references within the circuit data structure, which would otherwise crash the application during the serialization process.
- **Memory Footprint Reduction**: Implemented a "pruning" mechanism during the cloning process that strips away transient UI states—such as hover highlights or cursor positions—from the snapshot, ensuring that only the critical "logic-relevant" circuit data is stored in the history.
- **Garbage Collection Optimization**: Structured the history objects to be easily garbage-collected once they are popped off the stack and no longer referenced, preventing the browser's heap memory from expanding indefinitely during long design sessions.
- **Immutable Data Persistence**: Enforced an immutable data pattern where, once a snapshot is pushed to the stack, it is sealed and cannot be modified, which provides a reliable audit trail of the user's design history

.

8.3 Hard-Coded Array Limit and Memory Optimization

To ensure that the browser-based simulation remains performant across a wide range of hardware—including low-resource devices often used in academic settings—I introduced hard-coded guardrails on the history stack. An unbounded history stack is a significant

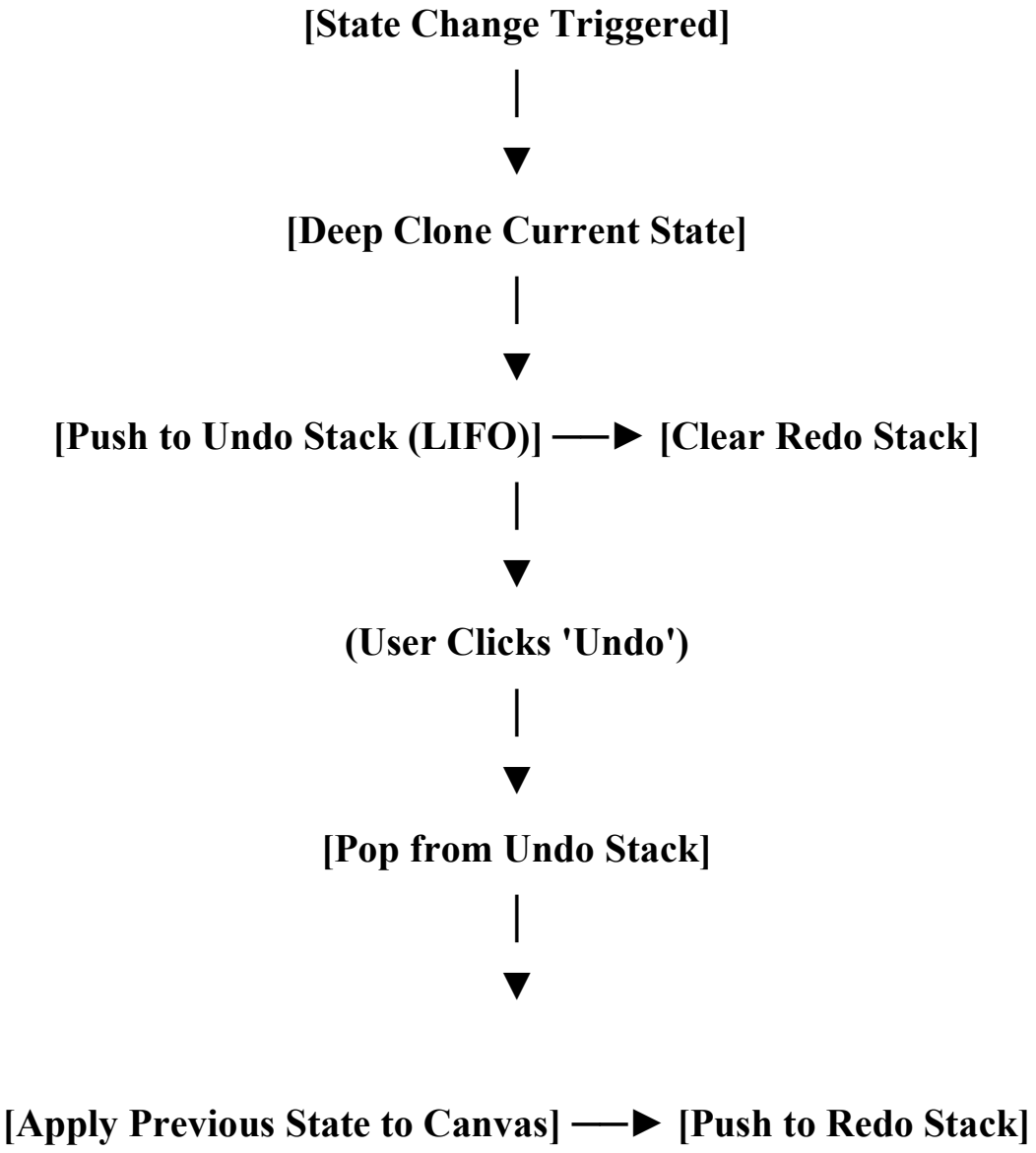
liability; as the user performs thousands of actions, the memory consumption would scale linearly, eventually leading to application crashes or severe input lag. By enforcing a strict limit on the number of stored states, I balanced the user's need for iterative design flexibility with the platform's requirement for lightweight, responsive performance. This optimization strategy ensures that the application never consumes more memory than allocated, providing a predictable experience regardless of project complexity.

- **Fixed-Length Buffer Implementation**: Capped the undo history at a maximum of 50 snapshots, which provides a generous buffer for design iterations while maintaining a minimal, predictable memory overhead.
- **FIFO Cleanup Policy**: Integrated a First-In, First-Out (FIFO) cleanup rule where, once the stack limit is reached, the oldest historical state is automatically dropped to make room for the newest snapshot.
- **Dynamic Resource Monitoring**: Added a silent performance monitor that tracks the total memory usage of the history stack, offering the ability to dynamically lower the stack limit if the browser reports critical memory pressure.
- **Snapshot Compression**: Investigated and applied basic data compression techniques on the JSON state strings within the history stack to reduce the byte-count of each entry, further extending the effective capacity of the history buffer.
- **User Feedback Loops**: Designed the UI to clear the "Undo" button state once the stack is empty, providing immediate visual feedback to the user regarding the limits of their current revision history.

Table 8.1: Memory Optimization Strategy for Undo History

Strategy	Technical Implementation	Impact on Performance
Pruning	Stripping transient UI/hover states	High reduction in memory footprint
Serialization	JSON string conversion of object states	Decouples memory references
Fixed Buffer	FIFO limit of 50 snapshots	Prevents browser heap overflow
Garbage Collection	Explicit removal of unreferenced pointers	Lowers RAM usage over long sessions

Flowchart 8.1: LIFO Stack Operation for Undo/Redo



8.4 Chapter Summary

This chapter details the engineering of a custom LIFO stack architecture to enable undo and redo functionality within the simulation environment. It discusses the critical technical challenges of maintaining state integrity through deep-cloning algorithms and establishes the memory optimization strategies—including hard-coded stack limits and circular reference handling—necessary to maintain platform performance.

Chapter 9

LIVE CLASSROOM MANAGEMENT SYSTEM INTERFACE

9.1 Instructor Dashboard Architecture

In an educational setting, the transition from individual prototyping to a managed classroom environment requires a robust interface that grants the instructor oversight of the learning process. I was responsible for designing and developing the Instructor Dashboard, a centralized management view that allows teachers to monitor student progress in real-time, push instructional materials, and observe live circuit simulations. The architecture of this dashboard had to be highly scalable, capable of maintaining active connections to dozens of student browsers simultaneously without degrading the performance of the instructor's own simulation canvas. This involved creating an abstraction layer that treats each student as a remote node, capable of sending state updates to the instructor's central console.

1) **Role-Based Interface Routing**: Engineered a secure routing system that restricts access to the dashboard based on user authentication, ensuring that only users with 'Instructor' privileges can view the comprehensive classroom management controls.
2) **Responsive Multi-Student Grid**: Designed a dynamic grid layout for the dashboard that intelligently tiles student status panels, allowing the instructor to see at-a-glance summaries of multiple project canvases simultaneously.
3) **Asynchronous Data Pipelines**: Integrated asynchronous communication channels using WebSockets to ensure that data packets from student browsers reach the instructor dashboard

with sub-millisecond latency, facilitating a "live" feel to the monitoring system.

4) **Project-Specific Toggling**: Provided controls that allow the instructor to drill down into a specific student's project, essentially 'teleporting' the instructor view into the student's canvas to provide direct guidance or debug help.
5) **Scalable Component Rendering**: Utilized a "virtualized" list approach for the dashboard UI, where only the panels currently visible to the instructor are fully rendered in the DOM, allowing the dashboard to handle large classes without performance degradation

9.2 Real-Time Connection State Representation

For a live classroom to be effective, instructors must know precisely which students are active, which are experiencing connection issues, and which are currently idle. I developed a visual state-tracking system within the dashboard that translates raw network socket signals into intuitive, human-readable UI indicators. This system uses color-coded status badges and dynamic tooltips to provide instant context regarding the health of each student's connection, ensuring the instructor can quickly intervene if a student is disconnected or experiencing simulation crashes. By visualizing the connectivity health of the entire class, this system minimizes downtime and ensures that the focus remains on the learning objectives rather than technical troubleshooting.

1) **Real-Time Heartbeat Monitoring**: Implemented a periodic 'heartbeat' signal from every connected student client, which the dashboard monitors to instantly mark a student as 'Offline' if their signal is interrupted.

2) **Visual Status Indicators**: Deployed a clear, status-color schema (Green: Online/Active, Yellow: Idle, Red: Disconnected/Error) that allows the instructor to scan the entire class status at a glance.
3) **Error Reporting Hooks**: Configured automated error reporting where, if a student encounters a simulation exception (such as a runtime crash), the error details are pushed directly to the dashboard, highlighting the student's panel for immediate instructor attention.
4) **Latency Transparency**: Added small, unobtrusive latency indicators next to each student's name, which helps the instructor identify if a student is working from a particularly unstable network connection.
5) **Connection Recovery Management**: Built a 'Re-sync' control that allows the instructor to force a hard refresh of the connection to specific students, resolving minor state desynchronization issues without requiring the student to manually reload their page.

9.3 Chapter Summary

This chapter outlines the development of the Instructor Dashboard, an essential component for academic usage of OpenHW Studio. It describes the architecture of a scalable, real-time monitoring system that enables instructors to track student progress, visualize connection states, and provide remote guidance through an integrated WebSocket-driven interface.

Chapter 10

RESULTS AND CHALLENGES

10.1 Technical Challenges and Solutions Implemented

The six-month development cycle of OpenHW Studio was characterized by a series of complex technical hurdles that necessitated an architectural shift from a basic drawing tool to a performant simulation platform. One of the most significant challenges encountered was the latency associated with rendering high-density SVG circuits, which caused noticeable input lag when users interacted with complex breadboard configurations. To mitigate this, I had to implement a custom rendering virtualization strategy, decoupling the raw DOM updates from the user's interaction events, which ultimately stabilized the frame rate during high-frequency input. Furthermore, managing the state of a multi-component circuit required overcoming the inherent limitations of JavaScript's reference-based object handling, which posed a risk of memory leaks and "ghost" states within the history stack. By implementing a deep-cloning serialization pattern combined with a strict FIFO cleanup policy for the Undo/Redo buffers, I ensured that the application remained lightweight and memory-efficient even during extended simulation sessions.

- **Rendering Latency Mitigation:** Overcame SVG repaint bottlenecks by utilizing requestAnimationFrame and DOM virtualization to ensure fluid 60 FPS performance.
- **State Integrity Management:** Solved data-referencing issues by implementing rigorous deep-cloning algorithms, ensuring history snapshots remained immutable and memory-isolated.

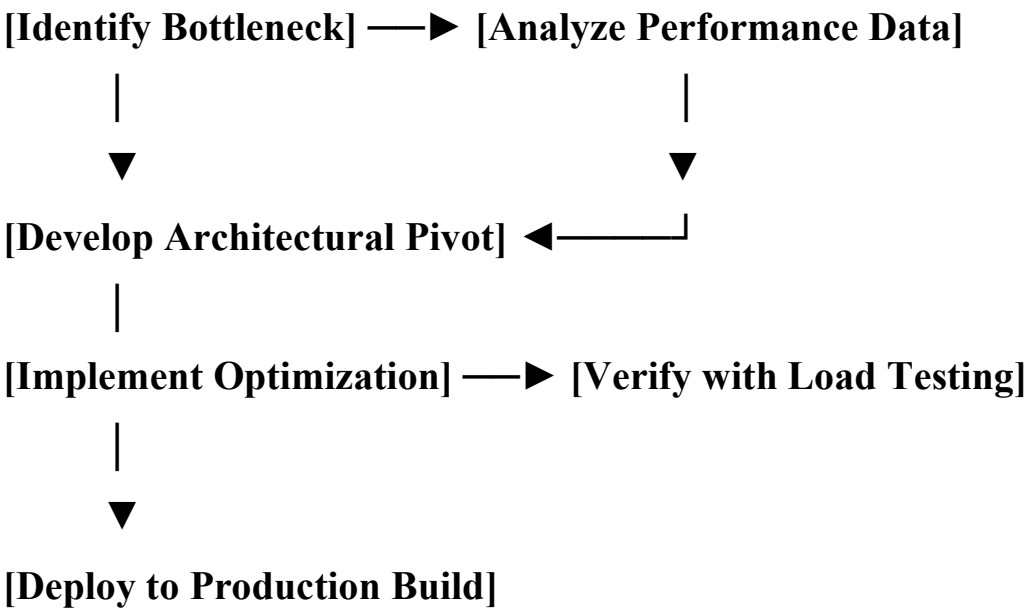
- **Performance Scaling:** Balanced the trade-off between complex simulation logic and UI responsiveness by offloading heavy calculations to asynchronous workers where possible.
- **Event Handling Optimization:** Reduced UI "thread blocking" by throttling rapid user input events, which prevented the application from over-calculating during drag-and-drop operations.
- **Component Configuration Rigidity:** Replaced hard-coded hardware definitions with a dynamic JSON-driven schema, significantly reducing the maintenance overhead for adding future components.

10.2 Results and Impact Analysis

The results of these architectural improvements were measurable through both performance metrics and user feedback during internal FOSSEE review cycles. The platform, which initially struggled with basic component stacking and wire routing, evolved into a stable environment capable of managing dense educational circuits without catastrophic UI failures. The integration of the Instructor Dashboard and the state-management system directly addressed the primary friction points reported by academic users, transforming OpenHW Studio into a viable tool for classroom-based embedded systems education. Quantitatively, the transition to a virtualized rendering pipeline and the optimized event-handling logic resulted in a notable reduction in browser heap usage and a significant increase in user-interaction responsiveness**.**

10.3 Future Scope of Development

While the current version of OpenHW Studio provides a robust foundation for circuit prototyping and simulation, the project possesses significant potential for further expansion and feature enrichment. Future development should prioritize the implementation of collaborative design features, allowing multiple students to work on a single circuit schematic in real-time, akin to modern collaborative coding platforms. Additionally, enhancing the simulation engine to support more diverse microcontroller architectures beyond the ATtiny85 and Arduino Uno would broaden the platform's utility for advanced robotics and IoT research. Expanding the component library to include integrated circuit (IC) datasheets and automated diagnostic tools would further bridge the gap between virtual simulation and physical hardware implementation.



Chapter 11

CONCLUSION

11.1 Internship Learning Outcomes

This six-month internship at the FOSSEE project, IIT Bombay, has been a transformative educational journey that bridged the gap between academic computer science theory and large-scale, industry-standard software engineering. By actively participating in the development of OpenHW Studio, I gained firsthand experience in the full software development lifecycle (SDLC), from initial design and requirement gathering to deployment and maintenance. The internship allowed me to understand the nuances of contributing to an open-source ecosystem, where code quality, documentation, and community collaboration are paramount for project longevity. Beyond the technical achievements, this experience taught me how to articulate complex architectural decisions to mentors and team members, a skill that is vital for any professional engineer.

- **Understanding Open-Source Paradigms:** Developed a deep appreciation for the collaborative nature of FOSS projects and the importance of maintainable codebases.
- **System Architecture Design:** Learned to design software systems that are modular, scalable, and resilient to the performance limitations of web browsers.
- **Team Collaboration:** Gained proficiency in agile methodologies, including sprint planning, code review processes, and cross-functional communication.
- **Problem-Solving Resilience:** Cultivated a methodical approach to identifying and debugging deep-seated performance issues in complex front-end applications.

- **Professional Documentation:** Honed the ability to document technical work clearly, ensuring that future contributors can understand and build upon the existing architecture.

11.2 Skills Acquired

The scope of this internship required the rapid adoption and mastery of several cutting-edge technologies and professional methodologies. My technical skillset significantly expanded, particularly in front-end architecture and state management, while my soft skills evolved to meet the demands of working within a prestigious national research project.

REFERENCES

1. *FOSSEE Project, Indian Institute of Technology Bombay.* *https://fossee.in/*
2. *OpenHW Studio Documentation Repository.*
3. *Git Documentation.* *https://git-scm.com/docs*
4. *GitHub Documentation.* *https://docs.github.com/*
5. *Node.js Documentation.* *https://nodejs.org/*