Introduction to WebAssembly: A Guide for 2nd-Year Computer Science Students

What is WebAssembly (WASM) and Why is it relevant to you?

WebAssembly (WASM) represents a paradigm shift in web development, serving as a safe, portable, low-level code format designed for efficient execution. Unlike traditional web technologies that rely solely on JavaScript, WebAssembly enables developers to run high-performance applications directly in browsers using languages like C, C++, Rust, and C#.

Think of WebAssembly as a universal translator for the web. While JavaScript has dominated browser programming for decades, WASM allows developers to bring the performance of systems programming languages to web applications. A 2024 benchmark by Wasmer demonstrated that WebAssembly modules could execute up to 20x faster than equivalent JavaScript code in compute-intensive tasks.

Originally designed for high-performance web applications, WASM is now expanding into server-side applications, IoT, and blockchain development, making it increasingly relevant for performance-critical computing scenarios.

As computer science students, you'll encounter numerous scenarios where WebAssembly becomes essential for performance-critical applications. Traditional JavaScript struggles with computationally intensive tasks like scientific computing, real-time data processing, and complex algorithms. WASM bridges this gap by enabling near-native performance.

Algorithmic implementations see particularly significant advantages. Sorting algorithms, graph algorithms, and numerical computations implemented in WASM can execute 5-20x faster than their JavaScript counterparts. This performance boost is crucial as web applications increasingly handle complex data processing traditionally confined to desktop software.

WebAssembly opens doors to:

- High-performance web applications competing with desktop software
- Cross-platform development with single codebases running on web, desktop, and server
- Legacy code modernization by compiling existing C/C++ libraries for web use
- Computationally intensive applications like cryptography, data analysis, and simulation tools
- Edge computing applications where performance and efficiency are critical

WebAssembly complements several Computer Science courses:

Data Structures and Algorithms: WASM enables implementing complex algorithms with near-native performance in web environments, applying your knowledge of sorting, graph traversal, and dynamic programming to high-performance web applications.

Computer Architecture and Systems Programming: WASM's stack-based virtual machine architecture and linear memory model directly reflect systems programming concepts, making performance and memory usage easier to understand.

Database Systems: WASM allows running database engines like SQLite directly in browsers, enabling sophisticated offline-first applications relevant to distributed systems scenarios.

Software Engineering Projects: Large-scale projects can leverage existing C/C++ libraries by compiling them to WASM, following good software engineering principles of code reuse and modularity.

Computer Networks: WASM's binary format is more efficient than JavaScript for network transmission, with sandboxed execution ideal for distributed computing scenarios.

The language-agnostic nature of WebAssembly means your programming skills in C++, Java, or Python can be directly leveraged for high-performance web development, bridging systems programming and modern application development.

Tutorial: Creating Your First Interactive WebAssembly Application with Rust

This tutorial will guide you through creating an interactive pixel art editor using Rust compiled to WebAssembly. We chose Rust because of its excellent WASM toolchain and memory safety features that prevent common web application vulnerabilities.

Prerequisites and Setup

Installing Rust and Required Tools

First, install Rust if you haven't already:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source ~/.cargo/env
```

Install the WebAssembly target for Rust:

```
rustup target add wasm32-unknown-unknown
```

Install `wasm-pack`, the primary tool for building Rust-generated WebAssembly:

```
curl https://rustwasm.github.io/wasm-pack/installer/init.sh -sSf | sh
```

Why These Tools?

- **rustup**: Manages Rust toolchain versions and targets
- **wasm32-unknown-unknown target**: Tells Rust to compile for WebAssembly without assuming any specific operating system
- **wasm-pack**: Handles the entire build pipeline from Rust source to npm-ready package, including generating JavaScript bindings

Project Structure

Create a new Rust library project:

```
cargo new --lib pixel-editor-wasm
cd pixel-editor-wasm
```

Modify `Cargo.toml` to configure WASM compilation:

```toml
[package]
name = "pixel-editor-wasm"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen = "0.2"
js-sys = "0.3"

[dependencies.web-sys]
version = "0.3"
features = [
  "console",
  "Document",
  "Element",
  "HtmlCanvasElement",
  "CanvasRenderingContext2d",
  "ImageData",
  "Window",
]
```

Configuration Explanation:

- `crate-type = ["cdylib"]`: Creates a dynamic library suitable for WASM
- `wasm-bindgen`: Provides bindings between Rust and JavaScript
- `web-sys`: Gives access to Web APIs from Rust
- The features list specifies which Web APIs we need

Core Rust Implementation

Understanding the Pixel Editor Architecture

Before diving into the code, let's understand what we're building and why each component matters:

The Problem We're Solving: Traditional JavaScript struggles with pixel manipulation because it requires intensive mathematical operations on large arrays. Each pixel requires 4 bytes (RGBA channels), so a 400x300 canvas needs 480,000 individual byte operations for a single frame update.

Our WASM Solution: By implementing the pixel logic in Rust and compiling to WebAssembly, we achieve:

- **Memory Safety**: Rust prevents buffer overflows and memory leaks that could crash the browser
- **Performance**: Near-native speed for mathematical operations and array manipulations
- **Type Safety**: Compile-time guarantees that prevent runtime errors

Key Components:

1. **PixelEditor Struct**: Manages the pixel buffer and current drawing state
2. **Memory Layout**: Uses a flat `Vec<u8>` that matches Canvas ImageData format (RGBA interleaved)
3. **Coordinate System**: Converts 2D coordinates (x,y) to 1D array indices using row-major layout
4. **Color Management**: Handles RGBA color values with proper alpha channel support

Why This Architecture Works: The linear memory model of WebAssembly perfectly matches how computer graphics work. GPUs expect pixel data in contiguous memory blocks, and our `Vec<u8>` provides exactly that - enabling zero-copy transfers between WASM and Canvas APIs.

Replace `src/lib.rs` with our pixel editor logic: (Note comments in code which explain further)

```rust
// Essential imports for WebAssembly integration
use wasm_bindgen::prelude::*;
use wasm_bindgen::JsCast;
use web_sys::{CanvasRenderingContext2d, HtmlCanvasElement, ImageData};

// Import the `console.log` function from the Web APIs
// This creates a bridge between WASM and JavaScript's console API
#[wasm_bindgen]
extern "C" {
    #[wasm_bindgen(js_namespace = console)]
    fn log(s: &str);
}

// Define a macro for easier logging with format string support
// This allows us to use printf-style formatting from within WASM
macro_rules! console_log {
    ($($t:tt)*) => (log(&format_args!($($t)*).to_string()))
}

// Main pixel editor structure that will be exposed to JavaScript
// The #[wasm_bindgen] attribute makes this struct accessible from JS
#[wasm_bindgen]
```

```rust
pub struct PixelEditor {
    width: usize,           // Canvas width in pixels
    height: usize,          // Canvas height in pixels
    pixels: Vec<u8>,        // Flat array storing RGBA values (4 bytes per pixel)
    current_color: [u8; 4], // Current brush color as [Red, Green, Blue, Alpha]
}

// Implementation block with methods exposed to JavaScript
#[wasm_bindgen]
impl PixelEditor {
    // Constructor that JavaScript can call with 'new PixelEditor(width, height)'
    #[wasm_bindgen(constructor)]
    pub fn new(width: usize, height: usize) -> PixelEditor {
        console_log!("Creating new PixelEditor: {}x{}", width, height);

        // Calculate total bytes needed: width × height × 4 channels (RGBA)
        let pixel_count = width * height * 4;
        let mut pixels = vec![255u8; pixel_count]; // Initialize all pixels to
white (255)

        // Ensure alpha channel is fully opaque for all pixels
        // We iterate through every 4th byte (alpha channel) and set it to 255
        for i in (3..pixels.len()).step_by(4) {
            pixels[i] = 255; // Alpha = 255 means fully opaque
        }

        PixelEditor {
            width,
            height,
            pixels,
            current_color: [0, 0, 0, 255], // Default to black with full opacity
        }
    }

    // Method to change the current drawing color
    // JavaScript will call this as: editor.set_color(255, 0, 0, 255) for red
    #[wasm_bindgen]
    pub fn set_color(&mut self, r: u8, g: u8, b: u8, a: u8) {
        self.current_color = [r, g, b, a];
        console_log!("Color set to: ({}, {}, {}, {})", r, g, b, a);
    }

    // Core drawing function - paints a single pixel at coordinates (x, y)
    // This demonstrates the critical 2D-to-1D coordinate conversion
    #[wasm_bindgen]
    pub fn paint_pixel(&mut self, x: usize, y: usize) {
        // Bounds checking prevents buffer overflow attacks
        if x < self.width && y < self.height {
            // Convert 2D coordinates to 1D array index
            // Formula: (row × width + column) × 4 channels
            // This is the standard row-major layout used by graphics systems
            let index = (y * self.width + x) * 4;

            // Set RGBA values at the calculated position
            self.pixels[index] = self.current_color[0];     // Red channel
            self.pixels[index + 1] = self.current_color[1]; // Green channel
            self.pixels[index + 2] = self.current_color[2]; // Blue channel
            self.pixels[index + 3] = self.current_color[3]; // Alpha channel
        }
    }

    // Returns a copy of the pixel buffer for JavaScript to render
    // JavaScript will convert this to ImageData for canvas rendering
    #[wasm_bindgen]
    pub fn get_pixels(&self) -> Vec<u8> {
        // We clone the data to maintain ownership boundaries between WASM and JS
        // This ensures memory safety but involves a copy operation
        self.pixels.clone()
    }

    // Utility function to clear the entire canvas to white
    // Demonstrates bulk memory operations in WASM
    #[wasm_bindgen]
    pub fn clear(&mut self) {
        // Iterate through every pixel (every 4 bytes) and set to white
        for i in (0..self.pixels.len()).step_by(4) {
            self.pixels[i] = 255;     // Red = 255 (white)
            self.pixels[i + 1] = 255; // Green = 255 (white)
            self.pixels[i + 2] = 255; // Blue = 255 (white)
            self.pixels[i + 3] = 255; // Alpha = 255 (fully opaque)
        }
        console_log!("Canvas cleared");
    }

    // Advanced function demonstrating algorithmic pixel processing to create
    //   a gradient. Play with this once you are comfortable with the above
    // Creates a diagonal gradient using mathematical calculations
    #[wasm_bindgen]
    pub fn fill_gradient(&mut self) {
        // Nested loops to process every pixel - this is computationally intensive
        for y in 0..self.height {
            for x in 0..self.width {
                // Calculate the 1D index for this 2D coordinate
                let index = (y * self.width + x) * 4;

                // Mathematical gradient calculation:
                // - Distance from top-left corner determines brightness
                // - (x + y) gives us diagonal distance
                // - Divide by total distance to normalize to 0-1 range
                // - Multiply by 255 to get color intensity
                let intensity = ((x + y) as f32 / (self.width + self.height) as
f32 * 255.0) as u8;

                // Apply the calculated intensity to all color channels
                // This creates a grayscale gradient from dark to light
                self.pixels[index] = intensity;     // Red = intensity
                self.pixels[index + 1] = intensity; // Green = intensity
                self.pixels[index + 2] = intensity; // Blue = intensity
                self.pixels[index + 3] = 255;       // Alpha = 255 (fully opaque)
            }
        }
        console_log!("Gradient applied");
    }
}
```

Key Implementation Details Explained:

1. **Memory Management Strategy**: We use `Vec<u8>` to store pixel data because WASM has a linear memory model - all data must be stored in a contiguous block. Each pixel requires exactly 4 bytes (RGBA channels) because this matches the Canvas ImageData format, eliminating conversion overhead when transferring data to JavaScript.

2. **WASM Bindings Deep Dive**: The `#[wasm_bindgen]` attribute generates two crucial components:

   - **Rust → WASM**: Compiles your Rust functions into WASM bytecode
   - **WASM → JS**: Creates JavaScript wrapper functions that handle type conversion between JS and WASM

   Without this attribute, your Rust functions would be compiled to WASM but remain inaccessible from JavaScript.

3. **Cross-Language Debugging**: We import `console.log` because WASM runs in a sandboxed environment with no direct access to browser APIs. This binding allows us to call JavaScript functions from within WASM, essential for debugging and understanding execution flow.

4. **Performance-Optimized Data Access**: The pixel access formula `(y * width + x) * 4` demonstrates row-major memory layout - the same pattern used by graphics hardware and image formats. This mathematical approach is faster than nested arrays because:

   - Single array access vs. double indirection
   - Better CPU cache locality
   - Matches how graphics memory is organized

5. **Type Safety Across Boundaries**: Notice how we use `usize` for coordinates and array indices, but `u8` for color values. This matches WASM's type system where memory addresses are pointer-sized, while color channels are byte-sized. The Rust compiler prevents type mismatches that could cause memory safety issues.

Compilation Process Deep Dive

Build the WebAssembly module:

```
wasm-pack build --target web --out-dir pkg
```

Understanding Each Compilation Stage:

1. **Rust → LLVM IR**: The Rust compiler (rustc) first converts your code to LLVM Intermediate Representation, an architecture-independent format that enables optimizations.
2. **LLVM IR → WASM Bytecode**: LLVM's WASM backend converts the IR to WebAssembly instructions. This stage applies crucial optimizations like:
   - Dead code elimination (removes unused functions)
   - Function inlining (reduces call overhead)
   - Loop optimization (vectorization where possible)
3. **Binding Generation**: `wasm-pack` analyzes your `#[wasm_bindgen]` annotations and generates:
   - **Type definitions**: `.d.ts` files for TypeScript integration
   - **JS glue code**: Functions that marshall data between JS and WASM
   - **Memory management**: Allocation/deallocation helpers for complex types
4. **Package Creation**: The final step creates an npm-compatible package structure, making your WASM module distributable and version-manageable.

Build Flags Explained:

- `--target web`: Generates ES6 modules instead of Node.js modules
- `--out-dir pkg`: Specifies output directory for generated files
- Optional flags you might use:
  - `--debug`: Preserves debug symbols for easier debugging
  - `--no-typescript`: Skips TypeScript definition generation
  - `--scope`: Sets npm package scope for publication

Compilation Performance: This process typically takes 10-30 seconds and produces optimized WebAssembly that's 60-80% smaller than equivalent unoptimized code. The binary format also loads 3-5x faster than parsing equivalent JavaScript.

Web Integration

Understanding the Browser-WASM Integration

Now that we have our compiled WASM module, we need to create a web interface that can load and interact with it. This involves several key concepts:

Module Loading: WASM modules are loaded asynchronously as ES6 modules. The browser downloads the `.wasm` binary separately and instantiates it in a sandboxed environment.

Data Transfer: We'll transfer pixel data between WASM's linear memory and the Canvas API using `ImageData`. This requires converting between Rust's `Vec<u8>` and JavaScript's `Uint8ClampedArray`.

Event Handling: Mouse interactions are captured in JavaScript, converted to pixel coordinates, and passed to WASM functions for processing. The updated pixel data is then rendered back to the canvas.

Performance Strategy: By keeping UI logic in JavaScript and computational logic in WASM, we maintain responsive user interactions while achieving near-native performance for pixel manipulation.

Create an HTML file to use our WASM module: (Note comments in code which explain further)

```html
<!DOCTYPE html>
<html>
<head>
    <title>Pixel Editor - WebAssembly Demo</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 20px;
            background-color: #f0f0f0;
        }

        .container {
            max-width: 800px;
            margin: 0 auto;
            background: white;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 2px 10px rgba(0,0,0,0.1);
        }

        canvas {
            border: 2px solid #333;
            cursor: crosshair;
            image-rendering: pixelated; /* Prevents browser from smoothing our
pixel art */
        }

        .controls {
            margin: 20px 0;
            display: flex;
            gap: 15px;
            align-items: center;
            flex-wrap: wrap;
        }

        .color-picker {
            width: 50px;
            height: 40px;
            border: none;
            border-radius: 4px;
            cursor: pointer;
        }

        button {
            padding: 10px 15px;
            background-color: #007bff;
            color: white;
            border: none;
            border-radius: 4px;
            cursor: pointer;
            font-size: 14px;
        }

        button:hover {
            background-color: #0056b3;
        }

        .info {
            margin-top: 20px;
            padding: 15px;
            background-color: #e9ecef;
            border-radius: 4px;
            font-size: 14px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>WebAssembly Pixel Editor</h1>
        <p>Click and drag on the canvas to draw. This editor is powered by Rust
compiled to WebAssembly!</p>

        <!-- Canvas element where our pixel editor will render -->
        <canvas id="canvas" width="400" height="300"></canvas>

        <div class="controls">
            <!-- Color picker for changing brush color -->
            <label>Color: <input type="color" id="colorPicker" class="color-
picker" value="#000000"></label>
            <!-- Buttons to trigger WASM functions -->
            <button id="clearBtn">Clear Canvas</button>
            <button id="gradientBtn">Apply Gradient</button>
        </div>
        <div class="info">
            <strong>Performance Info:</strong> This application demonstrates
WebAssembly's performance advantages for algorithmic operations.
            The pixel manipulation logic runs in compiled WebAssembly, showcasing
how compute-intensive tasks benefit from near-native performance.
            Open your browser's developer console to see debug messages from the
WASM module.
        </div>
    </div>

    <!-- ES6 module script - required for WASM loading -->
    <script type="module">
        // Import the WASM initialization function and our PixelEditor class
        // The 'init' function downloads and instantiates the .wasm binary
        import init, { PixelEditor } from './pkg/pixel_editor_wasm.js';

        async function run() {
            // Step 1: Initialize the WASM module
            // This downloads the .wasm file and sets up the WebAssembly instance
            await init();

            // Step 2: Get DOM elements we'll interact with
            const canvas = document.getElementById('canvas');
            const ctx = canvas.getContext('2d'); // 2D rendering context for pixel
manipulation
            const colorPicker = document.getElementById('colorPicker');
            const clearBtn = document.getElementById('clearBtn');
            const gradientBtn = document.getElementById('gradientBtn');

            // Step 3: Create our WASM pixel editor instance
            // This calls the Rust constructor we defined with #
[wasm_bindgen(constructor)]
            const editor = new PixelEditor(canvas.width, canvas.height);

            // Utility function: Convert hex color (from color picker) to RGB
values
            // WASM expects separate R, G, B, A values, not hex strings
            function hexToRgb(hex) {
                const result = /^#?([a-f\d]{2})([a-f\d]{2})([a-f\d]
{2})$/i.exec(hex);
                return result ? {
                    r: parseInt(result[1], 16), // Parse red component
                    g: parseInt(result[2], 16), // Parse green component
                    b: parseInt(result[3], 16)  // Parse blue component
                } : null;
            }

            // Core rendering function: Transfer pixel data from WASM to Canvas
            function render() {
                // Get pixel data from WASM (this returns a copy of the Vec<u8>)
                const pixelData = editor.get_pixels();

                // Create ImageData object that Canvas can understand
                // Uint8ClampedArray ensures values stay in 0-255 range
                const imageData = new ImageData(
                    new Uint8ClampedArray(pixelData), // Convert to clamped array
                    canvas.width,
                    canvas.height
                );

                // Draw the pixel data to the canvas at position (0,0)
                ctx.putImageData(imageData, 0, 0);
            }

            // Event Handler: Color picker changes
            colorPicker.addEventListener('change', (e) => {
                const rgb = hexToRgb(e.target.value);
                if (rgb) {
                    // Call WASM function to update the current color
                    // Alpha is set to 255 (fully opaque)
                    editor.set_color(rgb.r, rgb.g, rgb.b, 255);
                }
            });

            // Mouse drawing state management
            let isDrawing = false;

            // Event Handler: Mouse press starts drawing
            canvas.addEventListener('mousedown', (e) => {
                isDrawing = true;
                draw(e); // Draw initial pixel
            });

            // Event Handler: Mouse move continues drawing if pressed
            canvas.addEventListener('mousemove', (e) => {
                if (isDrawing) {
                    draw(e);
                }
            });

            // Event Handler: Mouse release stops drawing
            canvas.addEventListener('mouseup', () => {
                isDrawing = false;
            });

            // Event Handler: Mouse leaving canvas stops drawing
            canvas.addEventListener('mouseout', () => {
                isDrawing = false;
            });

            // Core drawing function: Convert mouse coordinates to pixel
coordinates
            function draw(e) {
                // Get canvas position relative to viewport
                const rect = canvas.getBoundingClientRect();

                // Calculate pixel coordinates from mouse position
                // Math.floor ensures we get integer pixel coordinates
                const x = Math.floor(e.clientX - rect.left);
                const y = Math.floor(e.clientY - rect.top);

                // Call WASM function to paint the pixel
                editor.paint_pixel(x, y);

                // Update the canvas display with new pixel data
                render();
            }

            // Event Handler: Clear button calls WASM clear function
            clearBtn.addEventListener('click', () => {
                editor.clear(); // Call Rust function to clear pixel buffer
                render();       // Update canvas display
            });

            // Event Handler: Gradient button calls WASM gradient function
            gradientBtn.addEventListener('click', () => {
                editor.fill_gradient(); // Call Rust function to generate gradient
                render();               // Update canvas display
            });

            // Initial render to show the white canvas
            render();

            console.log('Pixel Editor loaded successfully!');
        }

        // Start the application and catch any initialization errors
        run().catch(console.error);
    </script>
</body>
</html>
```

Key Integration Points Explained:

1. **Asynchronous Loading**: The `await init()` call is crucial - it downloads and instantiates the WASM module before we can use any WASM functions.

2. **Memory Bridge**: The `render()` function demonstrates how data flows from WASM linear memory to Canvas ImageData, enabling zero-copy pixel manipulation.

3. **Coordinate Translation**: The `draw()` function shows how we convert mouse coordinates (relative to viewport) to pixel coordinates (relative to canvas), handling browser scaling and positioning.

4. **Event-Driven Architecture**: Mouse events trigger JavaScript functions that call WASM methods, then immediately update the display - this hybrid approach keeps the UI responsive while leveraging WASM performance.

5. **Error Handling**: The `run().catch()` pattern ensures any WASM loading errors are properly reported, which is essential for debugging module initialization issues.

Debugging and Development Workflow

Understanding how to debug WASM applications is crucial for development:

Browser Developer Tools Integration

1. **Chrome DevTools**: Enable WASM debugging by opening DevTools → Settings → Experiments → "WebAssembly Debugging"
2. **Source Maps**: When built with `--debug`, you can set breakpoints directly in your Rust source code
3. **Memory Inspector**: Use the Memory tab to examine WASM linear memory and identify memory leaks

Common Development Issues

Performance Debugging:

```rust
use web_sys::Performance;

#[wasm_bindgen]
pub fn performance_test(&mut self) {
    let window = web_sys::window().unwrap();
    let performance = window.performance().unwrap();
    let start = performance.now();

    // Your performance-critical code here
    self.fill_gradient();

    let end = performance.now();
    console_log!("Operation took {} milliseconds", end - start);
}
```

Memory Management Verification:

- Use `wee_alloc` for smaller WASM binary size
- Monitor memory growth in DevTools Memory tab
- Implement custom allocators for specific use cases

Build Optimization Strategies

Release Builds:

```
wasm-pack build --target web --release
```

- Enables LLVM optimizations (-O3 equivalent)
- Reduces binary size by 40-60%
- Removes debug information and assertions

Size Optimization:

```
[profile.release]
opt-level = "s"  # Optimize for size
lto = true       # Link-time optimization
```

These optimizations are crucial for production deployment where bundle size affects load times.

1. Serve the files using a local web server (required for WASM security):

   ```
   # Python 3
   python3 -m http.server 8000

   # Node.js (if you have http-server installed)
   npx http-server
   ```

2. Navigate to `http://localhost:8000` in your browser

3. Test the functionality:

   - Change colors using the color picker
   - Draw on the canvas by clicking and dragging
   - Clear the canvas
   - Apply the gradient effect
   - Open developer console to see WASM debug messages

Understanding the Performance Benefits

This application demonstrates several key advantages of WebAssembly:

1. **Computational Efficiency**: Pixel manipulation operations run at near-native speed in WASM compared to JavaScript.

2. **Memory Management**: Rust's ownership system prevents memory leaks that commonly occur in JavaScript applications.

3. **Type Safety**: The Rust compiler catches errors at compile time that would cause runtime failures in JavaScript.

4. **Code Reuse**: The same Rust logic could be compiled for desktop, mobile, or server applications with minimal changes.

System Interaction Analysis

Understanding how the different components interact is crucial:

1. **Browser → JavaScript**: User interactions (mouse clicks, color changes) are captured by JavaScript event handlers.

2. **JavaScript → WASM**: JavaScript calls WASM functions through generated bindings. Data is serialized/deserialized at this boundary.

3. **WASM → Memory**: The WASM module manipulates a linear memory space that stores pixel data.

4. **WASM → JavaScript**: Pixel data is returned to JavaScript as a typed array for rendering.

5. **JavaScript → Canvas**: The HTML5 Canvas API renders the pixel data to the screen.

This architecture allows computationally intensive tasks to run in WASM while keeping the UI responsive through JavaScript.

Extending the Application

To further develop your understanding, consider these enhancements:

- **Brush Sizes**: Modify the `paint_pixel` function to paint larger brushes
- **Undo/Redo**: Implement a command pattern to track and reverse operations
- **Image Export**: Add functionality to export the canvas as PNG/JPEG
- **Performance Monitoring**: Add timing measurements to compare WASM vs JavaScript performance
- **Advanced Filters**: Implement blur, sharpen, or other image processing effects

Conclusion

This tutorial demonstrated the complete workflow of creating a WebAssembly application using Rust. You've learned how to set up the toolchain, write performance-critical code in Rust, compile it to WASM, and integrate it with web technologies. The pixel editor showcases WASM's strengths in computational tasks.

WebAssembly represents a significant evolution in software development, enabling developers to bring the performance and reliability of systems programming languages to web applications. As computer science professionals, understanding WASM is crucial for developing high-performance applications that can compete with native software while leveraging the ubiquity and accessibility of web platforms.

References

1. WebAssembly Community Group. (2025). WebAssembly 2.0 (Draft 2025-06-24). https://webassembly.github.io/spec/core/intro/introduction.html
2. Gallant, G. (2025, January 29). The State of WebAssembly – 2024 and 2025. Platform Uno. https://platform.uno/blog/state-of-webassembly-2024-2025/
3. Atak Interactive. (2025, March 10). WebAssembly in 2025: The Future of High-Performance Web Applications. https://www.atakinteractive.com/blog/webassembly-in-2025-the-future-of-high-performance-web-applications
4. Code Solutions Hub. (2024, October 18). The Rise of WASM (WebAssembly) in 2024: Why Every Developer Should Care. DEV Community. https://dev.to/codesolutionshub/the-rise-of-wasm-webassembly-in-2024-why-every-developer-should-care-6i0
5. Garg, A. (2025, June 9). Role of WebAssembly (WASM) in Web Apps in 2025. https://amitgarg.ca/webassembly-wasm-high-performance-web-apps-2025/
6. Mozilla Developer Network. Compiling from Rust to WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Rust_to_Wasm