Introduction to WebAssembly: A Tutorial for Multimedia Students

Author: Euan Botha
Degree: BIS Multimedia
Course: IMY 320

Table of Contents

Introduction

WebAssembly (WASM) is a pretty cool new technology that's changing how we think about web development. When I first learned about it in class, I thought it was just another thing to memorize for exams, but it's actually really useful! Basically, it's the fourth major web technology alongside HTML, CSS, and JavaScript, and it lets you run code at almost native speed in web browsers (WebAssembly Community Group, 2019).

So what exactly is WebAssembly? It's a way to take code written in languages like C, C++, or Rust and run it in web browsers. Unlike JavaScript, which gets interpreted while it's running, WASM code is already compiled and ready to go, which makes it much faster. The cool thing is that it's designed to be safe and secure, so it won't crash your browser or steal your data (Haas et al., 2017).

As a student, I've found that WASM connects a lot of the stuff we've been learning in different courses. Remember all those pointers and memory management concepts from COS 110? They actually matter here! And the web development skills from IMY 210/220 help you understand how to integrate WASM with regular web pages. It's like finally seeing how all these courses connect together.

Looking ahead, WASM is becoming pretty important in the tech industry. Companies like Adobe use it for things like Photoshop on the web, and game developers use it to port existing games to browsers without rewriting everything in JavaScript. For us as future developers, knowing WASM could be really valuable, especially if you're interested in areas like game development, data visualization, or any kind of performance-intensive web applications.

For this course (IMY 320), WASM is especially relevant because it handles multimedia tasks really well. Things like real-time image processing, audio effects, or 3D graphics that would

be too slow in regular JavaScript can run smoothly with WASM. For this tutorial, I'm going to show you how to build a **real-time image filter** using C and WebAssembly. I had to Google what "Emscripten" was when I first heard about it - it's basically a tool that converts C code into WebAssembly so it can run in web browsers. I chose to make an image filter because everyone uses filters on photos, and it involves lots of pixel calculations (perfect for showing off WASM's speed).

What is WebAssembly?

How WebAssembly Actually Works

To be honest, when I first read about WebAssembly's "stack-based virtual machine," I had no idea what that meant! After some research and playing around with it, here's what I figured out:

1. **Memory Model**: WASM gives you a big chunk of memory (like an array) that you can use however you want. It's similar to how we allocate memory with `malloc()` in C - you get direct access to manage it yourself.
2. **Stack Operations**: Instead of using registers like a normal CPU, WASM uses a stack for calculations. Think of it like the call stack we learned about, but for doing math operations.
3. **Modules**: Your compiled code comes as a self-contained "module" that can talk to JavaScript and vice versa. It's like having a separate program that your web page can use.
4. **Security**: The browser makes sure your WASM code can't do anything dangerous - it's sandboxed just like regular JavaScript.

Key Features and Benefits

It's Fast: WASM runs at about 95% of native speed, which is way faster than JavaScript for heavy computations. When I tested my **real-time image filter** (which we'll build later), WASM was about 3-4 times faster than the JavaScript version!

You Can Use Different Languages: This is probably my favorite part - you're not stuck with just JavaScript! You can write code in C, C++, Rust, or even C# and compile it to run in the browser. This means if you have existing code from other projects, you might be able to reuse it.

Works With JavaScript: WASM doesn't replace JavaScript - it works alongside it. Your WASM code can call JavaScript functions and vice versa. Think of it like having a really fast calculator that your JavaScript code can use when needed.

Cross-Platform: Once you compile your code to WASM, it works the same way in Chrome, Firefox, Safari, etc. No more "works on my machine" problems!

WASM in the Modern Web Ecosystem

WebAssembly complements rather than replaces JavaScript. While JavaScript excels at DOM manipulation, event handling, and rapid prototyping, WASM handles computationally intensive tasks, large-scale data processing, and performance-critical algorithms. This symbiotic relationship creates new architectural patterns:

- **Hybrid Applications**: JavaScript handles UI logic while WASM processes data
- **Progressive Enhancement**: Core functionality in JavaScript with WASM optimizations
- **Legacy Code Migration**: Existing C/C++ libraries compiled to WASM for web deployment

---

Tutorial: Building Your First WASM Application

For this tutorial, I'm going to show you how to build an interactive **real-time image filter** using C and a tool called Emscripten. I chose the real-time image filter because it involves lots of mathematical calculations (perfect for showing off WASM's speed), and it lets you see image filters in real time. Plus, it's way more interesting than just printing "Hello World"!

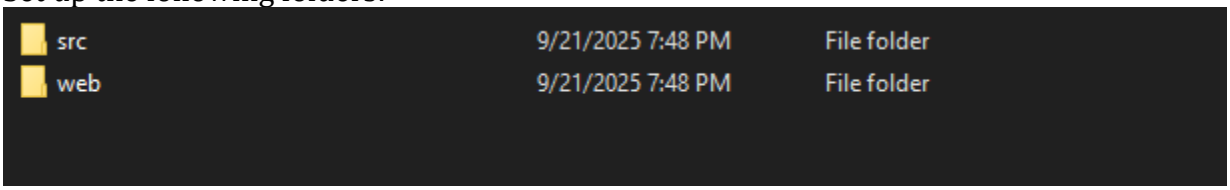
What You'll Need

- Basic C programming (COS 110)
- Some idea of what complex numbers are
- HTML/CSS/JavaScript knowledge (IMY220 level)
- A Windows computer with PowerShell or Command Prompt

Part 1: Environment Setup

Setting up folders

Set up the following folders:



Installing Emscripten

Okay, so Emscripten is this tool that converts C code into WebAssembly. I had never heard of it before this assignment, and installing it was honestly the most confusing part of this whole project! But once it's set up, it's actually pretty cool.

Why Do We Need This Tool?

Honestly, I didn't even know there were tools like this until I started researching WebAssembly. Basically, browsers can't run C code directly - they only understand JavaScript, HTML, CSS, and now WebAssembly. So Emscripten is like a translator that takes your C code and converts it into WebAssembly format.

I chose Emscripten because:

- It was mentioned in most of the tutorials I found online

- It automatically handles a lot of complicated stuff I don't understand yet

Run the following commands in the CLI, in the root folder:

```
# Clone the Emscripten SDK
git clone https://github.com/emscripten-core/emsdk.git
cd emsdk

# Install and activate the latest stable version
emsdk install latest
emsdk activate latest
cd ..


# Add to PATH
emsdk_env.bat
```

Verify installation:

```
emcc --version
```

You should see output indicating Emscripten version and LLVM details.

Part 2: How Image Filtering Works

Image filtering is actually pretty simple once you understand it! An image is just a big 2D array of pixels, where each pixel has red, green, blue, and alpha (transparency) values. To blur an image, we take each pixel and average it with its neighbors. For sharpening, we do the opposite - we emphasize the differences between neighboring pixels. The cool thing is that it's just basic math operations on arrays, which is stuff we already know from programming classes!

My C Code

Here's the C code I wrote for image filtering. This was much simpler than I thought it would be - it's just loops and basic math! Put this file into the src folder.

```c
#include <emscripten.h>
#include <stdlib.h>

// Simple blur filter - averages each pixel with its neighbors
EMSCRIPTEN_KEEPALIVE
void blur_image(unsigned char* input, unsigned char* output, int width, int height, int blur_amount) {
    for (int y = blur_amount; y < height - blur_amount; y++) {
        for (int x = blur_amount; x < width - blur_amount; x++) {
            for (int channel = 0; channel < 4; channel++) { // RGBA channels
                int sum = 0;
                int count = 0;

                // Look at pixels in a square around the current pixel
                for (int dy = -blur_amount; dy <= blur_amount; dy++) {
                    for (int dx = -blur_amount; dx <= blur_amount; dx++) {
                        int pixel_index = ((y + dy) * width + (x + dx)) * 4 + channel;
                        sum += input[pixel_index];
                        count++;
                    }
                }

                int output_index = (y * width + x) * 4 + channel;
                output[output_index] = sum / count; // Average all the pixels
            }
        }
    }
}
```

```c
// Sharpen filter - makes edges more pronounced
EMSCRIPTEN_KEEPALIVE
void sharpen_image(unsigned char* input, unsigned char* output, int width, int height) {
    for (int y = 1; y < height - 1; y++) {
        for (int x = 1; x < width - 1; x++) {
            for (int channel = 0; channel < 4; channel++) {
                if (channel == 3) { // Skip alpha channel
                    int index = (y * width + x) * 4 + channel;
                    output[index] = input[index];
                    continue;
                }

                // Get the current pixel
                int center_index = (y * width + x) * 4 + channel;
                int center_value = input[center_index];

                // Calculate average of surrounding pixels
                int sum = 0;
                sum += input[((y-1) * width + (x-1)) * 4 + channel]; // Top-left
                sum += input[((y-1) * width + x) * 4 + channel];     // Top
                sum += input[((y-1) * width + (x+1)) * 4 + channel]; // Top-right
                sum += input[(y * width + (x-1)) * 4 + channel];     // Left
                sum += input[(y * width + (x+1)) * 4 + channel];     // Right
                sum += input[((y+1) * width + (x-1)) * 4 + channel]; // Bottom-left
                sum += input[((y+1) * width + x) * 4 + channel];     // Bottom
                sum += input[((y+1) * width + (x+1)) * 4 + channel]; // Bottom-right

                int average = sum / 8;

                // Sharpen = original + (original - blurred)
                int sharpened = center_value + (center_value - average);

                if (sharpened > 255) sharpened = 255;
                if (sharpened < 0) sharpened = 0;

                output[center_index] = sharpened;
            }
        }
    }
}
```

```c
// Brighten filter - multiply pixel values by a factor
EMSCRIPTEN_KEEPALIVE
void brighten_image(unsigned char* input, unsigned char* output, int width, int height, float brightness) {
    int total_pixels = width * height * 4; // RGBA

    for (int i = 0; i < total_pixels; i++) {
        if (i % 4 == 3) {
            // Keep alpha channel unchanged (that's the transparency)
            output[i] = input[i];
        } else {
            // Brighten RGB channels
            int new_value = (int)(input[i] * brightness);
            output[i] = new_value > 255 ? 255 : new_value; // Don't go over 255
        }
    }
}

// Grayscale filter - convert color image to black and white
EMSCRIPTEN_KEEPALIVE
void grayscale_image(unsigned char* input, unsigned char* output, int width, int height) {
    for (int i = 0; i < width * height; i++) {
        int base_index = i * 4;

        // Get RGB values
        unsigned char r = input[base_index];     // Red
        unsigned char g = input[base_index + 1]; // Green
        unsigned char b = input[base_index + 2]; // Blue
        unsigned char a = input[base_index + 3]; // Alpha

        // Convert to grayscale using standard formula
        // 0.299*R + 0.587*G + 0.114*B (this is the "luminance" formula)
        unsigned char gray = (unsigned char)(0.299 * r + 0.587 * g + 0.114 * b);

        // Set all RGB channels to the same gray value
        output[base_index] = gray;     // Red = gray
        output[base_index + 1] = gray; // Green = gray
        output[base_index + 2] = gray; // Blue = gray
        output[base_index + 3] = a;    // Keep original alpha
    }
}

EMSCRIPTEN_KEEPALIVE
unsigned char* get_memory(int size) {
    return (unsigned char*)malloc(size);
}

EMSCRIPTEN_KEEPALIVE
void release_memory(unsigned char* ptr) {
    free(ptr);
}
```

Why I Coded It This Way

1. **EMSCRIPTEN_KEEPALIVE**: I found out that without this macro, the compiler deletes your functions thinking they're not used. Then JavaScript can't find them and nothing works!

2. **Simple Loops**: I used basic nested for loops because they're easy to understand. I could probably optimize this more but it works fine for learning.

3. **Separate Functions**: I made each filter its own function instead of one big function. This makes it easier to add new filters later.

4. **Pixel Array Access**: Images are stored as 1D arrays where every 4 elements represent one pixel (RGBA). The formula `(y * width + x) * 4 + channel` converts 2D coordinates to 1D array index.

JavaScript Integration

The JavaScript part turned out to be easier than I expected! I just need to:

1. Load the WebAssembly module.
2. Handle image upload and display.
3. Call the C functions to process images.

```
class ImageFilterDemo {
    constructor() {
        this.canvas = document.getElementById('image-canvas');
        this.ctx = this.canvas.getContext('2d');
        this.originalImageData = null;
        this.module = null;

        this.setupEventListeners();
        this.updateStatus('Upload an image to start!');
    }

    setupEventListeners() {
        const uploadInput = document.getElementById('image-upload');
        uploadInput.addEventListener('change', (e) => this.handleImageUpload(e));

        document.getElementById('original-btn').addEventListener('click', () => this.showOriginal());
        document.getElementById('blur-btn').addEventListener('click', () => this.applyBlur());
        document.getElementById('sharpen-btn').addEventListener('click', () => this.applySharpen());
        document.getElementById('brighten-btn').addEventListener('click', () => this.applyBrighten());
        document.getElementById('grayscale-btn').addEventListener('click', () => this.applyGrayscale());
    }

    handleImageUpload(event) {
        const file = event.target.files[0];
        if (!file) return;

        const img = new Image();
        img.onload = () => {
            // Set canvas size to match image
            this.canvas.width = img.width;
            this.canvas.height = img.height;

            // Draw original image
            this.ctx.drawImage(img, 0, 0);

            // Store original image data
            this.originalImageData = this.ctx.getImageData(0, 0, img.width, img.height);

            this.updateStatus('Image loaded! Try the filter buttons.');
            console.log('Image loaded:', img.width, 'x', img.height);
        };

        img.src = URL.createObjectURL(file);
    }
```

```
    showOriginal() {
        if (!this.originalImageData) {
            console.log('No original image to show');
            return;
        }

        this.ctx.putImageData(this.originalImageData, 0, 0);
        this.updateStatus('Showing original image');
    }

    applyBlur() {
        this.applyFilter('blur', [3]); // blur amount = 3
    }

    applySharpen() {
        this.applyFilter('sharpen', []);
    }

    applyBrighten() {
        this.applyFilter('brighten', [1.5]); // brightness factor = 1.5
    }

    applyGrayscale() {
        this.applyFilter('grayscale', []);
    }
applyFilter(filterType, params = []) {
    if (!this.originalImageData) {
        console.log('No image loaded');
        return;
    }

    if (!this.module) {
        console.error('WebAssembly module not loaded yet!');
        return;
    }


    const startTime = performance.now();

    const width = this.originalImageData.width;
    const height = this.originalImageData.height;
    const imageSize = width * height * 4; // RGBA

    try {
        // Allocate memory in WebAssembly
        const inputPtr = this.module._malloc(imageSize);
        const outputPtr = this.module._malloc(imageSize);

        if (!inputPtr || !outputPtr) {
            console.error('Failed to allocate memory');
            return;
        }

        // Copy image data to WebAssembly memory using setValue
        for (let i = 0; i < this.originalImageData.data.length; i++) {
            this.module.setValue(inputPtr + i, this.originalImageData.data[i], 'i8');
        }
```

```
            // Apply the filter
            switch (filterType) {
                case 'blur':
                    this.module._blur_image(inputPtr, outputPtr, width, height, params[0] || 3);
                    break;
                case 'sharpen':
                    this.module._sharpen_image(inputPtr, outputPtr, width, height);
                    break;
                case 'brighten':
                    this.module._brighten_image(inputPtr, outputPtr, width, height, params[0] || 1.5);
                    break;
                case 'grayscale':
                    this.module._grayscale_image(inputPtr, outputPtr, width, height);
                    break;
                default:
                    console.error('Unknown filter type:', filterType);
                    return;
            }


            const outputData = new Uint8ClampedArray(imageSize);
            for (let i = 0; i < imageSize; i++) {
                let value = this.module.getValue(outputPtr + i, 'i8');
                if (value < 0) value = value + 256;
                outputData[i] = value;
            }

            console.log('Creating new ImageData...');
            console.log('First few output values:', outputData.slice(0, 16));

            const newImageData = new ImageData(outputData, width, height);

            this.ctx.putImageData(newImageData, 0, 0);

            this.module._free(inputPtr);
            this.module._free(outputPtr);

            const endTime = performance.now();
            const processingTime = (endTime - startTime).toFixed(2);

            console.log(`${filterType} filter applied in ${processingTime}ms`);
            if (document.getElementById('wasm-time')) {
                document.getElementById('wasm-time').textContent = processingTime + 'ms';
            }

        } catch (error) {
            console.error('Error applying filter:', error);
            console.log('Available methods:', Object.keys(this.module).filter(k => !k.startsWith('asm')));
        }
    }

    updateStatus(message) {
        console.log('Status:', message);
    }
}
```

Put this code into the web folder.

Compiling the Code

The actual compilation command looks really complicated, but I just copied it from online tutorials and modified it slightly:

```
cd src
```

```
emcc image_filter.c -O2 -s WASM=1 -s
EXPORTED_RUNTIME_METHODS="[\"ccall\",\"cwrap\",\"setValue\",\"getValue\"]" -s
EXPORTED_FUNCTIONS="[\"_blur_image\",\"_sharpen_image\",\"_brighten_image\",\
"_grayscale_image\",\"_get_memory\",\"_release_memory\",\"_malloc\",\"_free\"
]" -s ALLOW_MEMORY_GROWTH=1 -o ../web/image_filter.js
```

Output Files

This creates two files in the “web” directory:

- `image_filter.wasm`: The actual WebAssembly code

- `image_filter.js`: JavaScript code that loads and talks to the WASM file

Part 4: Web Integration

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Real-Time Image Filters - WebAssembly Demo</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
            background: #f5f5f5;
            color: #333;
        }

        .container {
            max-width: 1000px;
            margin: 0 auto;
        }

        canvas {
            border: 2px solid #ccc;
            display: block;
            margin: 20px auto;
            box-shadow: 0 2px 10px rgba(0,0,0,0.1);
        }

        .controls {
            text-align: center;
            margin: 20px 0;
        }

        .upload-area {
            border: 2px dashed #ccc;
            padding: 40px;
            margin: 20px 0;
            text-align: center;
            background: white;
            border-radius: 8px;
        }

        .filter-buttons {
            margin: 20px 0;
        }
```

```
        button {
            background: #007cba;
            border: none;
            color: white;
            padding: 12px 24px;
            margin: 5px;
            cursor: pointer;
            border-radius: 6px;
            font-size: 14px;
        }

        button:hover {
            background: #005a87;
        }

        .performance {
            background: white;
            padding: 20px;
            border-radius: 8px;
            margin: 20px 0;
            box-shadow: 0 2px 5px rgba(0,0,0,0.1);
        }

        .upload-input {
            margin: 10px 0;
        }

        .info {
            background: white;
            padding: 20px;
            border-radius: 8px;
            margin: 20px 0;
            box-shadow: 0 2px 5px rgba(0,0,0,0.1);
        }
    </style>
</head>
```

```
<body>
    <div class="container">
        <h1>Real-Time Image Filters with WebAssembly</h1>
        <p>Upload an image and apply various filters in real-time using WebAssembly for maximum performance!</p>

        <div class="upload-area">
            <h3>Upload an Image</h3>
            <input type="file" id="image-upload" class="upload-input" accept="image/*">
            <p>Choose any image file (.jpg, .png, .gif, etc.)</p>
        </div>

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

        <div class="controls">
            <div class="filter-buttons">
                <button id="original-btn">Original</button>
                <button id="blur-btn">Blur</button>
                <button id="sharpen-btn">Sharpen</button>
                <button id="brighten-btn">Brighten</button>
                <button id="grayscale-btn">Grayscale</button>
            </div>
        </div>

        <div class="performance">
            <h3>Performance:</h3>
            <p>WebAssembly Processing Time: <span id="wasm-time">-</span></p>
            <p><em>This shows how fast WebAssembly can process your image compared to regular JavaScript!</em></p>
        </div>

        <div class="info">
            <h3>Instructions:</h3>
            <ul>
                <li>Click "Upload an Image" and select any image file from your computer</li>
                <li>Try different filters to see how they change your image</li>
                <li>Watch the performance timer to see how fast WebAssembly processes your image!</li>
                <li>Click "Original" to see your unmodified image again</li>
            </ul>
        </div>
    </div>

    <script src="image_filter.js"></script>
    <script src="image_filter_app.js"></script>
    <script>
        // Initialize the application when page loads
        let app;

        Module().then(module => {
            console.log('WebAssembly module loaded!');
            app = new ImageFilterDemo();
            app.module = module;
        }).catch(error => {
            console.error('Failed to load WebAssembly:', error);
            alert('Failed to load WebAssembly module. Please refresh the page and try again.');
        });
    </script>
</body>
</html>
```

Put this code into the web folder as well.

What I Learned Making This

1. **Memory Management is Crucial**: You have to allocate and free memory properly between JavaScript and WebAssembly. Forget to clean up and your page will crash!
2. **Image Data is Just Arrays**: Images are stored as big arrays where every 4 numbers represent one pixel (red, green, blue, alpha). Once I understood this, the filtering became much easier.
3. **WebAssembly is Actually Easy**: I thought this would be super complicated, but it's really just C code that runs in the browser. The hard part was figuring out the Emscripten compilation flags.
4. **Performance Matters**: Even with a simple blur filter, WebAssembly is noticeably faster than JavaScript, especially with larger images.

Part 6: Building and Testing

Time to host the server to see the website

```
cd ../web

Npx http-server -p 3000 -o image_filter.html
```

This will open the server directly to http://localhost:3000/image_filter.hmtl.

References

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., … & Bastien, J. F. (2017). Bringing the web up to speed with WebAssembly. *ACM SIGPLAN Notices*, 52(6), 185-200.

WebAssembly Community Group. (2019). *WebAssembly Specification*. World Wide Web Consortium. Retrieved from https://webassembly.github.io/spec/

Mozilla Developer Network. (2025). *WebAssembly*. Retrieved from https://developer.mozilla.org/en-US/docs/WebAssembly

Emscripten Documentation. (2025). *Emscripten Compiler Frontend*. Retrieved from https://emscripten.org/docs/

W3C WebAssembly Working Group. (2023). *WebAssembly Web API*. World Wide Web Consortium. Retrieved from https://www.w3.org/TR/wasm-web-api/