IMY 320 Written Research

Intro to WebAssembly
For 2nd Year BIS Multimedia Students

Introduction

Overview of WebAssembly

WebAssembly is a low-level byte format that allows you to use fast, lightweight and efficient programming languages like C, C++ and Rust in a web browser. JavaScript acts as a bridge between these languages and the web browser. This allows you to implement complex systems in a browser environment while utilising the power of these languages instead of solely relying on JavaScript. WebAssembly allows you to make portable web applications across all major browsers while achieving near-native performance.

Importance for your future

WebAssembly allows you to use apps that were once only accessible on a desktop in a browser while achieving near-native speeds and functionality. This is important as apps are so much more accessible now, and learning how to utilise WebAssembly will be vital to your career in the software development industry. It allows you to optimise a wide range of apps or software and run them in a browser. Examples of real-world applications using WebAssembly include: Figma, AutoCAD Web and AI that allows for image filters.

Application to Multimedia

During your degree in BIS Multimedia, you will work on countless software applications, from React web applications to low-level programming concepts. This experience will give you the technical knowledge needed to design and implement complex WebAssembly applications. Projects like your third-year game could be made accessible in a web browser using WebAssembly, allowing users to test/play your game in a browser regardless of the hardware requirements. Concurrency algorithms from COS 226, along with algorithms and

data structures from COS 212 and systems from COS 110 and COS 214 can now achieve near-native performance in a browser. All of this can be displayed in a browser and can be styled using all of the knowledge you learned in IMY 110. You can easily combine all aspects of your Multimedia degree into one system, from your UI/UX experience gained in IMY to your deep technical knowledge gained from all your COS modules.

Tutorial

This tutorial will guide you through the entire setup and development of a WebAssembly (WASM) application in C++, this will result in a simple interactive drawing board.

This tutorial covers the installation and setup for Windows only.

Step 1: Install Python

Python will be needed to run a simple HTTP server to serve the WASM files. Browsers block direct file:// access for WebAssembly due to security, so a local server is needed.

1. Go to the Python site
    a. https://www.python.org/downloads/windows/
2. Download the latest version (currently Python 3.13.7)
3. After downloading, run the installation wizard
    a. Check “Add Python to Path”
    b. Click Install Now
4. Verify the installation in command prompt
    a. python --version
    b. You should see the version you installed
    c. If you get an error or nothing happens, Python was not added to your computer’s PATH.
        i. What is PATH
            1. The PATH is just a list of folders that your computer checks when you type a command. If Python’s folder is not on that list, your computer won’t know where to find it.
            2. To fix this, make sure you checked “Add Python to PATH” during installation.

Step 2: Install Emscripten

Emscripten is a compiler toolchain that converts C and C++ code into WebAssembly (.wasm) along with the JavaScript code needed for all browsers. Without it, the C++ code cannot run in a web browser.

1. Make a folder to store Emscripten (e.g., C:\emsdk) and navigate to it in Command Prompt

```
cd C:\emsdk
```

2. Clone this Emscripten SDK GitHub repo onto your machine

```
git clone https://github.com/emscripten-core/emsdk.git
```

3. Navigate into that folder

```
cd emsdk
```

4. Fetch the latest changes

```
git pull
```

5. Install the latest SDK tools

```
.\emsdk install latest
```

6. Activate the last SDSK for your user

```
.\emsdk activate latest
```

7. Set up the environment for the current terminal session

```
.\emsdk_env.ps1
```

    a. Or in Command prompt

```
emsdk_env.bat
```

    b. Explanation

i. emsdk install latest downloads the latest compiler.
ii. emsdk activate latest sets it as the active version.
iii. emsdk_env.ps1 or emsdk_env.bat updates environment variables so emcc works in your terminal.

c. Verify Installation

```
emcc -v
```

i. If installed correctly, you will see the Emscripten version

d. If not installed correctly, here is the documentation:
i. https://emscripten.org/docs/getting_started/downloads.html

Step 3: Getting code

After setting up everything you need to run WebAssembly, we need to actually cover the code.

1. Make a folder where you want to store the code.

```
mkdir C:\WASM-Drawing

cd C:\WASM-Drawing
```

2. Clone this GitHub repo with the code onto your machine

```
git clone https://github.com/Ruan-le-Roux/WebAssembly
```

3. The file directory should look like this:

```
project-folder/
├── src/
│   └── drawing.cpp
├── README.md
├── drawing.js      (auto-generated)
├── drawing.wasm    (auto-generated)
└── index.html
```

Step 4: Compiling the program

This will cover the steps needed to compile and run the program.

1. Command

```
emcc src\drawing.cpp -o drawing.js -s WASM=1 -s
EXPORTED_FUNCTIONS=["_init","_draw_circle","_get_pixels","_clear_pixels"] -s
EXPORTED_RUNTIME_METHODS=["cwrap","HEAPU8"]
```

2. Explanation
    a. emcc src\drawing.cpp → Tells Emscripten to compile the C++ file.
    b. -o drawing.js → Creates two output files: drawing.js and drawing.wasm.
    c. -s WASM=1 → Ensures the program is compiled to WebAssembly (not just JavaScript).
    d. -s EXPORTED_FUNCTIONS=[...] → Lists the C++ functions we want to call from JavaScript (_init, _draw_circle, _get_pixels, _clear_pixels).
    e. -s EXPORTED_RUNTIME_METHODS=[...] → Gives us access to important runtime helpers from JavaScript (cwrap to call C++ functions, and HEAPU8 for working with memory).
3. When this is finished 2 new files will be created
    a. drawing.js → JavaScript “glue code” that connects the browser with WebAssembly.
    b. drawing.wasm → The actual compiled WebAssembly program.

Step 5: Running the program

After compiling the program you can run it and view the output in a browser.

1. Start a local server with Python

```
python -m http.server 8080
```

2. Open your browser and navigate to

```
http://localhost:8080/
```

3. Explanation
    a. The HTTP server is necessary because browsers block direct loading of WASM files from the filesystem.
    b. You can now interact with the drawing board application.

Step 6: Understanding the code

All of the explanations and documentation of all variables and functions are in the code files in the GitHub repository: https://github.com/Ruan-le-Roux/WebAssembly.

Step 7: Notes & Tips

1. Browser Cache: If you see errors like HEAPU8 not exported, clear your browser cache or do a hard refresh.
2. Code Understanding:
    a. WASM allows fast pixel manipulation in C++ while keeping JS minimal.
    b. cwrap is crucial for making the interface between JS and WASM seamless.
3. Extensions: You can add more brushes, shapes, or save functionality to expand this tutorial.

🎉 Congratulations!

You should now have a fully working WebAssembly Drawing Board application.

It runs inside your browser like a normal web page, but all the drawing logic is actually powered by C++ code compiled to WebAssembly.

References

1. MDN Web Docs. (2019). WebAssembly. [online] Available at: https://developer.mozilla.org/en-US/docs/WebAssembly.
2. emscripten.org. (n.d.). Download and install — Emscripten 3.1.25-git (dev) documentation. [online] Available at: https://emscripten.org/docs/getting_started/downloads.html.
3. Tutorialspoint.com. (2025). WebAssembly Tutorial. [online] Available at: https://www.tutorialspoint.com/webassembly/index.htm [Accessed 21 Sep. 2025].
4. Python.org. (2019). Python Releases for Windows. [online] Available at: https://www.python.org/downloads/windows/.
5. webassembly.org. (n.d.). WebAssembly. [online] Available at: https://webassembly.org/.