

IMY 320 Individual Research and Writing

Name: Sipho Sehlapelo

Student Number: 23535212

An Introduction to WebAssembly for BSc Computer Science Students

The foundation of the modern-day web is built on these 3 trios HTML, CSS and JavaScript. While JavaScript is a powerful and versatile language, its performance can fall short for computationally intensive tasks such as complex simulations, game engines or even multimedia editing. This is where WebAssembly comes in. WebAssembly (abbreviated as Wasm or WASM) is a breakthrough web standard that is designed to solve this problem. It is defined as a portable binary instruction format for a stack-based virtual machine, which is created to be an efficient compilation target for languages like C, C++ and Rust. So, to put it into simple terms it allows developers such as yourself to write heavy and speed hungry code for their applications in these fast languages (C or C++ or Rust) and run them in the browser almost as fast as if they were running directly on a computer. It is important to note that WASM is not a replacement for JavaScript, it was designed to work in harmony with it. This enables both of them to call each other's functions seamlessly and creates a powerful, hybrid programming model for the web. (WebAssembly)

For us as BSc computer science students, understanding WASM is important for engaging with high performance web applications in the future. As the industry is quickly moving towards complex web applications that were once only possible as native desktop software, this is a trend known as Progressive Web Apps (PWAs). WASM is the key technology that is making this possible. Knowing that it opens doors to advanced areas such as cloud computing (using WASI to run code securely on servers) and blockchain (for smart contracts). WASM shows you that you can connect low-level system programming with modern web development. Which is a valuable skill that lets you go beyond just using web technology to actually be innovating with it. (WASI.)

This ties directly to our degree. In COS 212(Data Structures and Algorithms) we study how efficient algorithms are and with WASM, we can actually implement these algorithms in C++ and run them on the web. In COS 216 we learned core web technologies (HTML, CSS, JS), which gives us the essential foundation for integrating and interacting with a WASM module.

In COS 333 (Programming Languages), we studied compilation and language design, which directly connects to WASM as a compilation target. WASM is the bridge that lets us use everything we know from low-level logic to web design to building powerful apps in the browser. The next tutorial will give you a hands-on introduction and demonstration to how it works. (Wong)

Tutorial

Since the target audience for this assignment is primarily 2nd year Computer science students I will be using C++ for our fast language as students have some experience from since first year.

Setting Up Your Environment

Step 1: Is to install Emscripten

What is Emscripten? This is the tool that is going to help turn your C++ code into WASM, so it will compile your C++ code into WASM (.wasm) including some JavaScript glue code.

So, the first thing is to Install **Python** and **CMAKE** (if you haven't already installed them):
Just to note you won't be using these languages directly but emscripten needs them.

Python: <https://www.python.org/downloads/>

Navigate to that website and then follow these steps to install it.

On the page, you'll see a **big yellow button** that says something like:
"Download Python 3.x.x" (the latest version).

Click it.

After downloading, **double-click** the installer file (python-3.x.x.exe).

On the first screen and this **VERY IMPORTANT**:
Tick the box **"Add Python to PATH"** at the bottom.

Click **Install Now** and wait for it to finish.

Then Navigate to you command line (cmd) and type in "python --version" it shows you the version which shows you that you have successfully installed python.

```
C:\Users\letha>python --version
Python 3.12.3
```

CMAKE: <https://cmake.org/download/>

Navigate to that website and then follow these steps to install it.

Scroll down to **Latest Release**. Under “Windows (win64-x64 Installer)”, click the link to download. (It will be something like cmake-3.xx.x-windows-x86_64.msi).

After downloading, double-click the installer.

Keep the default options and click **Next** → **Next** → **Install**.

Then Navigate to you command line (cmd) and type in “cmake --version” it shows you the version which shows you that you have successfully installed cmake

```
C:\Users\letha>cmake --version
cmake version 4.1.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

GIT: <https://git-scm.com/downloads/win>

git is what you going to use to clone the repo for the **Emscripten**.

Navigate to that website and then follow these steps to install it.

Run the installer file (Git-x.x.x-64-bit.exe).

The installer has a lot of steps — but you can **keep everything default** and just click **Next** each time.

- The only choice: When it asks, “*Choosing the default editor used by Git*”, just leave it on **Vim** (default) or change it to **Notepad** if you’re more comfortable.

Finish installation.

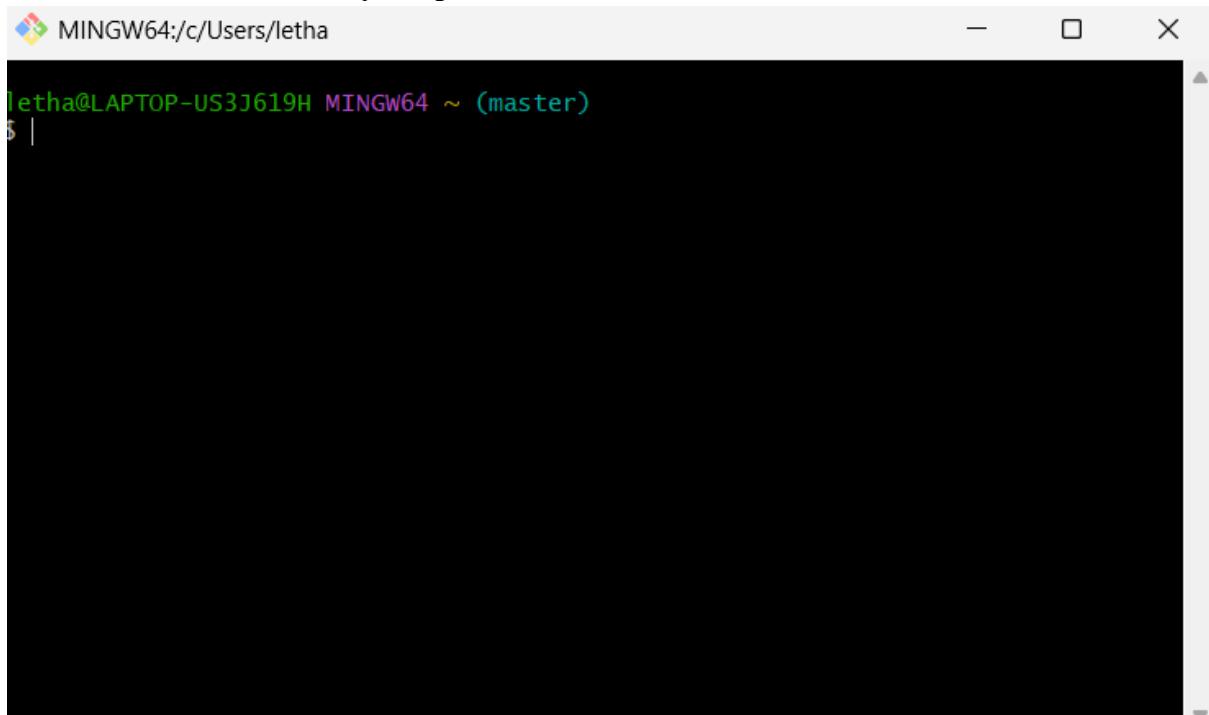
Then Navigate to you command line (cmd) and type in “git --version” it shows you the version which shows you that you have successfully installed git.

```
C:\Users\letha>git --version
git version 2.47.0.windows.2
```

Now we will install the main thing we wanted **Emscripten**.

First thing open Git Bash (it comes with git when install it) so search it on your pc it

should look like this after you open it



```
MINGW64:/c/Users/letha  
letha@LAPTOP-US3J619H MINGW64 ~ (master)  
$ |
```

Then enter these commands in the terminal of git bash:

1: Clone Emscripten

1. Open **Git Bash** or **Command Prompt**.
2. Run:

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

2: Install the SDK

Normally, you can run the following inside Git Bash:

```
./emsdk install latest  
./emsdk activate latest  
source ./emsdk_env.sh
```

This downloads and sets up the latest Emscripten compiler.

Important: If you get a Python error in Git Bash

While I followed those steps I got this error:

Python was not found; run without arguments to install from the Microsoft Store...

```
letha@LAPTOP-US3J619H MINGW64 ~ (master)
$ git clone https://github.com/emscripten-core/emsdk.git
Cloning into 'emsdk'...
remote: Enumerating objects: 4568, done.
remote: Counting objects: 100% (31/31), done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 4568 (delta 22), reused 12 (delta 12), pack-reused 4537 (from 2)
Receiving objects: 100% (4568/4568), 2.52 MiB | 6.91 MiB/s, done.
Resolving deltas: 100% (3018/3018), done.

letha@LAPTOP-US3J619H MINGW64 ~ (master)
$ cd emsdk

letha@LAPTOP-US3J619H MINGW64 ~/emsdk (main)
$ ./emsdk install latest
Python was not found; run without arguments to install from the Microsoft Store,
or disable this shortcut from Settings > Apps > Advanced app settings > App exe
cution aliases.
```

If this happens, don't panic. Just switch to **Command Prompt (or PowerShell)** and do the installation there instead:

```
cd C:\Users\<YourUsername>\emsdk
```

```
emsdk install latest
```

```
emsdk activate latest
```

```
emsdk_env.bat
```

- emsdk install latest → downloads the tools (Node.js, Python, LLVM, etc.)
- emsdk activate latest → makes this version the active one.
- emsdk_env.bat → sets up environment variables so you can use emcc in this terminal session.

```
C:\Users\letha>cd emsdk

C:\Users\letha\emsdk>emsdk install latest
Resolving SDK alias 'latest' to '4.0.15'
Resolving SDK version '4.0.15' to 'sdk-releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'
Installing SDK 'sdk-releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'..
Installing tool 'node-22.16.0-64bit'..
Downloading: C:/Users/letha/emsdk/downloads/node-v22.16.0-win-x64.zip from https://storage.googleapis.com/webassembly/emscripten-releases-builds/deps/node-v22.16.0-win-x64.zip, 37497627 Bytes
Unpacking 'C:/Users/letha/emsdk/downloads/node-v22.16.0-win-x64.zip' to 'C:/Users/letha/emsdk/node/22.16.0_64bit'
Done installing tool 'node-22.16.0-64bit'..
Installing tool 'python-3.13.3-64bit'..
Downloading: C:/Users/letha/emsdk/downloads/python-3.13.3-win-amd64.zip from https://storage.googleapis.com/webassembly/emscripten-releases-builds/deps/python-3.13.3-win-amd64.zip, 29736374 Bytes
Unpacking 'C:/Users/letha/emsdk/downloads/python-3.13.3-win-amd64.zip' to 'C:/Users/letha/emsdk/python/3.13.3_64bit'
Done installing tool 'python-3.13.3-64bit'..
Installing tool 'releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'..
Downloading: C:/Users/letha/emsdk/downloads/b412b6307e541b93dd93f01b61181e15c17302ec-wasm-binaries.zip from https://storage.googleapis.com/webassembly/emscripten-releases-builds/win/b412b6307e541b93dd93f01b61181e15c17302ec-wasm-binaries.zip, 633557034 Bytes
Unpacking 'C:/Users/letha/emsdk/downloads/b412b6307e541b93dd93f01b61181e15c17302ec-wasm-binaries.zip' to 'C:/Users/letha/emsdk/upstream'
Done installing tool 'releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'..
Done installing SDK 'sdk-releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'..

C:\Users\letha\emsdk>emsdk activate latest
Resolving SDK alias 'latest' to '4.0.15'
Resolving SDK version '4.0.15' to 'sdk-releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit'
Setting the following tools as active:
  node-22.16.0-64bit
  python-3.13.3-64bit
  releases-b412b6307e541b93dd93f01b61181e15c17302ec-64bit

Next steps:
- Consider running 'emsdk activate' with --permanent or --system
  to have emsdk settings available on startup.
Adding directories to PATH:
PATH += C:\Users\letha\emsdk
PATH += C:\Users\letha\emsdk\upstream\emscripten

Setting environment variables:
PATH = C:\Users\letha\emsdk;C:\Users\letha\emsdk\upstream\emscripten;C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Common Files\
Setting environment variables:
PATH = C:\Users\letha\emsdk;C:\Users\letha\emsdk\upstream\emscripten;C:\Program Files\Common Files\Oracle\Java\javapath;C:\Program Files (x86)\Common Files\
Oracle\Java\javapath;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Windows\System32\OpenSSH\;C:\Program Files\dotnet\;C:\Program Files\HP\HP One Agent;C:\Program Files\MariaDB 11.3\bin;C:\Program Files\nodejs\;C:\ProgramData\chocolatey\bin;C:\Program File
s\Git\cmd;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files\
Docker\Docker\resources\bin;C:\Program Files\swi\bin;C:\Program Files (x86)\BaseX\bin;C:\Program Files\CMake\bin;C:\Users\letha\AppData\Local\Programs\Pyt
hon\Python312\Scripts\;C:\Users\letha\AppData\Local\Programs\Python\Python312\;C:\Users\letha\AppData\Local\Programs\Python\Launcher\;C:\Program Files\Java\
jdk1.8.0_292\bin;C:\Users\letha\AppData\Local\Microsoft\WindowsApps;C:\Users\letha\AppData\Local\Programs\Microsoft VS Code\bin;C:\Users\letha\AppData\Roami
ng\npm;C:\Users\letha\AppData\Local\GitHubDesktop\bin;C:\Users\letha\lmsstudio\bin;C:\Users\letha\AppData\Local\Programs\MikTeX\miktex\bin\x64\;C:\Users\le
tha\AppData\Local\Programs\cursor\resources\app\bin;C:\Users\letha\AppData\Local\Programs\Ollama
EMSDK = C:/Users/letha/emsdk
EMSDK_NODE = C:/Users/letha/emsdk/node/22.16.0_64bit/bin/node.exe
EMSDK_PYTHON = C:/Users/letha/emsdk/python/3.13.3_64bit/python.exe
Clearing existing environment variable: EMSDK_PY
The changes made to environment variables only apply to the currently running shell instance. Use the 'emsdk_env.bat' to re-enter this environment later, or
if you'd like to register this environment permanently, rerun this command with the option --permanent.

C:\Users\letha\emsdk>emsdk_env.bat
Setting up EMSDK environment (suppress these messages with EMSDK_QUIET=1)
Setting environment variables:
Clearing existing environment variable: EMSDK_PY
```

3: Test Installation

After activation, I tested with:

emcc -v

If you see version information for emcc, Emscripten is ready!

```
C:\Users\letha\emsdk>emcc -v
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 4.0.15 (09f52557f0d48b65b8c724853ed8f4e8bf80e669)
clang version 22.0.0git (https://github.com/llvm/llvm-project 3388d40684742e950b3c5d1d2daf5a40695cfc1)
Target: wasm32-unknown-emscripten
Thread model: posix
InstalledDir: C:\Users\letha\emsdk\upstream\bin
```

Step 2: Create Your Project Folder

We will be making a Rock Paper Scissors simple game for the purpose of this tutorial

Before we start writing any code, we need a **place to organize everything**. Think of this like setting up a locker before putting your books inside — you want everything neat and easy to find.

2.1 Make a Project Folder

- Go to your desktop or Documents (anywhere you like).

- Right-click → **New** → **Folder**.
- Name it: **wasm-rps-game**

Why?

This keeps all your files for the Rock-Paper-Scissors game in one place. Later, when you upload to GitHub, it's just this folder.

2.2 Create the Files Inside

Now inside this folder, we'll make **three important files**.

1. **rps.cpp** → our **C++ game logic**
 - This is where the *brains of the game* live.
 - It will decide: if you pick Rock and the computer picks Scissors → you win.
 - We write it in C++ so we can compile it into WebAssembly (WASM).
2. **index.html** → our **webpage**
 - HTML is the *skeleton of the website*.
 - This is the page you'll actually open in your browser.
 - It will have buttons ("Rock", "Paper", "Scissors") and a place to show the result.
3. **script.js** → our **JavaScript bridge**
 - JavaScript is like the *translator* between the web browser and your C++/WASM code.
 - It will:
 - Load the WASM file compiled from rps.cpp.
 - Send your choice (Rock, Paper, or Scissors) into WASM.
 - Get the result back and show it on the webpage.

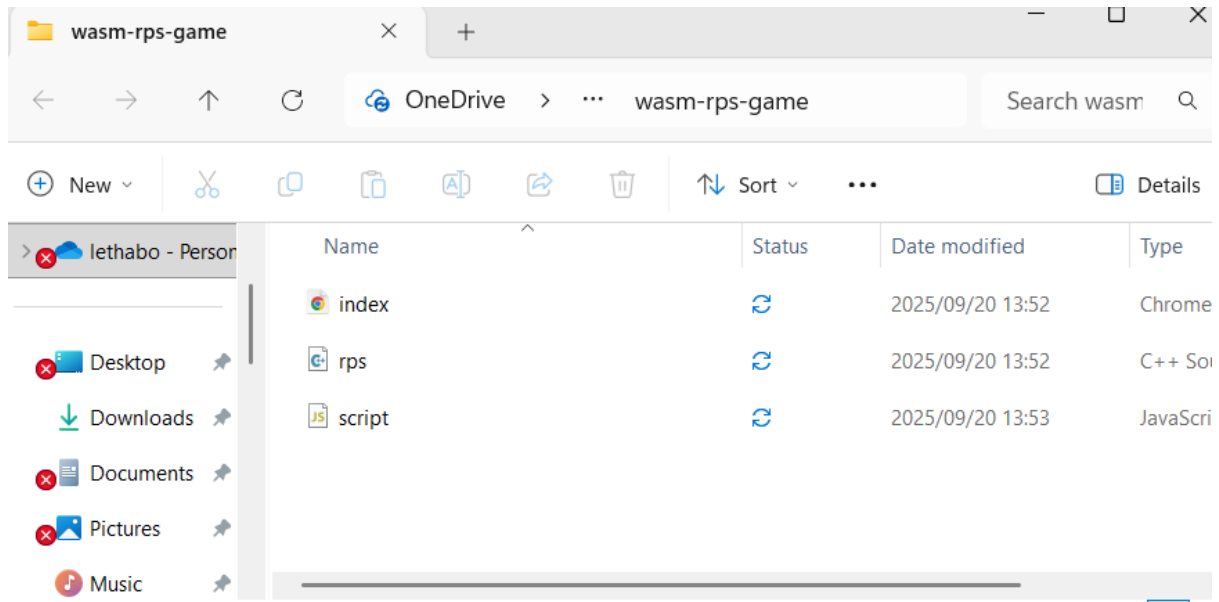
Why separate files?

- Keeping logic (C++) separate from presentation (HTML) and control (JavaScript) makes it easier to **understand, debug, and explain**.
- This also matches what you've learned in COS 216: separation of concerns (HTML = structure, CSS = style, JS = behaviour). Here we're adding **C++ = performance logic**.

Screenshot to take: File Explorer showing the three files inside wasm-rps-game (empty for now).

2.3 Which file runs where?

- `rps.cpp` → runs in WASM (compiled to `.wasm` by Emscripten).
- `script.js` → runs in your browser, talking to WASM.
- `index.html` → runs in your browser, showing you buttons and results.



Now just to note this is specific to the project we are building, the files would be different for another project.

Step 3: Write the C++ Game Logic (rps.cpp)

```
#include <string>
#include <iostream>
#include <cstdlib>

using namespace std;
extern "C" {
    const char* play(int player, int computer) {
        if (player == computer) {
            return "It's a tie!";
        } else if ((player == 0 && computer == 2) ||
                    (player == 1 && computer == 0) ||
                    (player == 2 && computer == 1)) {
            return "You win!";
        } else {
            return "Computer wins!";
        }
    }
}
```

Breakdown

#include <string>

This includes the string library in C++.

We don't actually use `std::string` here (we return plain C-style strings), but it's often included for string handling.

In bigger projects, if you wanted to pass full sentences, you'd use this.

extern "C" { ... }

This is VERY important.

Normally, when you write C++ functions, the compiler changes their names internally (a process called name mangling).

But WebAssembly needs to know exactly what the function name is so JavaScript can call it.

extern "C" tells the compiler:

“Do not mangle these function names. Keep them plain so other languages (like JavaScript) can see them.”

Without this, JavaScript wouldn't be able to find `play()` when we try to call it.

`const char* play(int player, int computer)`

This declares our function `play`.

`int player` = the number for the player's choice.

`int computer` = the number for the computer's choice.

`const char*` = means the function will return a pointer to a text string like "You Win!".

Why not just string?

WASM works better with simple types (`int`, `float`, `char*`).

Returning `const char*` makes it easy to send back plain text like "Draw".

`if (player == computer) return "Draw";`

If both the player and computer chose the same number → it's a draw.

Example: `player = 0` (Rock), `computer = 0` (Rock) → "Draw".

`(player == 0 && computer == 2) || ...`

This checks all the winning conditions for the player:

Rock (0) beats Scissors (2).

Paper (1) beats Rock (0).

Scissors (2) beats Paper (1).

If any of these are true → return "You Win!".

`return "Computer Wins!";`

If it's not a draw, and the player didn't win → the computer must have won.

Example: `player = 0` (Rock), `computer = 1` (Paper) → "Computer Wins!".

Step 4: Compile C++ → WASM

Open your terminal in your folder structure and run this command:

```
emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=['_play'] -o rps.js
```

What this does

1. **emcc rps.cpp**

- Tells Emscripten to compile your C++ file (rps.cpp).

2. **-s WASM=1**

- Ensures output is a WebAssembly .wasm file (not just JavaScript).

3. **-s EXPORTED_FUNCTIONS=['_play']**

- By default, Emscripten hides your functions.
- This tells it: *“Make the C++ function play() available to JavaScript.”*
- Notice the underscore (_play) → Emscripten adds _ in front of C++ function names internally.

4. **-o rps.js**

- Produces two files:
 - rps.js → “glue code” that knows how to load and talk to WASM.
 - rps.wasm → the actual WebAssembly binary compiled from C++.

While running this command I ran into a problem

```
PS C:\Users\letha\OneDrive\Desktop\wasm-rps-game> emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=['_play'] -o rps.js
>>
emcc : The term 'emcc' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or
if a path was included, verify that the path is correct and try again.
At line:1 char:1
+ emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=['_play'] -o rps.js
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (emcc:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException
```

The issue for me was Powershells command terminal so I needed to switch to the CMD command terminal to do that just follow these steps

CTRL `

the terminal will pop up and you will see a + next to where it says Powershell and click it and select command prompt

Once switched to this run this to check If emcc is there or not:

```
cd C:\Users\<YourUsername>\emsdk
```

```
emcc -v
```

when it shows this that means it's there on your system:

```
C:\Users\letha\emsdk>emcc -v
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 4.0.15 (09f52557f0d48b6
5b8c724853ed8f4e8bf80e669)
clang version 22.0.0git (https://github.com/llvm/llvm-project 3388d40684742e950b3c5d1d2d4fe5a40
695cfc1)
Target: wasm32-unknown-emsripten
Thread model: posix
InstalledDir: C:\Users\letha\emsdk\upstream\bin
```

Then Navigate back to the folder structure:

```
cd ..
```

```
cd \OneDrive\Desktop\wasm-rps-game>
```

so we can run the initial command again so run this again:

```
emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=['_play'] -o rps.js
```

I then ran into another issue which had is a **tiny syntax issue** with how Windows (and PowerShell/CMD) handle quotes.

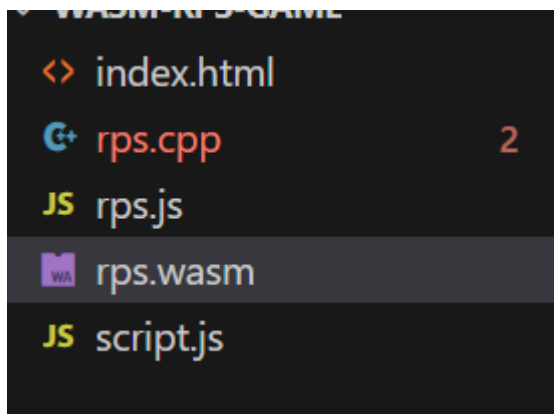
```
C:\Users\letha\OneDrive\Desktop\wasm-rps-game>emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=['_
_play'] -o rps.js
shared:INFO: (Emscripten: Running sanity checks)
emcc: error: invalid export name: "['_play']"
```

To fix run this now:

```
emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=["_play"] -o rps.js
```

```
C:\Users\letha\OneDrive\Desktop\wasm-rps-game>emcc rps.cpp -s WASM=1 -s EXPORTED_FUNCTIONS=["_
play"] -o rps.js
```

And it should work if your folder structure looks like this:



Step 5: Write you html code(index.html)

This is the front-end

write what is specific to your simple game or project or you can ask ChatGPT to generate it for if need be but just remind this is just for this tutorial so it simple, depending on what you are building the complexity may change and AI generated code

may not help so here is my code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Rock Paper Scissors (WASM)</title>
</head>
<body>
  <h1>Rock Paper Scissors</h1>
  <button onclick="playGame(0)" disabled>Rock</button>
  <button onclick="playGame(1)" disabled>Paper</button>
  <button onclick="playGame(2)" disabled>Scissors</button>
  <p id="result">Loading WASM...</p>

  <!-- Pre-configure Module BEFORE loading rps.js -->
  <script>
    var Module = {
      onRuntimeInitialized: function() {
        console.log("WASM Loaded!");
        // Enable buttons once WASM is ready
        document.querySelectorAll('button').forEach(btn => btn.disabled = false);
        document.getElementById("result").innerText = "Ready to play!";
      }
    };
  </script>

  <!-- Load rps.js (Emscripten glue code) -->
  <script src="rps.js"></script>

  <!-- Load your script.js AFTER rps.js -->
  <script src="script.js"></script>
</body>
</html>
```

Pretty basic but serves for the purpose of this assignment.

But here is the breakdown of the code:

```
var Module = { ... }
```

When you compile with Emscripten, it generates rps.js (the glue code).

That file looks for a global object called Module.

Here, we define Module and give it an onRuntimeInitialized callback.

Meaning:

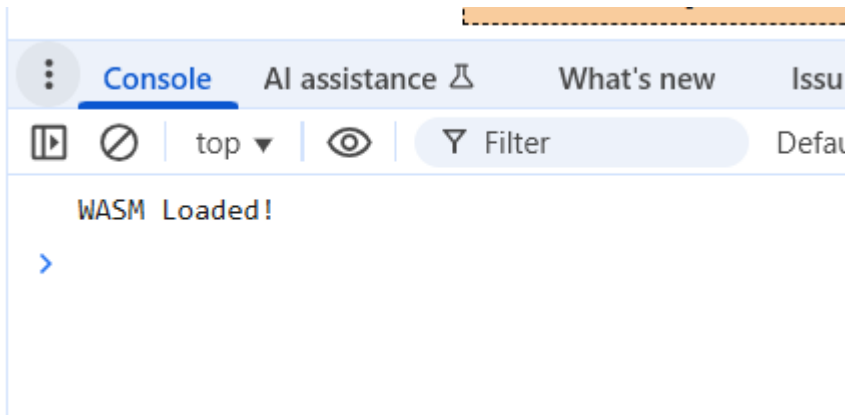
As soon as WASM is fully loaded into memory, it will run this function:

```
console.log("WASM Loaded!");
```

```
document.querySelectorAll('button').forEach(btn => btn.disabled = false);
```

```
document.getElementById("result").innerText = "Ready to play!";
```

- The first line is for debugging (so we know everything is ready).



- The second line enables the buttons (before this, they were disabled).
- The third line updates the page, so the user sees “Ready to play!”.

This ensures the user cannot play until WASM is ready.

Step 6: Create script.js

This will load the WASM and connect it to the buttons.

```
// Don't redefine Module - it's already created by rps.js
// The Module object is configured in index.html before rps.js loads

function playGame(playerChoice) {
  // Check if WASM is loaded and the function is available
  if (!Module._play) {
    document.getElementById("result").innerText = "WASM not loaded yet!";
    return;
  }

  let computerChoice = Math.floor(Math.random() * 3);

  // Since your WASM doesn't export string conversion functions,
  // let's implement the game logic in JavaScript instead
  let result;
  if (playerChoice == computerChoice) {
    result = "It's a tie!";
  } else if ((playerChoice == 0 && computerChoice == 2) ||
             (playerChoice == 1 && computerChoice == 0) ||
             (playerChoice == 2 && computerChoice == 1)) {
    result = "You win!";
  } else {
    result = "Computer wins!";
  }

  document.getElementById("result").innerText =
    "You: " + ["Rock", "Paper", "Scissors"][playerChoice] +
    " | Computer: " + ["Rock", "Paper", "Scissors"][computerChoice] +
    " → " + result;
}
```

Breakdown

```
function playGame(playerChoice) { ... }
```

This function runs whenever the user clicks one of the buttons (Rock, Paper, or Scissors).

```
if (!Module._play) {
```

```
  document.getElementById("result").innerText = "WASM not loaded yet!";
```

```
  return;
```

```
}
```

This is a safety check.

If `_play` (the C++ function) isn't available yet, the function stops.

Instead of crashing, the page shows “WASM not loaded yet!”.

let computerChoice = Math.floor(Math.random() * 3);

Math.random() → gives a number between 0 and 1.

Multiply by 3 → gives 0, 1, or 2 (with decimals).

Math.floor() → rounds it down to an integer.

So, this picks the computer’s choice randomly:

0 = Rock

1 = Paper

2 = Scissors

let result;

if (playerChoice == computerChoice) {

result = "It's a tie!";

} else if ((playerChoice == 0 && computerChoice == 2) ||

(playerChoice == 1 && computerChoice == 0) ||

(playerChoice == 2 && computerChoice == 1)) {

result = "You win!";

} else {

result = "Computer wins!";

}

This block decides who won the round.

If both choices are the same → tie.

If your choice beats the computer → you win.

Otherwise → the computer wins.

document.getElementById("result").innerText =

"You: " + ["Rock", "Paper", "Scissors"][playerChoice] +


```
" | Computer: " + ["Rock", "Paper", "Scissors"][computerChoice] +  
" → " + result;
```

This updates the webpage with the full game result.

It shows:

Your choice (Rock, Paper, or Scissors).

The computer's choice.

The outcome (Win, Lose, or Tie).

Example:

You: Rock | Computer: Scissors → You win!

Why do we need this file?

C++ (rps.cpp) → the brain of the game.

HTML (index.html) → the face of the game (buttons + display).

JavaScript (script.js) → the translator/bridge.

Without script.js, the HTML page wouldn't know how to connect button clicks to the WASM module.

Step 7: Run in your Browser

You **can't just open index.html** because WASM needs a server.

Simplest way → use Python to serve files:

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

```
C:\Users\letha\OneDrive\Desktop\wasm-rps-game>python -m http.server  
Serving HTTP on :: port 8000 (http://[::]:8000/) ...  
█
```

Then open <http://localhost:8000> in your browser.

Rock Paper Scissors

You: Paper | Computer: Scissors → Computer wins!

Then just click the buttons to play the game.

References

WASI. n.d. <<https://wasi.dev/>>.

WebAssembly. 2025 September 2025. <<https://webassembly.org/>>.

Wong, YL (Yan). *Lecture 3.1 Web Development trends*. n.d.

ALL FILES AND DOCUMENTS ARE IN THIS REPO:

<https://github.com/Letha456/wasm-rps-game>