WebAssembly (WASM)

Compte-rendu de la présentation de Thierry Gayet, LinuxMaine – 25/10/2025

Table des matières

1 Introduction à WebAssembly....2
Définition....2
2 Principes de fonctionnement....3
Compilation et exécution....3
Format binaire vs Assembly....3
Langages supportés....3
3 Cas d'usage....4
Dans le navigateur....4
Hors navigateur....4
4 Avantages de WebAssembly....4
Performance....4
Sécurité....4
Portabilité....4
Flexibilité....4
5 Emscripten : L'outil de compilation....5
Présentation....5
Installation....5
Compilation basique....5
Composants intégrés....5
6 Travaux pratiques....6
Helloworld....6
Call-wasm-function-from-js-v1....7
Call-wasm-function-from-js-v2....8
Array-and-Tab....9
7 Cas d’usage de emscripten (exemples)....11
Exécution hors navigateur (avec wasmer)....11
Compiler plusieurs sources....11
Générer une bibliothèque, l’utiliser....12
Utilisation avec Node.js....13
8 Outils d'exécution (Runtimes)....13
Wasmer....13
Wasmtime....13
WasmEdge....14
9 WebAssembly et le noyau Linux....14
WasmLinux....14
10 Glossaire technique....14
11 L'avenir de WebAssembly....15
Perspectives....15
Citation de Brendan Eich (créateur de JavaScript)....15
12 Conclusion....15

1 Introduction à WebAssembly

Définition

WebAssembly (abrégé Wasm) est un standard du World Wide Web pour le développement d'applications, conçu pour offrir des performances supérieures à JavaScript. Il consiste en :

- Un bytecode binaire portable
- Sa représentation textuelle
- Un environnement d'exécution en bac à sable (sandbox) compatible avec JavaScript
- Une exécution possible dans et hors navigateur

1 Historique clé

- **Juin 2015** : Première annonce de WebAssembly
- **Mars 2016** : Préversion dans Chrome, Edge et Firefox
- **Octobre 2016** : Version candidate binaire
- **Mars 2017** : Livraison par défaut dans les navigateurs

2 Logo et symbologie

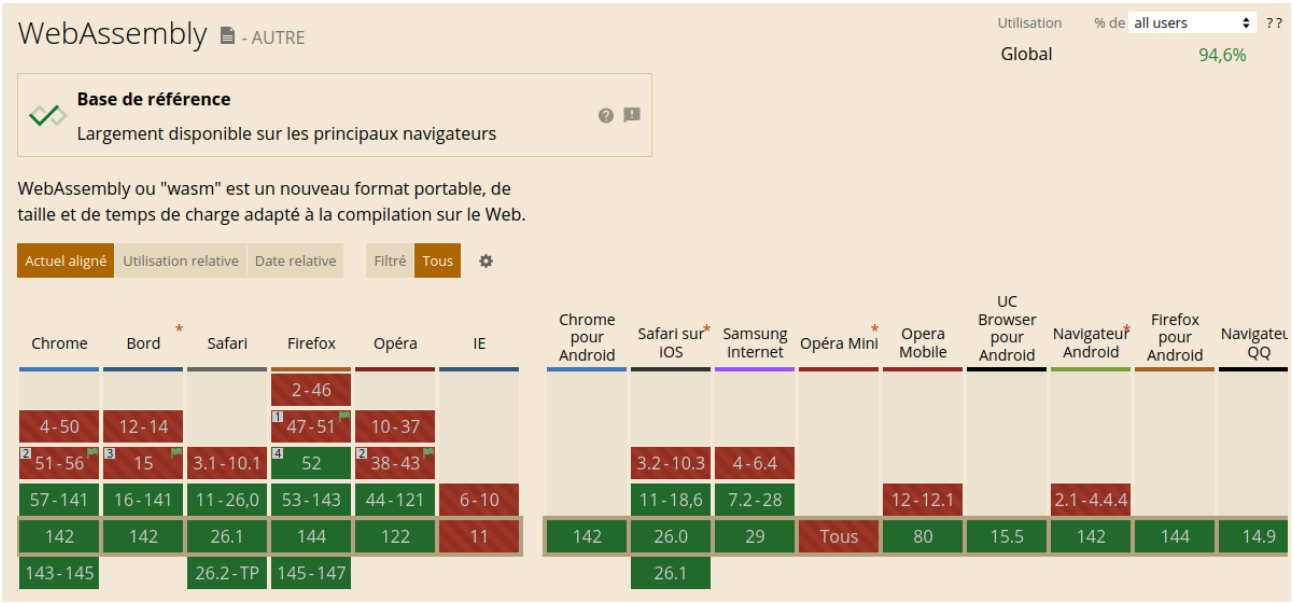
Le logo WASM, sélectionné en février 2017, s'inspire d'un circuit intégré (IC) avec :

- Une encoche supérieure rappelant l'orientation d'un IC
- Une référence au bas niveau (structures de données, mémoire, CPU)
- Une couleur complémentaire à JavaScript pour souligner leur complémentarité



Figure 1: Logo WASM

3 Les navigateurs supportés



https://caniuse.com/wasm

2 Principes de fonctionnement

Compilation et exécution

WebAssembly transforme le code source de langages de haut niveau en bytecode binaire basé sur un arbre syntaxique abstrait (AST). Ce processus comprend :

1. **Compilation** : Le code source (C, C++, Rust, etc.) est compilé en bytecode WASM
2. **Téléchargement et interprétation** : Le bytecode est chargé par le navigateur
3. **Exécution rapide** : Le navigateur compile le bytecode en langage machine natif
4. **Interopérabilité** : Communication fluide avec JavaScript via une API

Format binaire vs Assembly

Contrairement à JavaScript qui nécessite un parsing textuel, WebAssembly utilise un format AST binaire qui :

- Réduit considérablement le temps de chargement
- Offre des performances proches du code natif
- Permet une représentation compacte du code

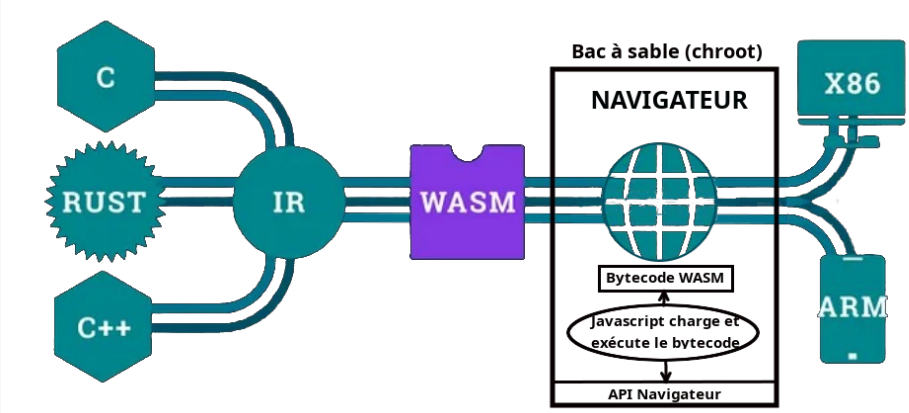


Figure 2: Web Assembly, schéma de fonctionnement

Langages supportés

WebAssembly peut être généré à partir de nombreux langages :

- **C/C++** : via Emscripten (le plus mature)
- **Rust** : avec le module wasm-bindgen
- **Go** : support natif
- **Java** : via TeaVM, JWebAssembly, Bytecoder
- **Kotlin** : backend LLVM Kotlin Native
- **.NET, C#, F#** : via Blazor
- **FreePascal** : avec pas2js
- Et bien d'autres : Ada, Lua, OCaml, PHP, Python, Ruby, Fortran

3 Cas d'usage

Dans le navigateur

- Édition de vidéos et musique
- Jeux 3D avec OpenGL/WebGL
- Réalité augmentée et virtuelle
- Reconnaissance d'images
- Applications vectorielles
- Visualisation scientifique
- Interpréteurs de langages
- Outils de développement
- Cryptographie
- Clients de base de données

Hors navigateur

- Traitement côté serveur
- Applications natives mobiles sécurisées
- Remplacement des DLL par des modules WASM
- Appareils sans système d'exploitation (IoT)
- Applications cloud et edge computing

4 Avantages de WebAssembly

Performance

- Conception bas niveau avec exécution très efficace
- Performances proches du code natif
- Réduction du temps de parsing par rapport à JavaScript
- Accès direct à la mémoire et aux registres

Sécurité

- Exécution en environnement sandboxé
- Vérification de l'origine du code
- Restrictions d'accès strictes
- Isolation du système d'exploitation

Portabilité

- Indépendant de l'architecture (ARM, x86, x64)
- Support universel des navigateurs modernes
- Aucun nouveau langage à apprendre
- Modules autonomes et réutilisables

Flexibilité

- Large adoption par les principaux navigateurs
- Open source avec code vérifiable
- Pas de limitation utilisateur liée à la surcharge système
- Compatible avec l'écosystème JavaScript existant

5 Emscripten : L'outil de compilation

Présentation

Emscripten est une chaîne d'outils open source complète permettant de compiler du code C/C++ vers WebAssembly. Il génère généralement trois fichiers :

- Un module binaire `.wasm`
- Un fichier JavaScript "glue" (pont entre C++ et le Web)
- Un fichier `.html` de démonstration

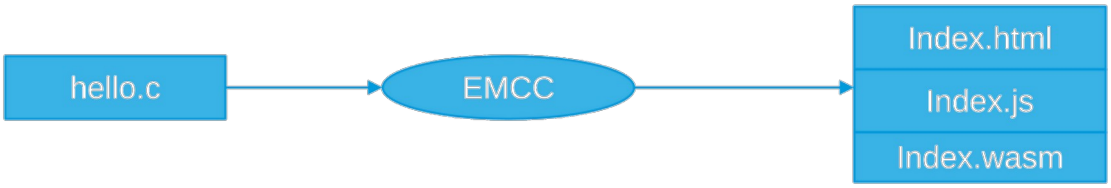


Figure 3: Compiler du code C/C++ avec emscripten

Installation

```
./emsdk update

./emsdk install latest

./emsdk activate latest
```

Compilation basique

```
# Compilation simple

emcc hello.c -s WASM=1 -o hello.html

# Pour un fichier unique (sans serveur HTTP)

emcc -sSINGLE_FILE hello.c -o hello.html

# Pour exécution CLI

emcc hello.c -s STANDALONE_WASM -o hello.wasm
```

Composants intégrés

- **LLVM/Clang** : production du bytecode intermédiaire
- **Binaryen** : optimisation vers WebAssembly
- **Embind et EM_JS** : interactions C++/JavaScript
- **Système de fichiers virtuel** : émulation POSIX dans le navigateur

6 Travaux pratiques

Nota: Repo git des Tps: *git clone* https://framagit.org/linuxmaine/formations

cd ~/formations/WebAssembly/labs/web/

Helloworld

But : Mon premier programme en Webassembly, un genre de « Hello Word ! »

Fichier : Hello.c

```c
#include <stdio.h>

int main()
{
    puts("Hello LinuX Maine");
    return 0;
}
```

Fichier Makefile

```
all: run

.PHONY: all check clean build run

IN_C     := hello.c
OUT_HTML := index.html
OUT_SIMPLE_HTML := index.simplefile.html
OUT_JS   := index.js
OUT_WASM := index.wasm
OUT_ALL  := $(OUT_HTML) $(OUT_JS) $(OUT_WASM)
PORT     := 8080

check:
        @command -v emcc >/dev/null 2>&1 || { \
                echo " - Checking emcc compiler : NOK "; \
                echo "   Please install Emscripten (https://emscripten.org/docs/getting_started/downloads.html)"; \
                echo "   or"; \
                echo "   sudo apt install emscripten"; \
                exit 1; \
        }
        @echo " - Checking emcc compiler : OK"

clean: check
        @echo " - Cleaning "
        @rm -f $(OUT_ALL) $(OUT_SIMPLE_HTML)

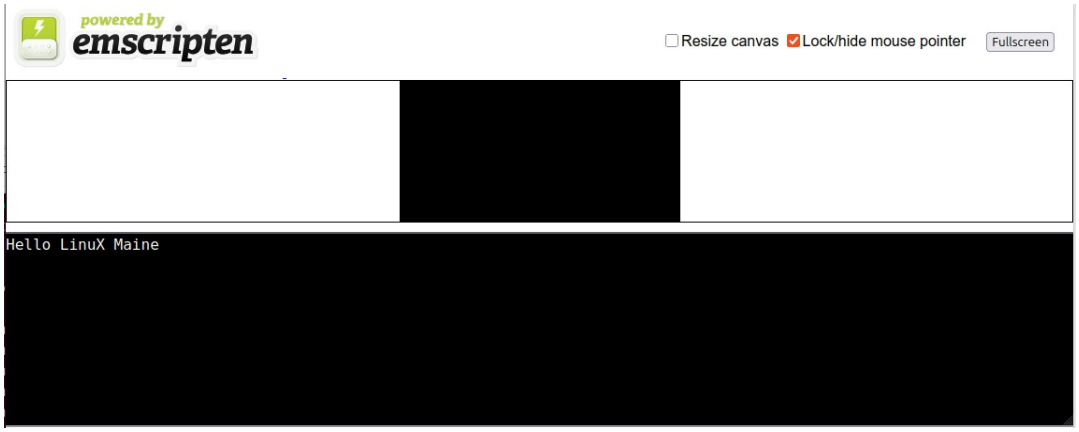
build: clean
        @echo " - Building "
        @emcc $(IN_C) -o $(OUT_HTML)
        @emcc -sSINGLE_FILE $(IN_C) -o $(OUT_SIMPLE_HTML)
        @file $(OUT_WASM)
        @ls -alh

run: build $(OUT_ALL)
        @echo " - Running "
        @echo "--> Open http://localhost:8080/hello.html in your favorite web browser..."
        @python3 -m http.server $(PORT)
```

Bash: make

```
~/formations/WebAssembly/labs/web/helloworld$ make
 - Checking emcc compiler : OK
 - Cleaning
 - Building
index.wasm: WebAssembly (wasm) binary module version 0x1 (MVP)
total 416K
drwxrwxr-x 2 [redacted] 4.0K nov  2 09:56 .
drwxrwxr-x 6 [redacted] 4.0K oct 25 13:52 ..
-rw-rw-r-- 1 [redacted]   81 oct 25 13:52 hello.c
-rw-rw-r-- 1 [redacted] 101K nov  2 09:56 index.html
-rw-rw-r-- 1 [redacted]  89K nov  2 09:56 index.js
-rw-rw-r-- 1 [redacted] 196K nov  2 09:56 index.simplefile.html
-rwxrwxr-x 1 [redacted] 2.8K nov  2 09:56 index.wasm
-rw-rw-r-- 1 [redacted]  928 oct 25 13:52 Makefile
-rw-rw-r-- 1 [redacted]  817 oct 25 13:52 Makefile.old
 - Running
--> Open http://localhost:8080/hello.html in your favorite web browser...
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

Dans la console du navigateur :



Call-wasm-function-from-js-v1

But : Appel d’une fonction C, affichage dans la console.

Fichier : sum.c

```c
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE int sum(int a, int b) {
    return a + b;
}
```

Fichier : script.js

```javascript
Module.onRuntimeInitialized = function() {
    const numA = 32;
    const numB = 10;

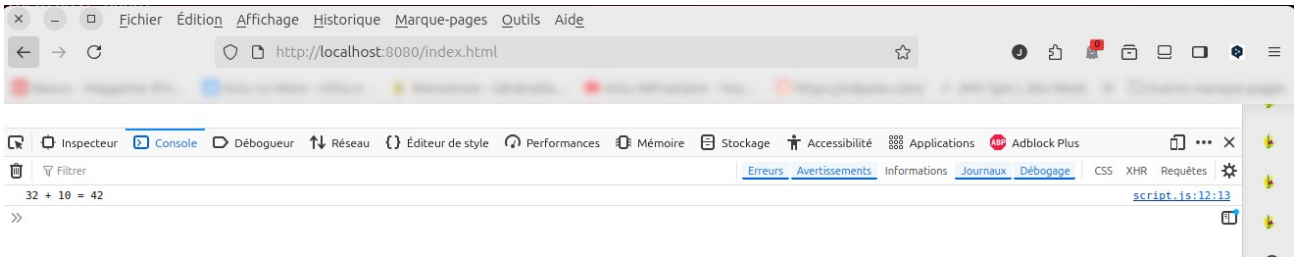
    const result = Module.ccall(
        "sum",                  // Nom de la fonction WASM à appeler
        "number",               // Type de la valeur de retour
        ["number", "number"],   // Types des paramètres de la fonction
        [numA, numB],           // Les paramètres de la fonction
    );

    console.log(`${numA} + ${numB} = ${result}`);
};
```

Fichier : index.html

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <link rel="icon" href="data:,">
        <title>WebAssembly - LinuxMaine - 2.1</title>
    </head>
    <body>
        <script src="sum.wasm.js"></script>
        <script src="script.js"></script>
    </body>
</html>
```

Dans la console du navigateur :



Call-wasm-function-from-js-v2

But : Appel d’une fonction C, affichage dans une page Web.

Fichier : sum.c

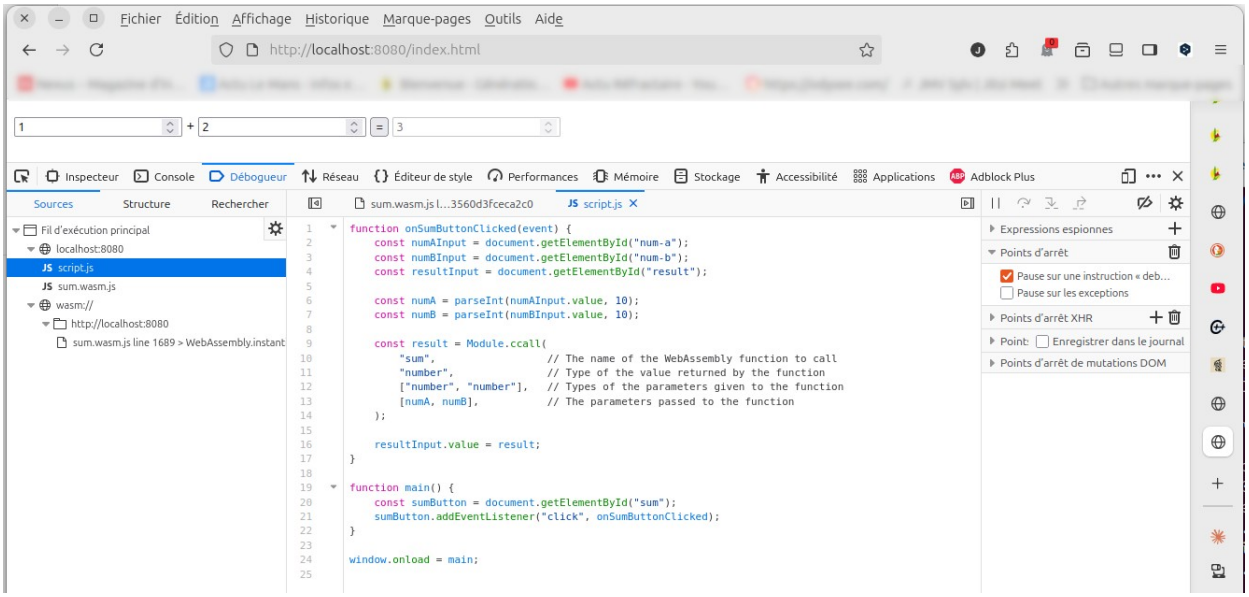
```c
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE int sum(int a, int b) {
    return a + b;
}
```

Fichier index.html

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <link rel="icon" href="data:,">
        <title>WebAssembly - LinuxMaine - 2.2</title>
    </head>
    <body>
        <p>
            <input type="number" id="num-a" />
            +
            <input type="number" id="num-b" />
            <button id="sum">=</button>
            <input type="number" id="result" disabled />
        </p>
        <script src="sum.wasm.js"></script>
        <script src="script.js"></script>
    </body>
</html>
```

Dans le navigateur (l’onglet *Débogueur* affiche le fichier script.js):



Array-and-Tab

But : Seuillage d’une image jpeg en deux couleurs

Fichier : index.html

```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>WebAssembly LinuxMaine - 3</title>
        <link rel="icon" href="data:,">
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <style>
            html, body {
                text-align: center;
                font-family: sans-serif;
            }

            img, canvas {
                border: red solid 1px;
            }

            input {
                vertical-align: middle;
            }
        </style>
    </head>
    <body>
        <p><img src="image.jpg" id="image" /></p>
        <p>
            <label>Seuil : </label>
            <input type="range" min="0" max="255" value="127" id="threshold" />
        </p>
        <p><canvas id="canvas"></canvas></p>
        <script src="threshold.wasm.js"></script>
        <script src="script.js"></script>
    </body>
</html>
```

Fichier : Treshold.c

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

EMSCRIPTEN_KEEPALIVE uint8_t* allocBuffer(int size) {
    return malloc(size * sizeof(uint8_t));
}

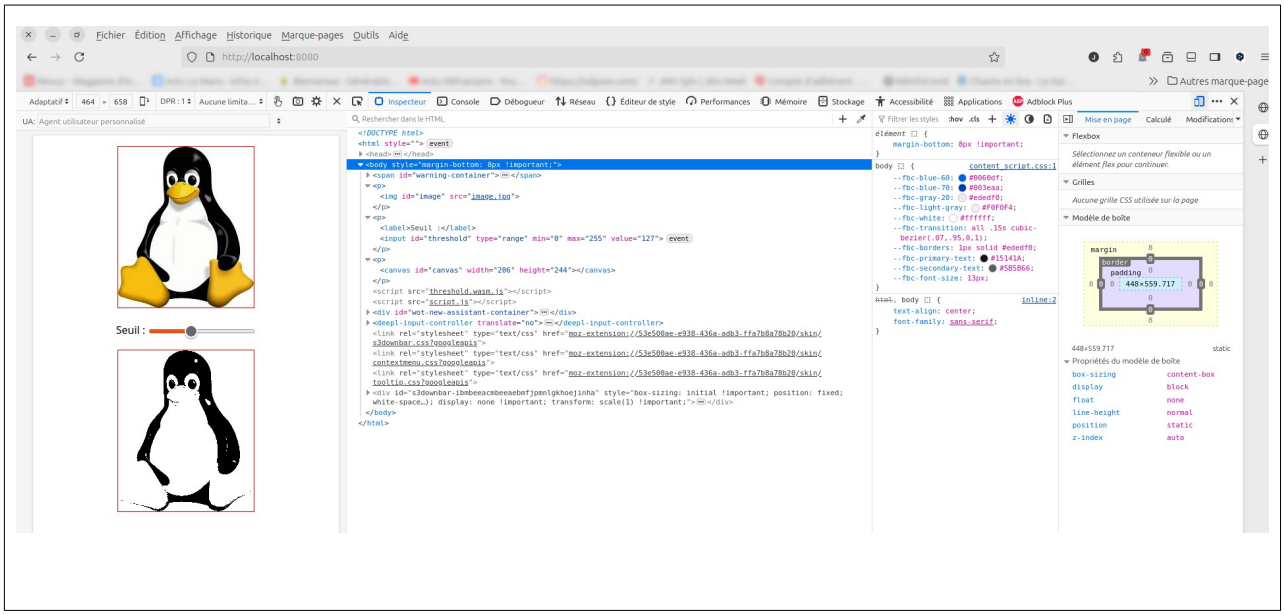
EMSCRIPTEN_KEEPALIVE void freeBuffer(uint8_t* buffer) {
    free(buffer);
}

EMSCRIPTEN_KEEPALIVE void threshold(uint8_t* pixels, int width, int height, int threshold) {
    // There is 4 canals in canvas image data (Red, Green, Blue, Alpha), so its
    // length is: width × height × 4
    int array_length = width * height * 4;

    // i = i + 4 as we loop over pixels, not over channels.
    for (int i = 0 ; i < array_length ; i += 4) {
        int red = pixels[i+0];
        int green = pixels[i+1];
        int blue = pixels[i+2];
        int brightness = 0.299 * red + 0.587 * green + 0.114 * blue;

        // If the computed brightness is under our threshold, the pixel is
        // black else, it is white.
        if (brightness < threshold) {
            pixels[i+0] = 0;
            pixels[i+1] = 0;
            pixels[i+2] = 0;
        } else {
            pixels[i+0] = 255;
            pixels[i+1] = 255;
            pixels[i+2] = 255;
        }
    }
}
```

Dans le navigateur (vue du fichier html) :



Fichier: script.js

Ouvrir | scrip... ~/form... | Enregistrer

```javascript
// Bindings of the WASM exported functions
const API = {
    allocBuffer: Module.cwrap("allocBuffer", "number", ["number"]),
    freeBuffer: Module.cwrap("freeBuffer", "", ["number"]),
    threshold: Module.cwrap("threshold", "", ["number", "number", "number", "number"]),
};

function thresholdImageOnCanvas(image, canvas, threshold=127) {
    // Get the 2D context of the canvas.
    const ctx = canvas.getContext("2d");

    // Draw the original image on the canvas.
    ctx.drawImage(image, 0, 0);

    // Get the pixels that compose the canvas.
    //
    // The imageData object contains:
    //
    // {
    //     width: ...,
    //     height: ...,
    //     data: [red1, green1, blue1, alpha1, red2, green2, blue2, alpha2, …]
    //     //     ---------------------------  ---------------------------
    //     //         PIXEL 1 (top-left)                PIXEL 2
    // }
    //
    // NOTE: `imageData.data` is an Uint8ClampedArray.
    const imageData = ctx.getImageData(0, 0, canvas.width, canvas.height);

    // Allocate a buffer that will contain our pixels in the WASM Module.
    //
    // `pixel_p` is a pointer to that buffer (in the JavaScript world, it is
    // simply a number).
    const pixels_p = API.allocBuffer(imageData.data.length);

    // Copy the pixels into the allocated buffer.
    //
    // `Module.HEAP8` is an Uint8Array that represents the "RAM" (heap) of the
    // WASM Module. The pixel_p pointer is simply an offset on this array.
    Module.HEAP8.set(imageData.data, pixels_p);

    // Call the WASM function that performs the operations on the pixels
    API.threshold(pixels_p, imageData.width, imageData.height, threshold);

    // Copy-back the modified pixels in the imageData.
    imageData.data.set(new Uint8Array(
        Module.HEAP8.buffer, pixels_p, imageData.data.length
    ));

    // Put-back the pixels in the canvas.
    ctx.putImageData(imageData, 0, 0);

    // Free the allocated buffer.
    API.freeBuffer(pixels_p);
}

function onThresholdInput(event) {
    const image = document.getElementById("image");
    const canvas = document.getElementById("canvas");
    const thresholdRange = this;

    thresholdImageOnCanvas(image, canvas, parseInt(thresholdRange.value, 10));
}

function main() {
    const image = document.getElementById("image");
    const canvas = document.getElementById("canvas");
    const thresholdRange = document.getElementById("threshold");

    canvas.width = image.width;
    canvas.height = image.height;

    thresholdRange.addEventListener("input", onThresholdInput);

    // Run the function once when the WASM module is initialized.
    Module.onRuntimeInitialized = function() {
        thresholdImageOnCanvas(image, canvas);
    };
}

window.onload = main;
```

JavaScript | Largeur des tabulations : 8 | Lig 4, Col 8 | OVR

7 Cas d'usage de emscripten[1]

Exécution hors navigateur (avec wasmer[2])

Tp : Cas_usage_1

Installation de wasmer

```
$ curl https://get.wasmer.io -sSfL | sh
$ source ~/.wasmer/wasmer.sh
```

Fichier : hello.c

```c
#include <stdio.h>
int main(int argc, char **argv){
    printf("hello world\n");
}
```

Compilation

```
$ emcc hello.c -o hello.wasm -s WASM=1
```

Exécution avec Wasmer

```
~/formations/WebAssembly/labs/web/Cas_usage_1
:~/formations/WebAssembly/labs/web/Cas_usage_1$ wasmer run hello.wasm
Compiling to WebAssembly
hello world
```

Compiler plusieurs sources

Tp : Cas_usage_2

Fichiers : demolib.hpp, demolib.cpp

```c
float add(float, float);
float mul(float, float);
unsigned int fibonacci(unsigned int n);
```

```c
#include <math.h>
#include "demolib.hpp"
#include <stdio.h>
float add(float x, float y){
    return x+y;
}
float mul(float x, float y){
    return x*y;
}
unsigned int fibonacci(unsigned int n){
    if(n<2){
        return n;
    }
    else{
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

Fichier : demo.cpp

```c
#include <iostream>
#include "demolib.hpp"
int main(){
    int a=add(10,20);
    std::cout<<"10+20="<<a<<std::endl;
    int m=mul(10,20);
    std::cout<<"10x20="<<m<<std::endl;
    std::cout<<"Fibonacci(16)="<<fibonacci(16)<<std::endl;
}
```

Compilation :

```
$ emcc demolib.cpp demo.cpp -o demo.wasm -s WASM=1
```

Exécution :

```
$ wasmer run demo.wasm
Compiling to WebAssembly
10+20=30
10x20=200
Fibonacci(16)=987
```

1 Pour ces cas d'usage, les dossiers qui vont suivre, « Cas_usage_1 » à « Cas_usage_4 », ainsi que leurs fichiers sont disponibles au téléchargement, ils sont à créer de toutes pièces.

2 Voir page 14

Générer une bibliothèque, l'utiliser.

Tp : Cas_usage_3

Fichiers : arithm.hpp, arithm.cpp

arithm.hpp

```
1 int add(int, int);
2 int mul(int, int);
3 float divide(float,float);
```

arithm.cpp

```
1 #include <math.h>
2 #include "arithm.hpp"
3 #include <stdio.h>
4 int add(int x, int y){
5      return x+y;
6 }
7 int mul(int x, int y){
8     return x*y;
9 }
10 float divide(float x, float y){
11     if (y==0) {
12         //NAN: Not a number; défini dans math.h
13         return NAN;
14     } else {
15         return x/y;
16     }
17 }
```

Fichiers : fibonacci.hpp, fibonnacci.cpp

fibonacci.hpp

```
1 using namespace std;
2 unsigned int fibonacci(unsigned int);
3 string concat(string,string);
```

fibonacci.cpp

```
1 #include <string>
2 #include "fibonacci.hpp"
3 unsigned int fibonacci(unsigned int n){
4     if(n<2){
5         return n;
6     }
7     else{
8         return fibonacci(n-1)+fibonacci(n-2);
9     }
10 }
11 string concat(string a, string b){
12     return a+b;
13 }
```

Générer la bibliothèque

```
$ em++ arithm.cpp -c -o arithm.o -Oz -s SIDE_MODULE=1 -s WASM=1        Fichier produit : arithm.o
$ em++ fibonacci.cpp -c -o fibonacci.o -Oz -s SIDE_MODULE=1 -s WASM=1  Fichier produit : fibonacci.o
$ emar rcs demolib.a arithm.o fibonacci.o                              Fichier produit : demolib.a
```

Utiliser la bibliothèque.

Fichier : demoproj.cpp

```
1 #include <iostream>
2 #include "arithm.hpp"
3 #include "fibonacci.hpp"
4 int main(){
5     int a=add(10,20);
6     std::cout<<"10+20="<<a<<std::endl;
7     int m=mul(10,20);
8     std::cout<<"10x20="<<m<<std::endl;
9     float d=divide(30,10);
10    std::cout<<"30/10="<<d<<std::endl;
11    std::cout<<"Fibonacci(16):"<<fibonacci(16)<<std::endl;
12    string s=concat("Hello ", "World");
13    std::cout<<s<<std::endl;
14 }
```

Compilation :

```
emcc demolib.a demoproj.cpp -o demoproj.wasm -s WASM=1
```

Exécution : $wasmer run demoproj.wasm

```
:~/formations/WebAssembly/labs/web/Cas_usage_3$ wasmer run demoproj.wasm
Compiling to WebAssembly                                          10+20=30

10x20=200
30/10=3
Fibonacci(16):987
Hello World
```

Utilisation avec Node.js[3]

Tp : Cas_usage_4

Fichiers : demolib.h, demolib.c

demolib.h

```c
int fnadd(int, int);
int fnmul(int,int);
unsigned int fibonacci(unsigned int);
```

demolib.c

```c
int fnadd(int x, int y){
    return x+y;
}
int fnmul(int x,int y){
    return x*y;
}
unsigned int fibonacci(unsigned int n){
    int z;
    if(n<2){
        z=n;
    }
    else{
        z=fibonacci(n-1)+ fibonacci(n-2);
    }
    return z;
}
```

Compilation :

$ emcc demolib.c -Oz -s WASM=1 -s SIDE_MODULE=1 -o demolib.wasm

Fichier : demo.js

```javascript
const fs=require("fs");
async function demo() {
    let wasm=fs.readFileSync("demolib.wasm");
    let lib = await WebAssembly.instantiate(wasm);
    let arith = lib.instance.exports
    console.log(arith.fnadd(10,20));
    console.log(arith.fnmul(100,9));
    console.log(arith.divide(30,20));
    console.log(arith.divide(30,0));
    let fmax = 20
    for(var i=0; i <= fmax; i++) {
    console.log("fib(" + i + ")=" + arith.fibonacci(i));
    }
};
demo().then();
```

Exécution : $ node demo.js

```
:~/formations/WebAssembly/labs/web/Cas_usage_4$ node demo.js
30
900
fib(0)=0
fib(1)=1
fib(2)=1
fib(3)=2
fib(4)=3
fib(5)=5
fib(6)=8
fib(7)=13
fib(8)=21
fib(9)=34
fib(10)=55
fib(11)=89
fib(12)=144
fib(13)=233
fib(14)=377
fib(15)=610
fib(16)=987
fib(17)=1597
fib(18)=2584
fib(19)=4181
fib(20)=6765
```

3 La version de node.js utilisé ici est : v24.11.0, le comportement peurt varier d'une version à l'autre.

8 Outils d'exécution (Runtimes)

Wasmer

Runtime multi-backend pour WebAssembly compatible WASI.

bash

```
# Installation
curl https://get.wasmer.io -sSfL | sh

# Exécution
wasmer run hello.wasm
wasmer run hello.wasm -- arg1 arg2
```

Wasmtime

Runtime officiel du Bytecode Alliance, performant et sécurisé.

bash

```
# Installation
curl https://wasmtime.dev/install.sh -sSf | bash

# Exécution
wasmtime hello.wasm
wasmtime run hello.wasm --dir=. -- arg1 arg2
```

WasmEdge

Orienté performance et intégration Cloud/IA avec extensions réseau et TensorFlow.

bash

```
# Installation
curl -sSf https://raw.githubusercontent.com/WasmEdge/WasmEdge/master/utils/install.sh |
bash

# Exécution
wasmedge hello.wasm
```

9 WebAssembly et le noyau Linux

WasmLinux

Projet permettant d'exécuter un noyau Linux complet dans le navigateur via WebAssembly.

1 Caractéristiques :

- Basé sur LKL (Linux Kernel Library)
- Exécutable entièrement en WASM
- Interface via BusyBox (commandes : ls, vi, dmesg, ifconfig, ping, top)
- Démo en ligne disponible à : https://wasmlinux-demo.pages.dev/

2 Limitations actuelles :

- Pas d'envoi asynchrone de signaux
- Noyau NOMMU (pas de mmap ni fork)
- Pas de véritable mode noyau

3 LKL (Linux Kernel Library)

Permet la réutilisation maximale du code du noyau Linux avec un minimum d'efforts. Applications :

- Création d'applications en espace utilisateur lisant les systèmes de fichiers Linux
- Pilotes noyau pour d'autres OS
- Support dans les chargeurs de démarrage

10 Glossaire technique

- **Binaryen** : Compilateur de bytecode en WASM binaire exécutable
- **S-Expression** : Format textuel organisé comme Lisp
- **MVP** (Minimal Viable Product) : Version minimale viable de WASM
- **WABT** : WebAssembly Binary Toolkit (suite d'outils)
- **Wast** : WebAssembly Text Format (format textuel)
- **WAVM** : WebAssembly Virtual Machine
- **WASI** : WebAssembly System Interface (accès contrôlé aux ressources système)

11 L'avenir de WebAssembly

Perspectives

- Exécution d'applications côté serveur (comme Node.js pour JavaScript)
- Développement de jeux mobiles haute performance
- Génération de contenu multimédia HD
- Création de simulateurs scientifiques
- Solutions de visioconférence
- Big Data sans traitement côté serveur

Citation de Brendan Eich (créateur de JavaScript)

"Plutôt que de vouloir tuer JavaScript, ce qui n'est pas faisable, ce que nous essayons de faire, c'est de répondre à de vrais problèmes d'ingénierie. Avec un arbre syntaxique abstrait compressé, 20 fois plus rapide, il ne faudrait que quelques secondes au lieu de 20 à 30. Il y a donc une vraie raison pour WASM, et c'est une raison valable."

12 Conclusion

WebAssembly représente une véritable révolution pour le développement web et au-delà. En offrant des performances proches du natif, une sécurité renforcée et une portabilité maximale, WASM ouvre de nouvelles possibilités pour :

- Le portage d'applications natives vers le web
- L'exécution de code haute performance dans le navigateur
- Le développement d'applications embarquées et IoT
- L'intégration dans des environnements cloud et edge

Avec le support universel des navigateurs modernes, une communauté active et des outils matures comme Emscripten, WebAssembly est devenu un standard incontournable du développement moderne.