CPPP 2021 PARIS

Sandbox Games: Using WebAssembly and C++ to make a simple game

Ólafur Waage

MUREX

CODE RECKONS

Science to the CORE

Ólafur Waage

Senior Software Developer - TurtleSec AS

@olafurw on Twitter

TurtleSec

> *I must go forward where I have never been instead of backwards where I have.*

I must go forward where I have never been instead of backwards where I have.

- *Winnie the Pooh*

WHAT THIS TALK IS AND ISN'T

WHAT THIS TALK IS AND ISN’T

This is not a game development or game design talk. A while ago I was making a game using WebAssembly and these are the walls I encountered along the way

WHAT THIS TALK IS AND ISN’T

This is not a game development or game design talk. A while ago I was making a game using WebAssembly and these are the walls I encountered along the way

This is not a comprehensive talk about WebAssembly.

WHAT THIS TALK IS AND ISN’T

This is not a game development or game design talk. A while ago I was making a game using WebAssembly and these are the walls I encountered along the way

This is not a comprehensive talk about WebAssembly.

The idea here is to be pragmatic and learn what this tool has to offer and what problems it can solve.

What is WebAssembly?

What is WebAssembly?

How can something be neither Web nor Assembly?

WHAT IS WEBASSEMBLY?

WebAssembly is a binary format* originally designed to allow for performant execution of code within browsers.

WHAT IS WEBASSEMBLY?

WebAssembly is a binary format* originally designed to allow for performant execution of code within browsers.

- Announced 2015

WHAT IS WEBASSEMBLY?

WebAssembly is a binary format* originally designed to allow for performant execution of code within browsers.

- Announced 2015
- Working Drafts in 2018

WHAT IS WEBASSEMBLY?

WebAssembly is a binary format* originally designed to allow for performant execution of code within browsers.

- Announced 2015
- Working Drafts in 2018
- W3C recommendation in 2019

WHAT IS WEBASSEMBLY?

WebAssembly is a binary format* originally designed to allow for performant execution of code within browsers.

- Announced 2015
- Working Drafts in 2018
- W3C recommendation in 2019

WebAssembly can be thought of as the target output of any language and in recent times can be executed outside of the web.

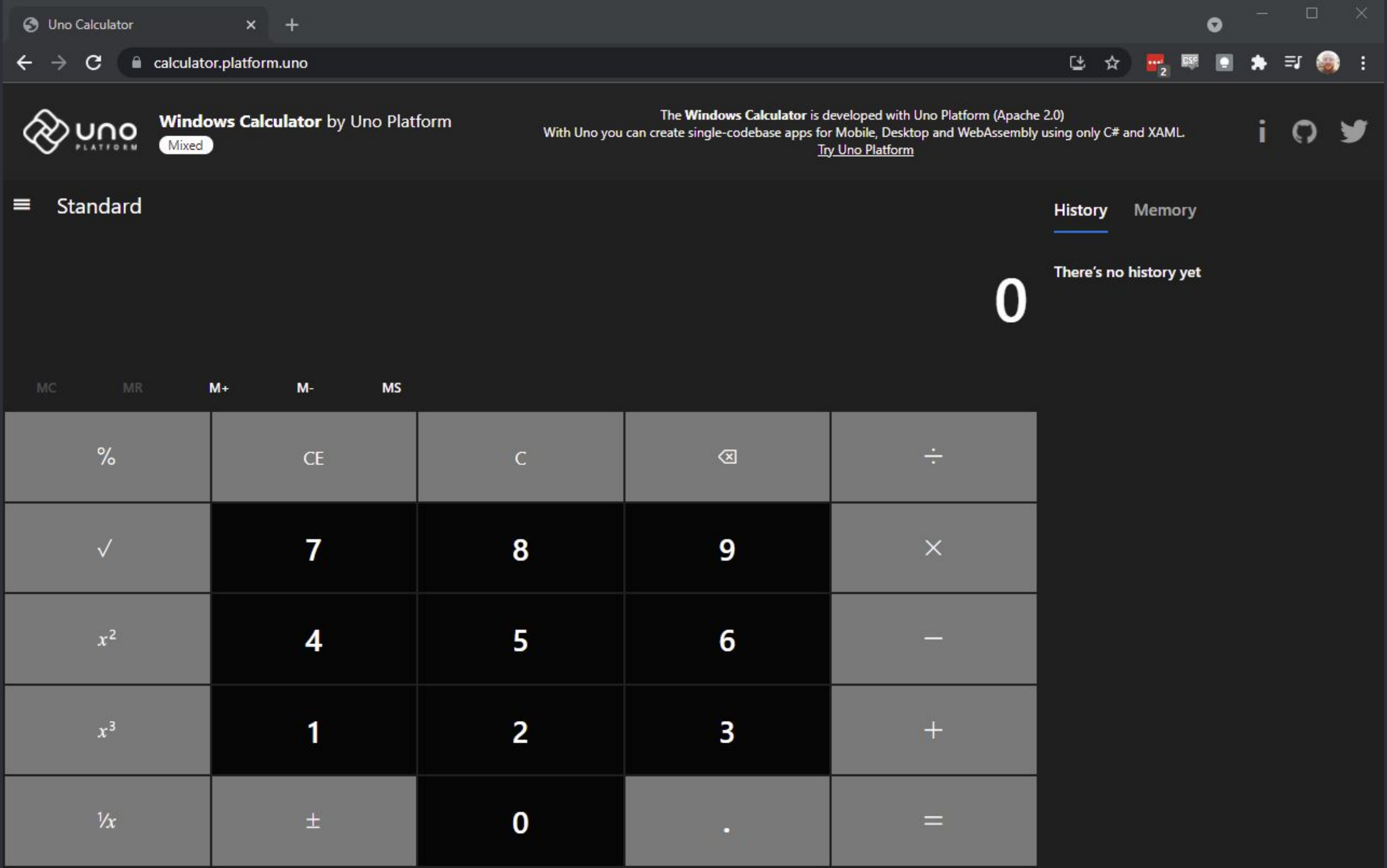
WEBASSEMBLY EXAMPLES?

Many of you might associate WebAssembly with games only, and even though this talk is also doing that, WebAssembly has so much more to offer.

WEBASSEMBLY EXAMPLES?

Many of you might associate WebAssembly with games only, and even though this talk is also doing that, WebAssembly has so much more to offer.

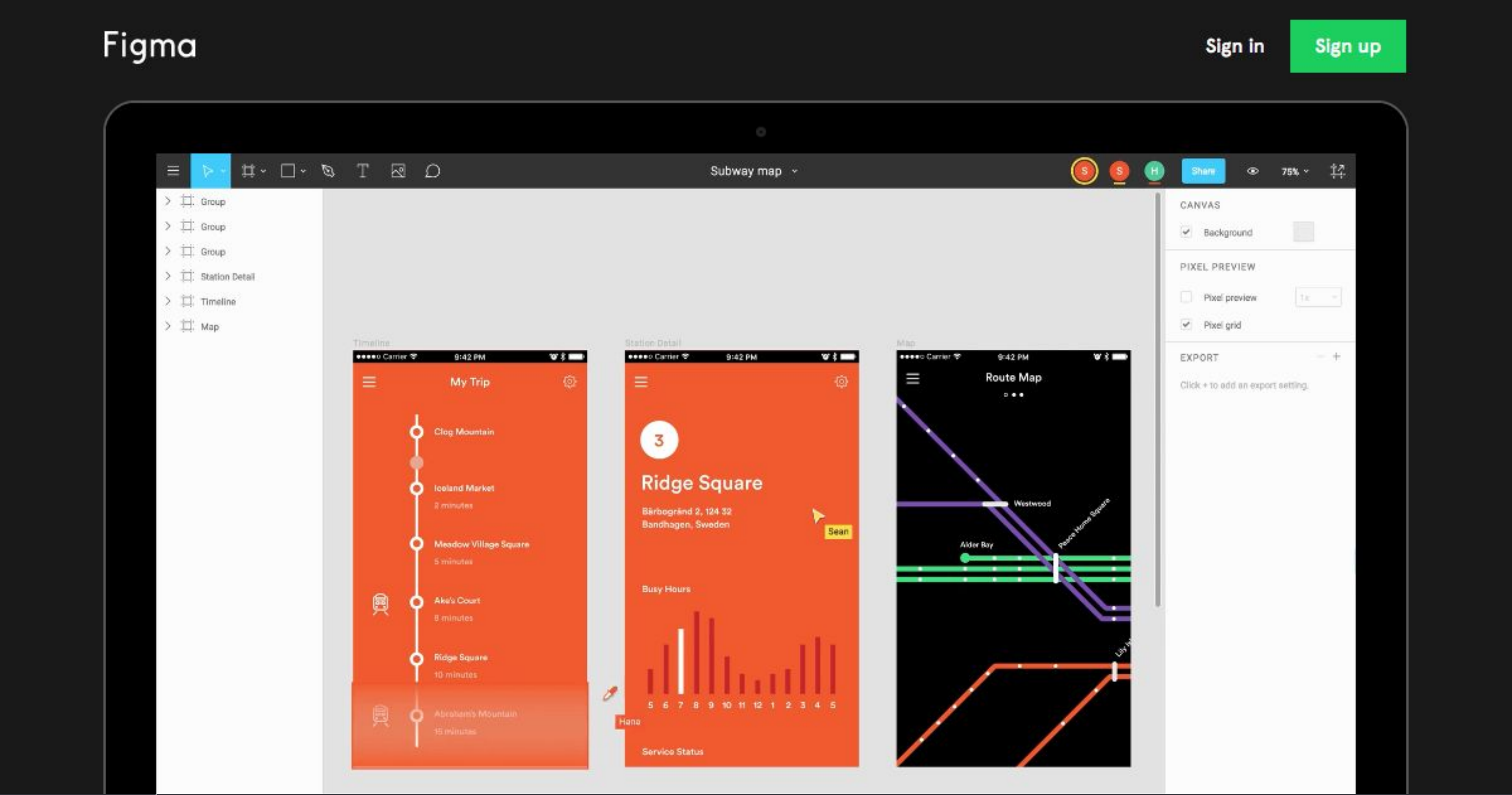
Here are some examples of things you might not have thought are written with WebAssembly.



Windows Calculator by Uno Platform

The **Windows Calculator** is developed with Uno Platform (Apache 2.0)
With Uno you can create single-codebase apps for Mobile, Desktop and WebAssembly using only C# and XAML.
Try Uno Platform

There's no history yet

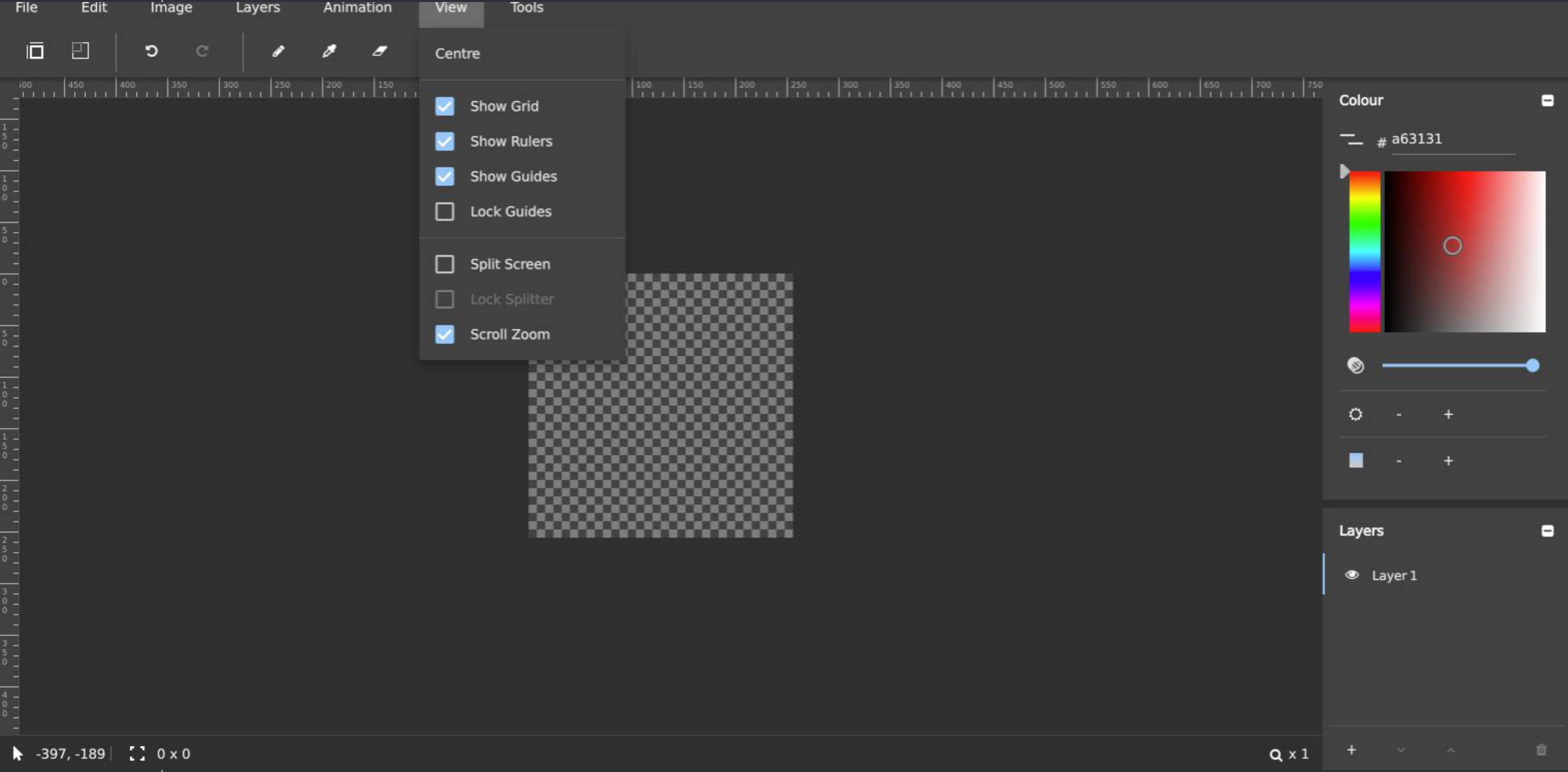


☑ Background

☐ Pixel preview

☑ Pixel grid

Click + to add an export setting.



☑ Show Grid

☑ Show Rulers

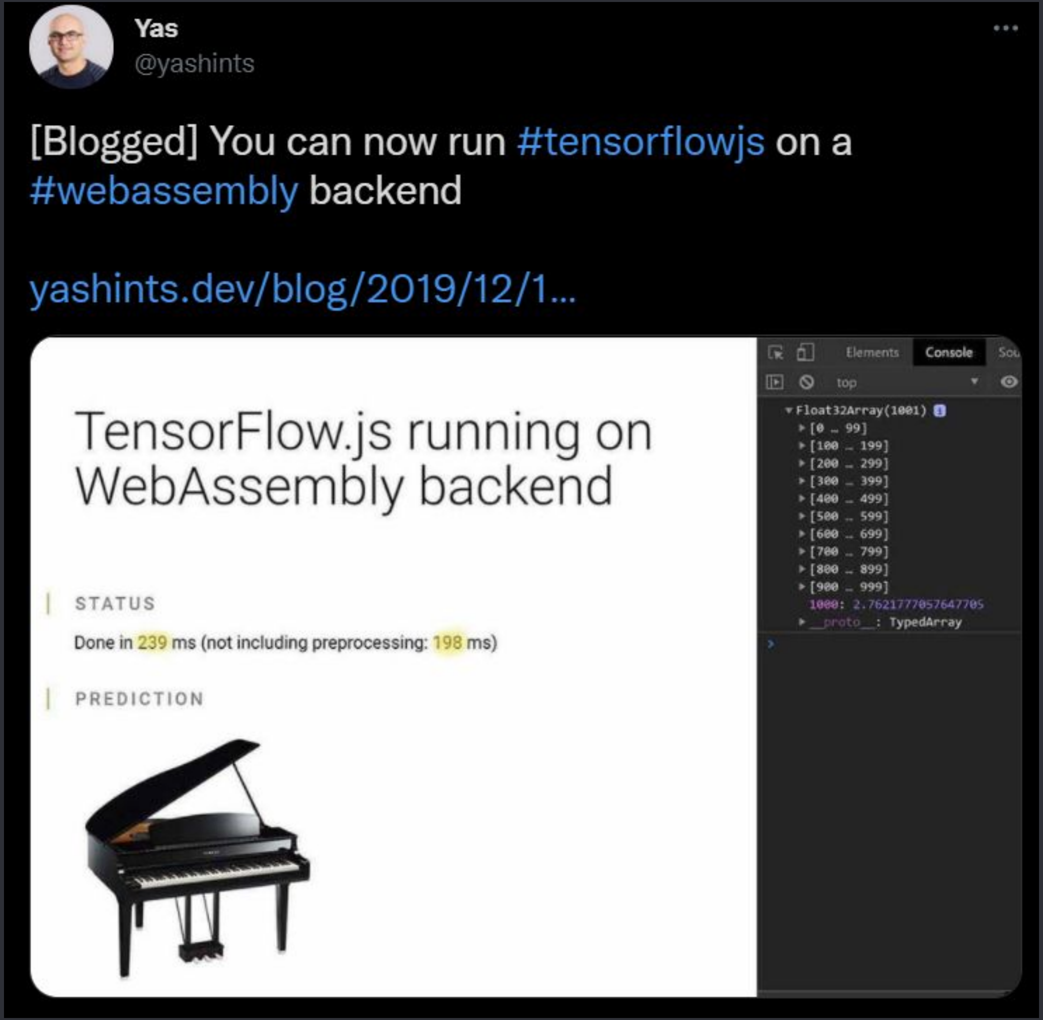
☑ Show Guides

□ Lock Guides

□ Split Screen

□ Lock Splitter

☑ Scroll Zoom



Yas
@yashints

[Blogged] You can now run #tensorflowjs on a #webassembly backend

yashints.dev/blog/2019/12/1...

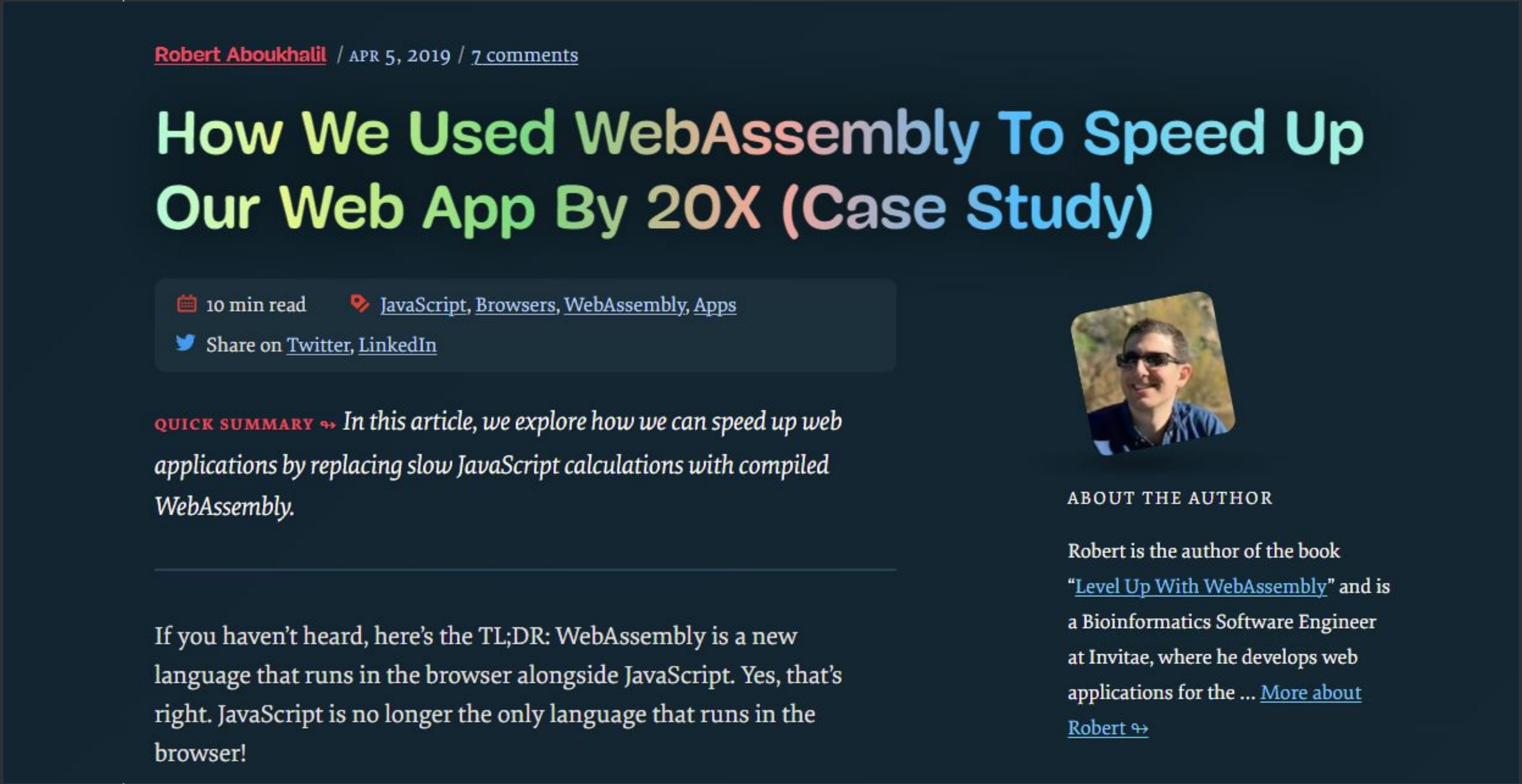
TensorFlow.js running on WebAssembly backend

STATUS

Done in 239 ms (not including preprocessing: 198 ms)

PREDICTION

```
▾ Float32Array(1001)
  ▸ [0 … 99]
  ▸ [100 … 199]
  ▸ [200 … 299]
  ▸ [300 … 399]
  ▸ [400 … 499]
  ▸ [500 … 599]
  ▸ [600 … 699]
  ▸ [700 … 799]
  ▸ [800 … 899]
  ▸ [900 … 999]
    1000: 2.7621777057647705
  ▸ __proto__: TypedArray
>
```



Robert Aboukhalil / APR 5, 2019 / 7 comments

How We Used WebAssembly To Speed Up Our Web App By 20X (Case Study)

10 min read JavaScript, Browsers, WebAssembly, Apps

Share on Twitter, LinkedIn

QUICK SUMMARY ↬ *In this article, we explore how we can speed up web applications by replacing slow JavaScript calculations with compiled WebAssembly.*

If you haven't heard, here's the TL;DR: WebAssembly is a new language that runs in the browser alongside JavaScript. Yes, that's right. JavaScript is no longer the only language that runs in the browser!

ABOUT THE AUTHOR

Robert is the author of the book "Level Up With WebAssembly" and is a Bioinformatics Software Engineer at Invitae, where he develops web applications for the ... More about Robert ↬

What is Emscripten?

What is Emscripten?

WebAssembly before WebAssembly

WHAT IS EMSCRIPTEN?

We originally had asm.js from Mozilla which had similar goals to WebAssembly, to run efficient code on the web.

WHAT IS EMSCRIPTEN?

We originally had asm.js from Mozilla which had similar goals to WebAssembly, to run efficient code on the web.

asm.js is a subset of JavaScript and your lower level code would then be transpiled into it.

WHAT IS EMSCRIPTEN?

We originally had asm.js from Mozilla which had similar goals to WebAssembly, to run efficient code on the web.

asm.js is a subset of JavaScript and your lower level code would then be transpiled into it.

This is where Emscripten came into play.

WHAT IS EMSCRIPTEN?

Emscripten is based on the LLVM/Clang toolchains which allows you target WebAssembly as the binary output.

WHAT IS EMSCRIPTEN?

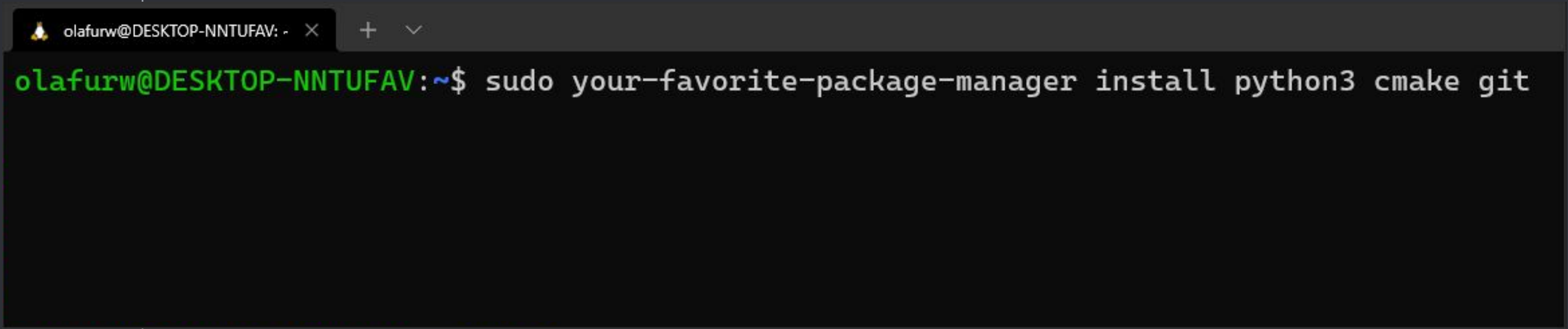
Emscripten is based on the LLVM/Clang toolchains which allows you target WebAssembly as the binary output.

This allows you to get many different types of outputs, not only WASM files but .js and .html

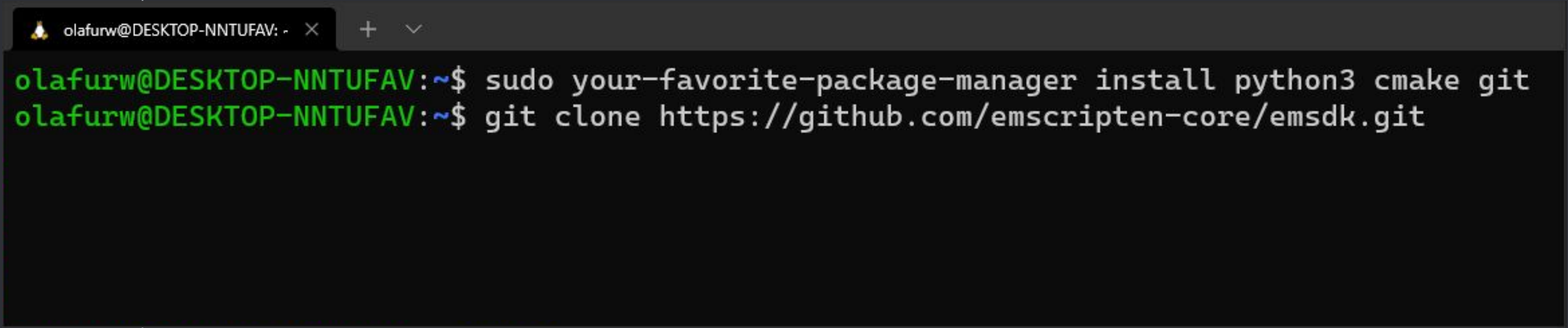
INSTALLING EMSCRIPTEN

Let’s go over the installation process and setup a simple development environment.

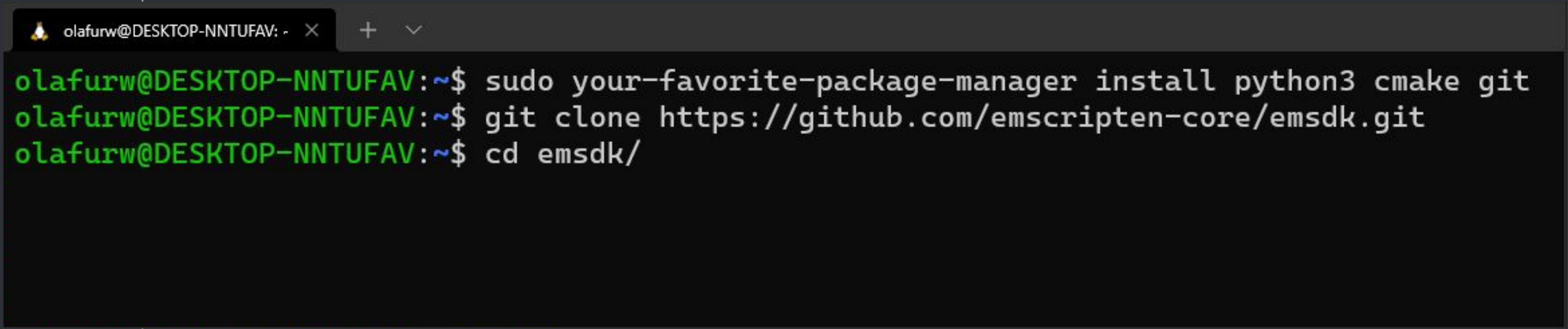
- Text editor is VSCode
- WSL2 running Ubuntu 20.04
- https://github.com/olafurw/talk-cppp-webassembly



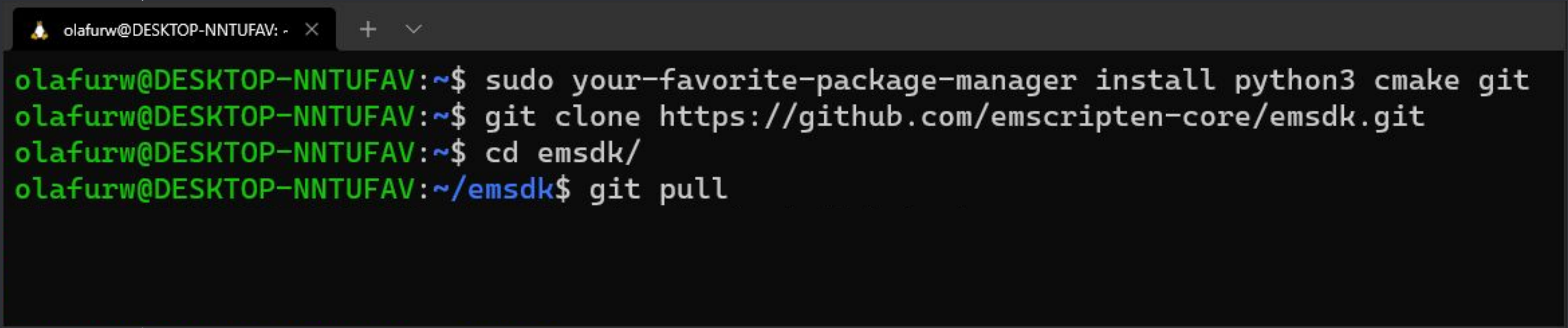
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
```



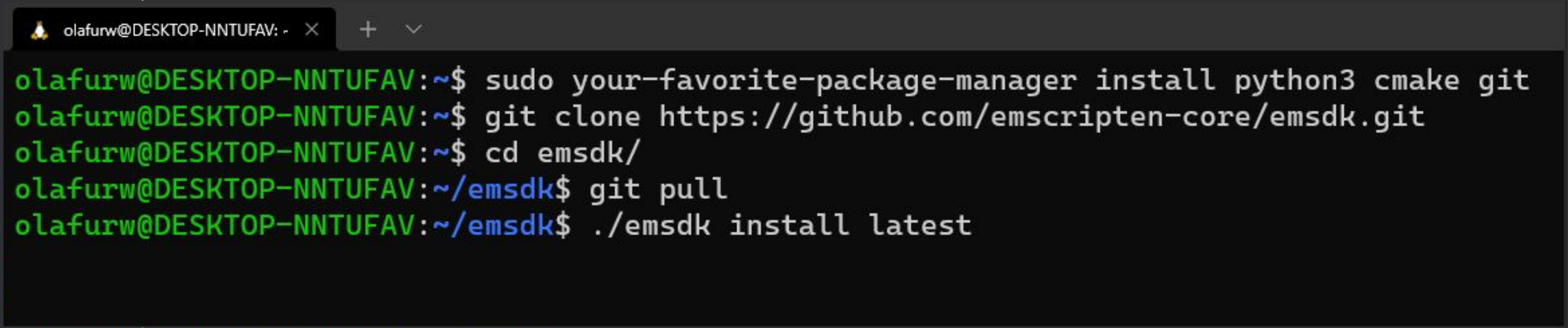
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
```



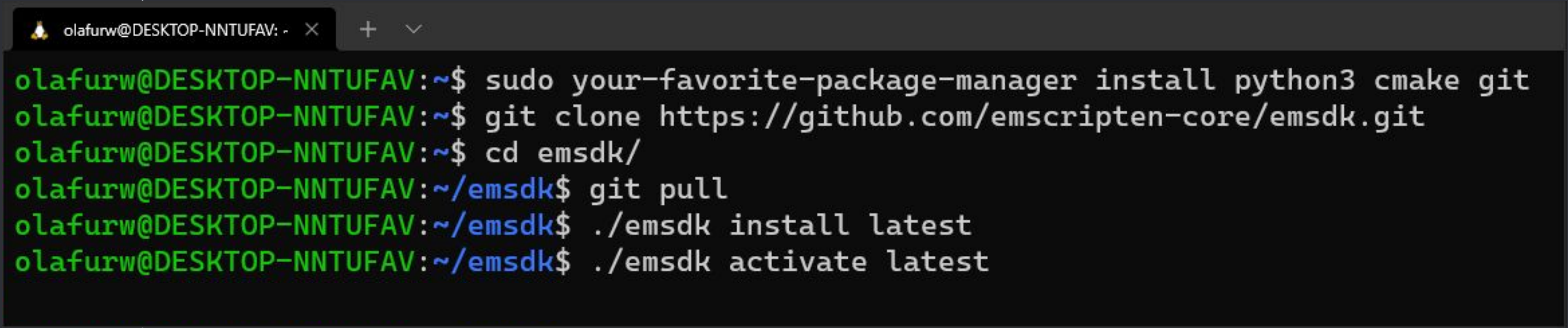
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
olafurw@DESKTOP-NNTUFAV:~$ cd emsdk/
```



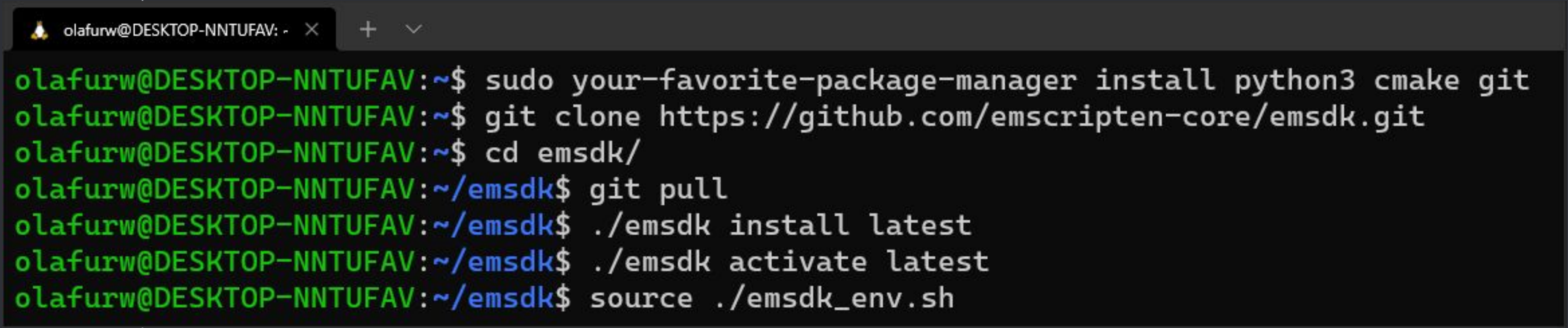
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
olafurw@DESKTOP-NNTUFAV:~$ cd emsdk/
olafurw@DESKTOP-NNTUFAV:~/emsdk$ git pull
```



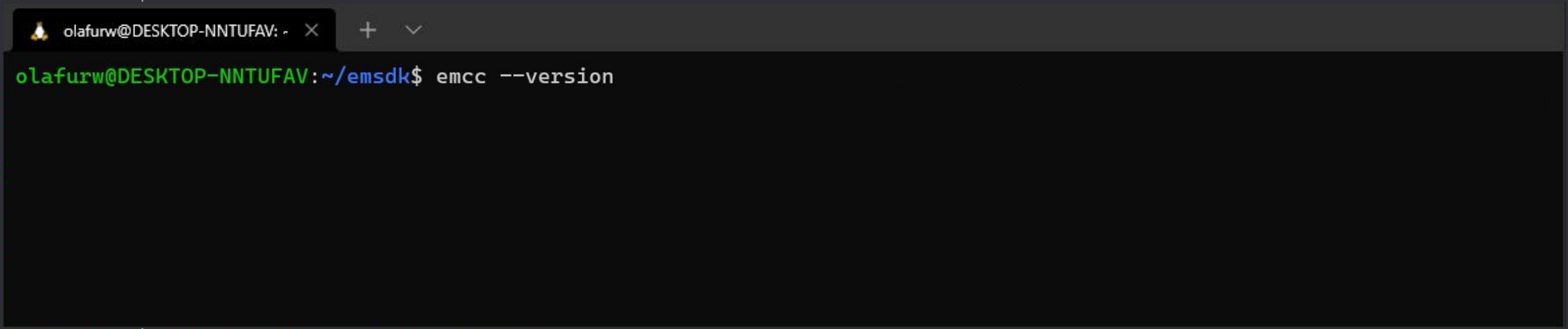
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
olafurw@DESKTOP-NNTUFAV:~$ cd emsdk/
olafurw@DESKTOP-NNTUFAV:~/emsdk$ git pull
olafurw@DESKTOP-NNTUFAV:~/emsdk$ ./emsdk install latest
```



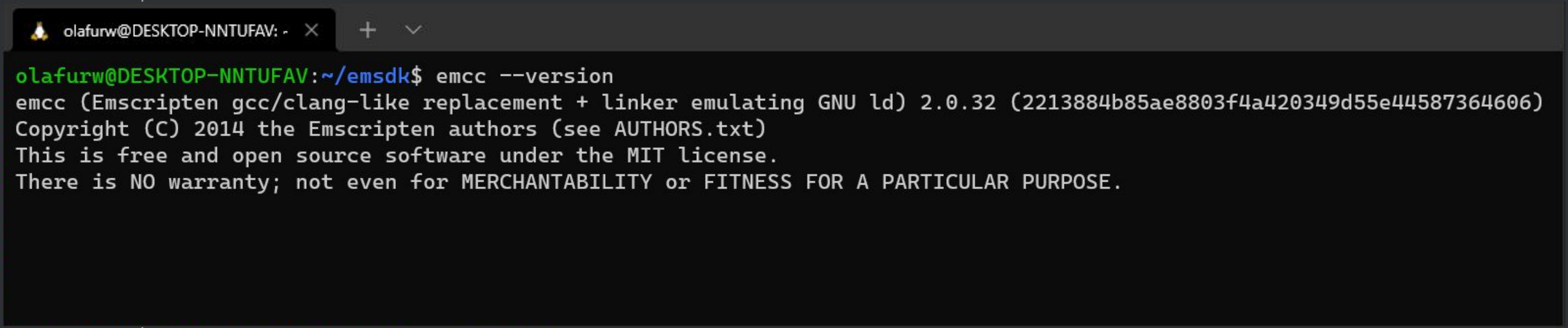
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
olafurw@DESKTOP-NNTUFAV:~$ cd emsdk/
olafurw@DESKTOP-NNTUFAV:~/emsdk$ git pull
olafurw@DESKTOP-NNTUFAV:~/emsdk$ ./emsdk install latest
olafurw@DESKTOP-NNTUFAV:~/emsdk$ ./emsdk activate latest
```



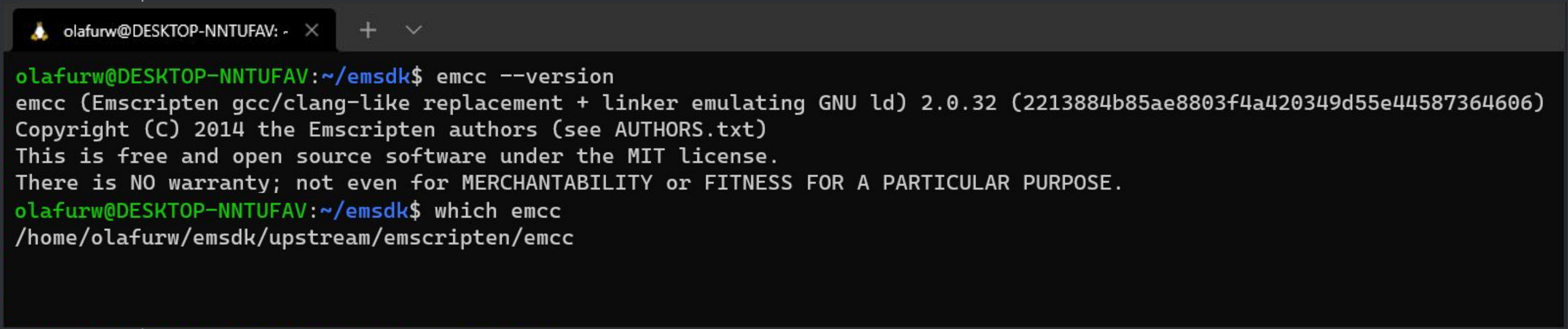
```
olafurw@DESKTOP-NNTUFAV:~$ sudo your-favorite-package-manager install python3 cmake git
olafurw@DESKTOP-NNTUFAV:~$ git clone https://github.com/emscripten-core/emsdk.git
olafurw@DESKTOP-NNTUFAV:~$ cd emsdk/
olafurw@DESKTOP-NNTUFAV:~/emsdk$ git pull
olafurw@DESKTOP-NNTUFAV:~/emsdk$ ./emsdk install latest
olafurw@DESKTOP-NNTUFAV:~/emsdk$ ./emsdk activate latest
olafurw@DESKTOP-NNTUFAV:~/emsdk$ source ./emsdk_env.sh
```



```
olafurw@DESKTOP-NNTUFAV:~/emsdk$ emcc --version
```



```
olafurw@DESKTOP-NNTUFAV:~/emsdk$ emcc --version
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 2.0.32 (2213884b85ae8803f4a420349d55e44587364606)
Copyright (C) 2014 the Emscripten authors (see AUTHORS.txt)
This is free and open source software under the MIT license.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```



```
olafurw@DESKTOP-NNTUFAV:~/emsdk$ emcc --version
emcc (Emscripten gcc/clang-like replacement + linker emulating GNU ld) 2.0.32 (2213884b85ae8803f4a420349d55e44587364606)
Copyright (C) 2014 the Emscripten authors (see AUTHORS.txt)
This is free and open source software under the MIT license.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
olafurw@DESKTOP-NNTUFAV:~/emsdk$ which emcc
/home/olafurw/emsdk/upstream/emscripten/emcc
```

HEY, WORLD, WHAT IS UP?

Now we have the Emscripten compiler installed in our system.

HEY, WORLD, WHAT IS UP?

Now we have the Emscripten compiler installed in our system.

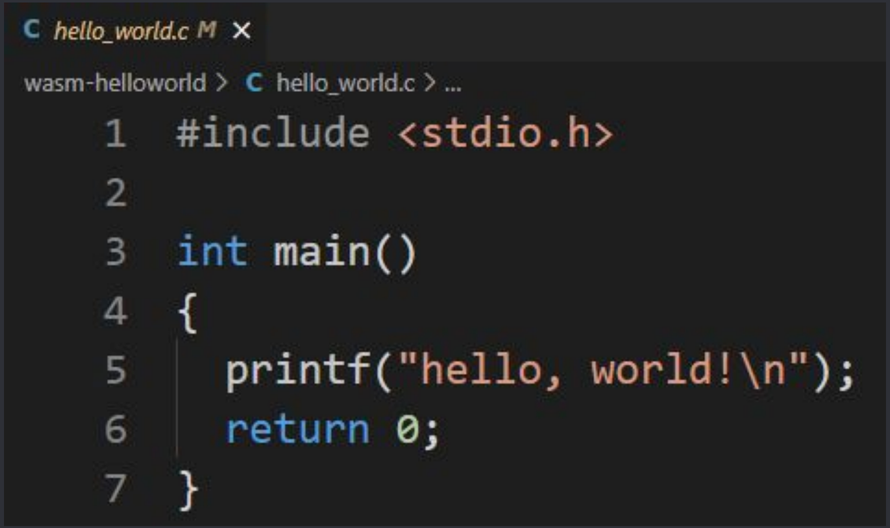
Time for the time honored tradition of the hello world example.

HEY, WORLD, WHAT IS UP?

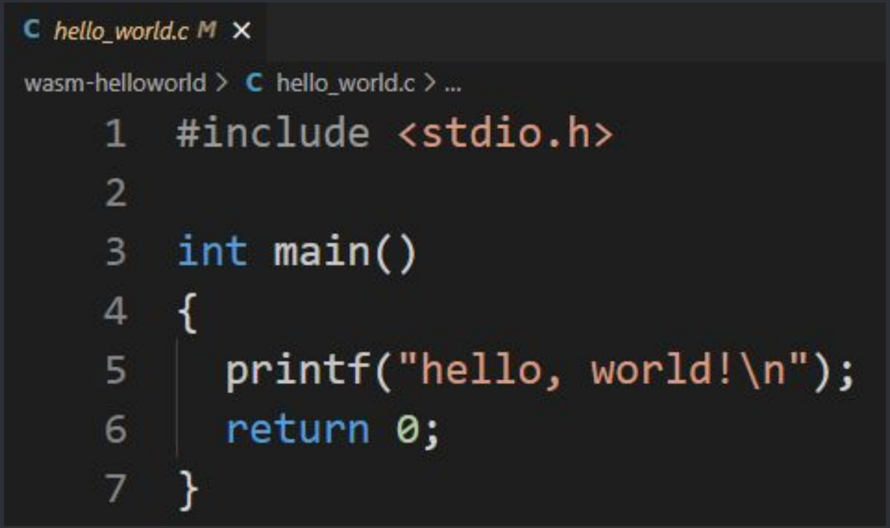
Now we have the Emscripten compiler installed in our system.

Time for the time honored tradition of the hello world example.

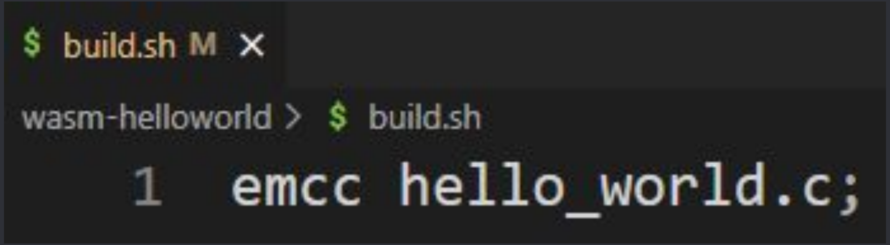
But there are a few more steps in this one than you'd normally expect.



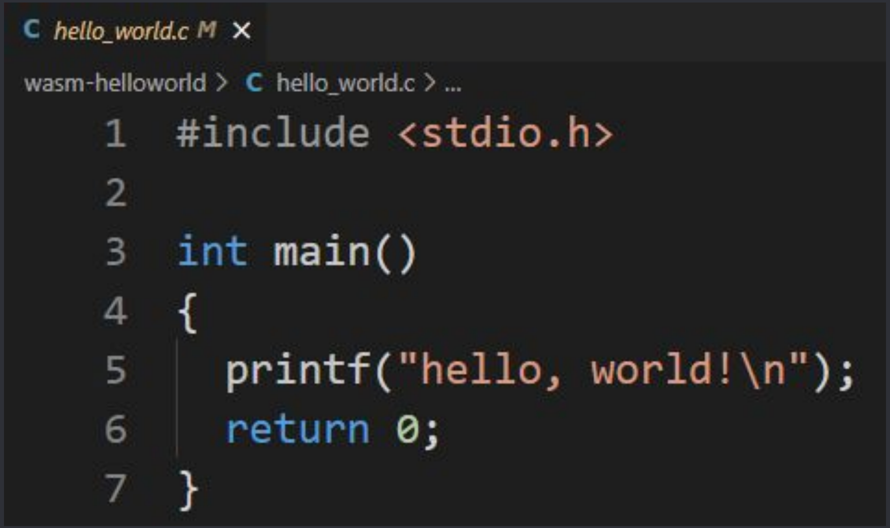
```c
#include <stdio.h>

int main()
{
  printf("hello, world!\n");
  return 0;
}
```



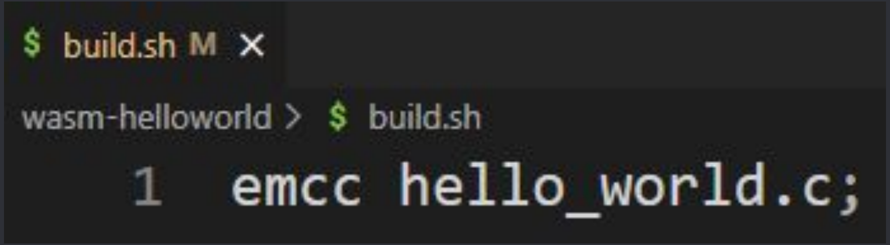
```c
#include <stdio.h>

int main()
{
    printf("hello, world!\n");
    return 0;
}
```



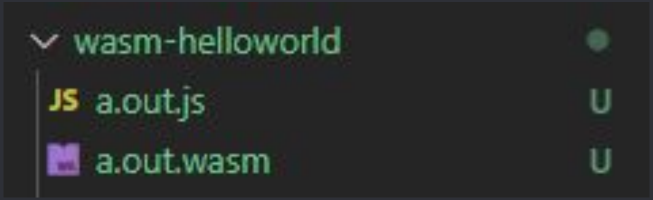
```sh
emcc hello_world.c;
```

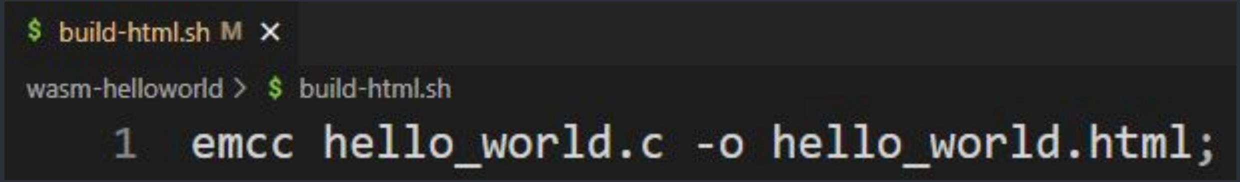


```c
#include <stdio.h>

int main()
{
  printf("hello, world!\n");
  return 0;
}
```

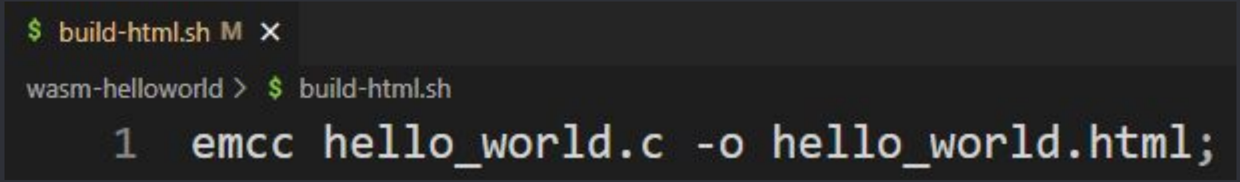


```sh
emcc hello_world.c;
```

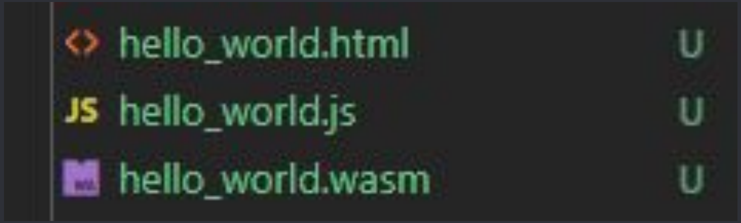




```
emcc hello_world.c -o hello_world.html;
```



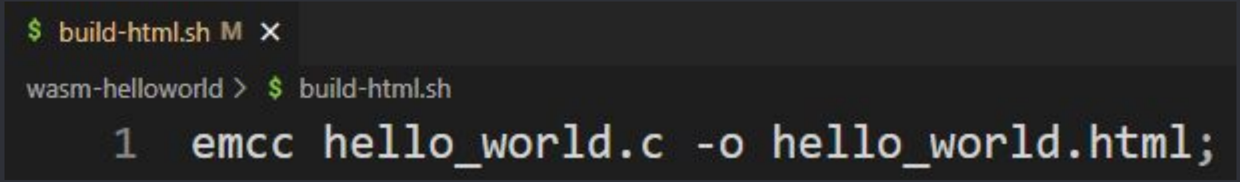
```
emcc hello_world.c -o hello_world.html;
```



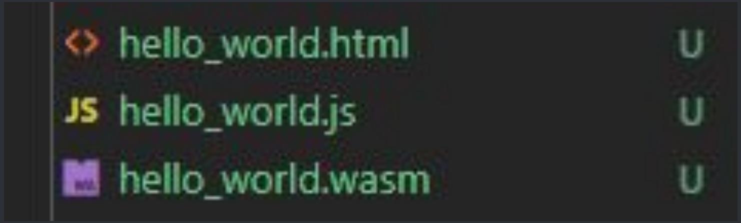
hello_world.html U

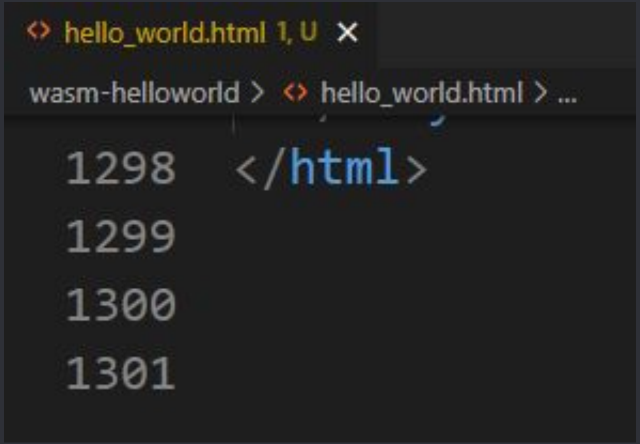
hello_world.js U

hello_world.wasm U

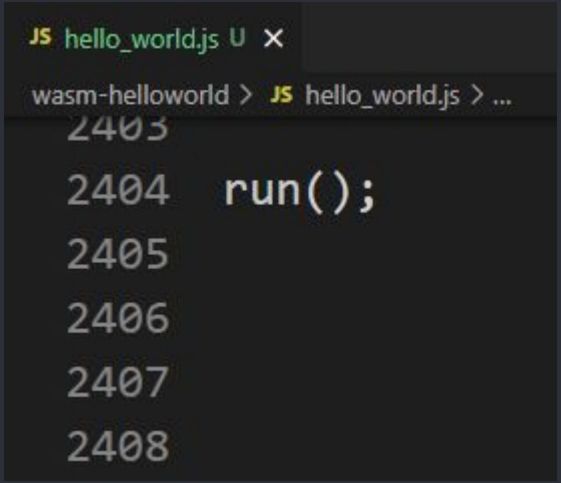


```
emcc hello_world.c -o hello_world.html;
```





```
</html>
```



```
run();
```

JUST RUN IT ALREADY!

Yes, with nodejs we can run the .js files just fine.

JUST RUN IT ALREADY!

Yes, with nodejs we can run the .js files just fine.

But let’s start by opening the HTML file directly.
Should be no problem, right?

1

WALL NUMBER 1

Of CORS there’s a problem here

YOUR SAFETY IS PARAMOUNT

Browsers don’t like opening random files from whatever location you decide.

YOUR SAFETY IS PARAMOUNT

Browsers don’t like opening random files from whatever location you decide.

There’s a thing called “Cross-origin resource sharing (CORS)”. By default browsers don’t like loading external files from disk using file://

YOUR SAFETY IS PARAMOUNT

Browsers don’t like opening random files from whatever location you decide.

There’s a thing called “Cross-origin resource sharing (CORS)”. By default browsers don’t like loading external files from disk using file://

The browser will load the html file fine but any external dependency will probably be blocked.

RUN EM RUN!

Best way to solve this is to run a webserver that is going to host the files.

RUN EM RUN!

Best way to solve this is to run a webserver that is going to host the files.

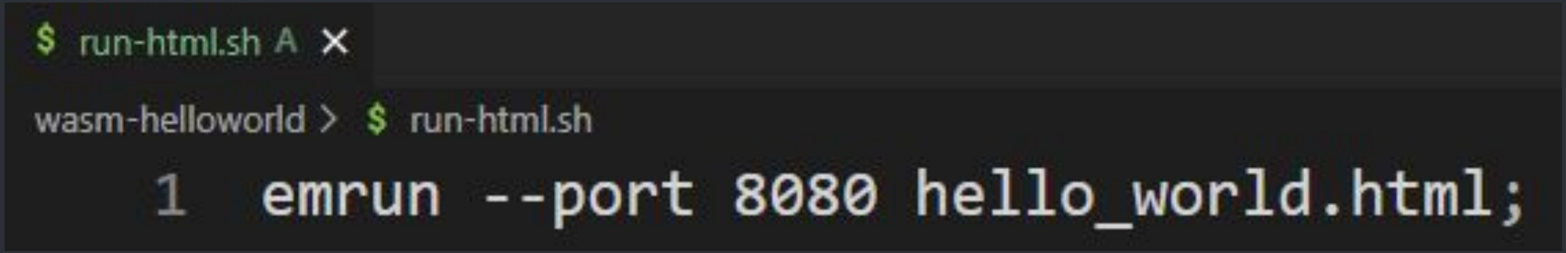
What I use while developing is emrun, a tool that comes with emscripten.

RUN EM RUN!

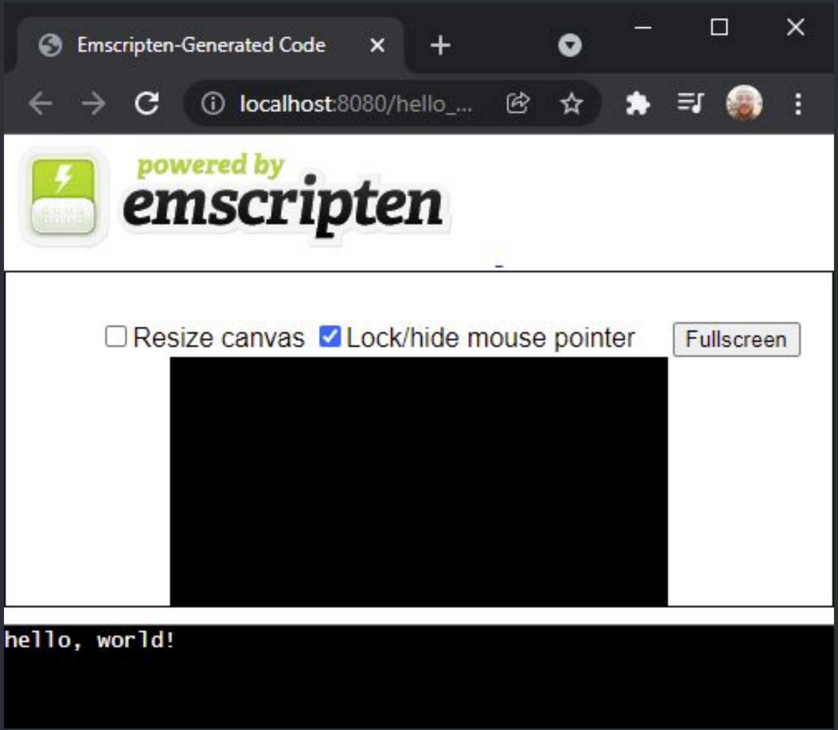
Best way to solve this is to run a webserver that is going to host the files.

What I use while developing is emrun, a tool that comes with emscripten.

emrun is a simple webserver but for our development purposes it is good enough.



```
emrun --port 8080 hello_world.html;
```



□ Resize canvas ☑ Lock/hide mouse pointer Fullscreen

```
hello, world!
```

VIDEO GAMES!

Now let’s look at the game we will be “making”.

We are going to make a simple sliding puzzle game, similar to games like “Threes” and “2048”

ENOUGH FUN

Now let’s covert this game over to WebAssembly.

ENOUGH FUN

Now let’s covert this game over to WebAssembly.

There are two ways to do this.

ENOUGH FUN

Now let’s covert this game over to WebAssembly.

There are two ways to do this.

- Keep the drawing in JS and game logic in C++

ENOUGH FUN

Now let’s covert this game over to WebAssembly.

There are two ways to do this.

- Keep the drawing in JS and game logic in C++
- Do everything in C++

ENOUGH FUN

Now let’s covert this game over to WebAssembly.

There are two ways to do this.

- Keep the drawing in JS and game logic in C++
- Do everything in C++

We will look at both, and the walls we hit along the way.

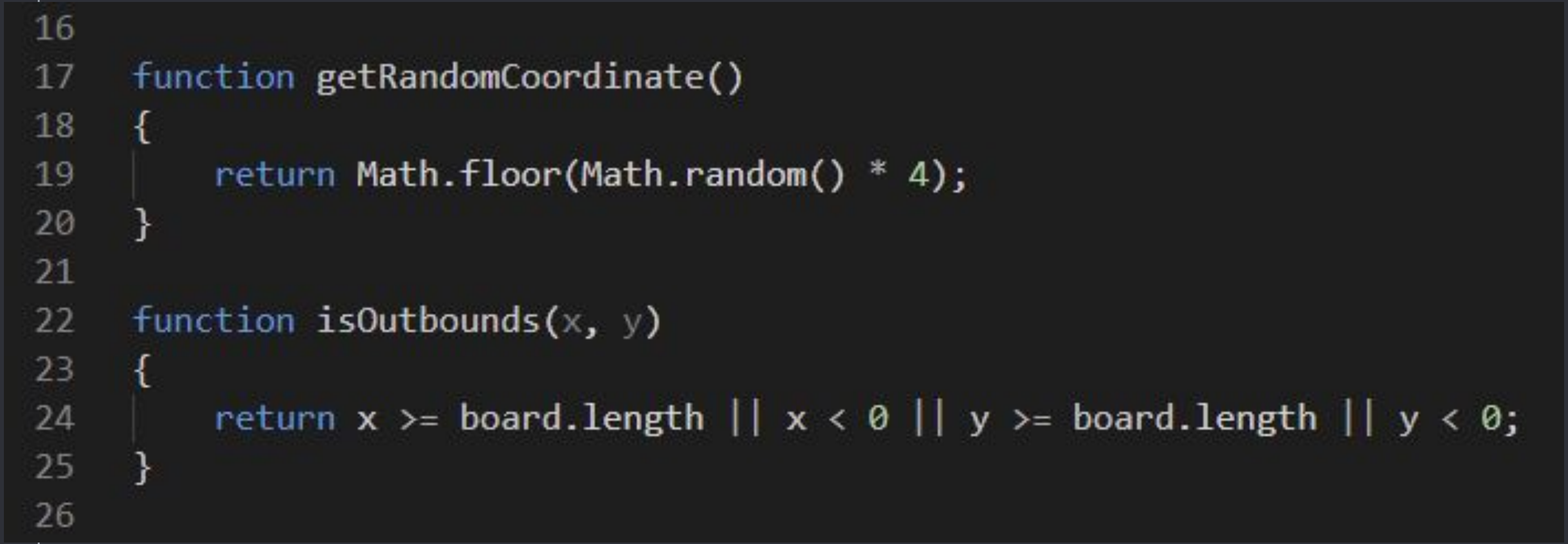
LET’S START CONVERTING

So let’s take some of the functions we have in the JS version and convert them over to C++

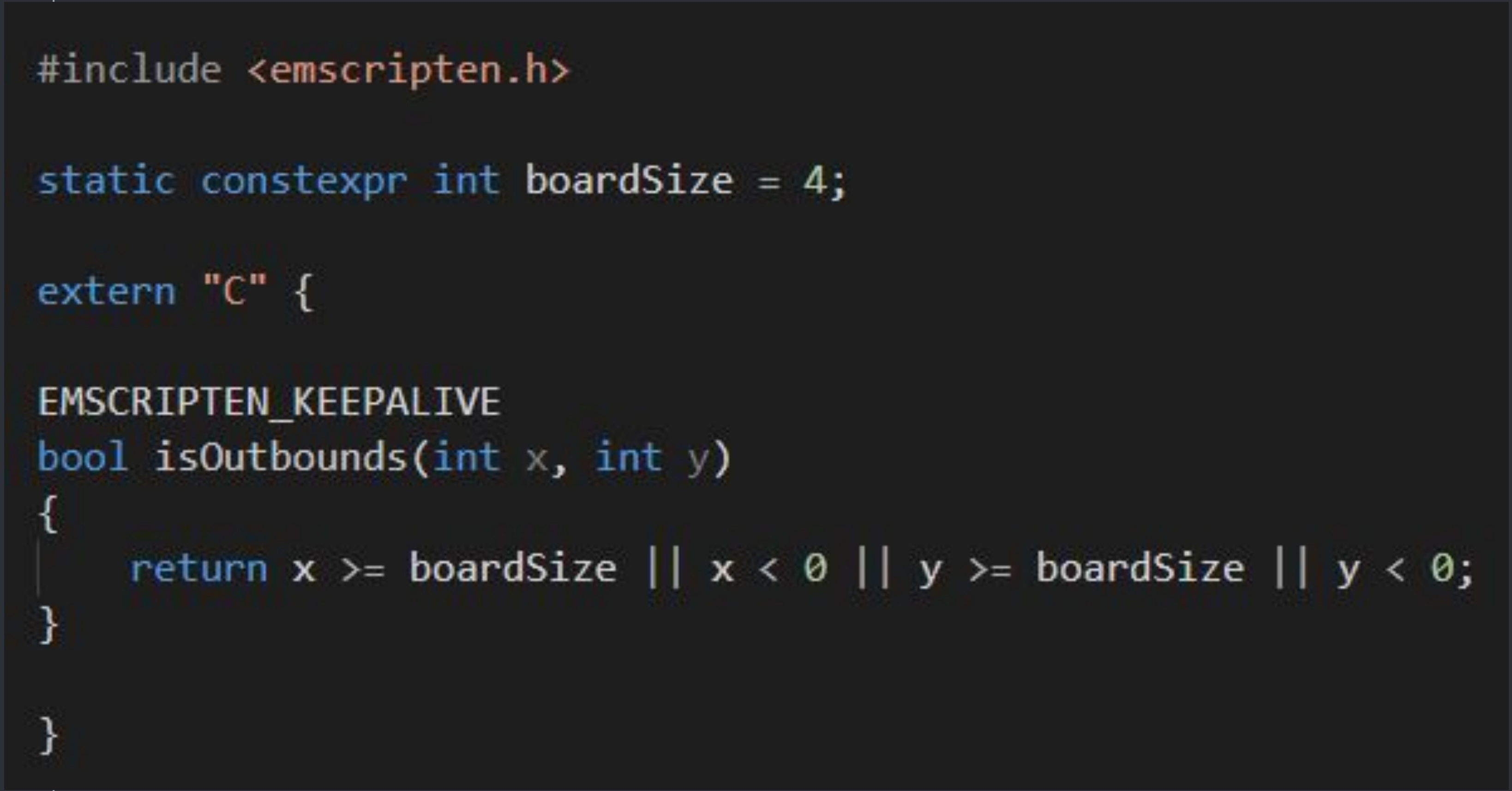
LET’S START CONVERTING

So let’s take some of the functions we have in the JS version and convert them over to C++

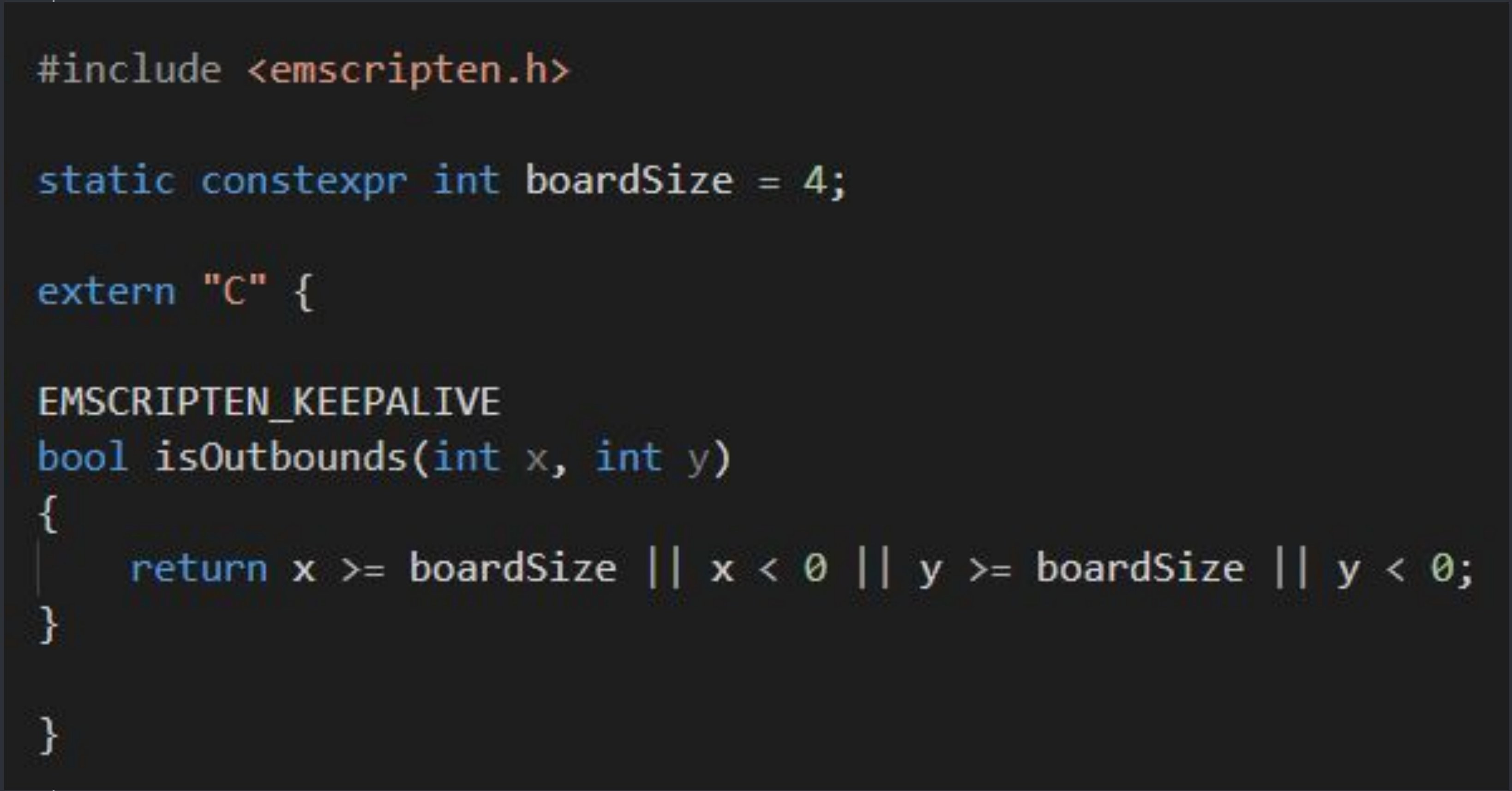
Some of them don’t even need to know about game state, so let’s start with them.



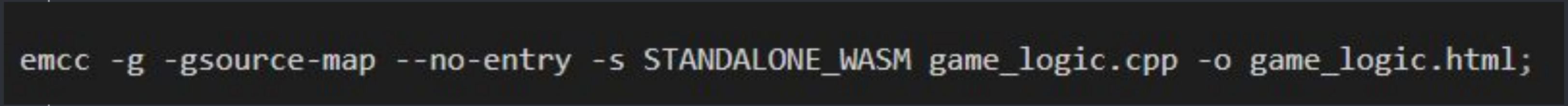
```

function getRandomCoordinate()
{
    return Math.floor(Math.random() * 4);
}

function isOutbounds(x, y)
{
    return x >= board.length || x < 0 || y >= board.length || y < 0;
}

```



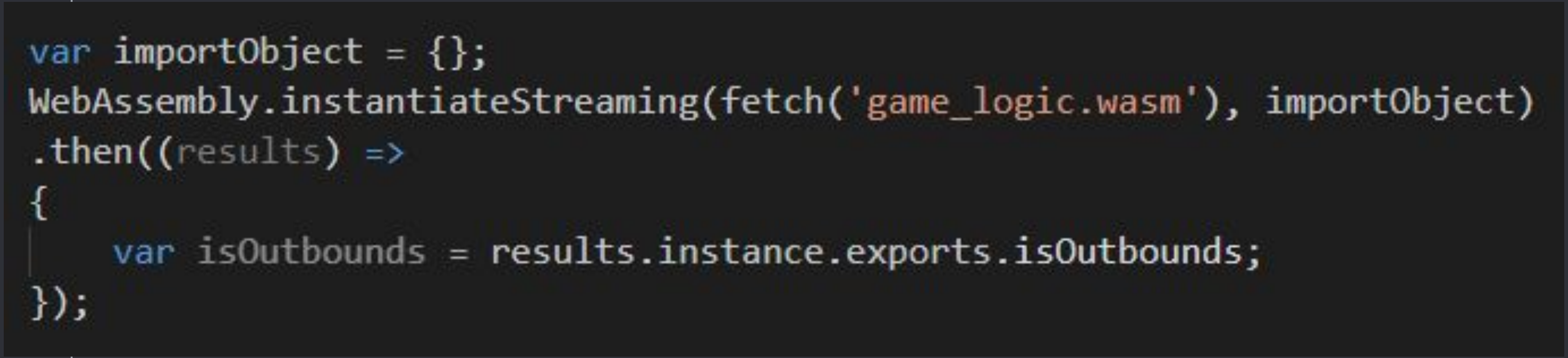
```
#include <emscripten.h>

static constexpr int boardSize = 4;

extern "C" {

EMSCRIPTEN_KEEPALIVE
bool isOutbounds(int x, int y)
{
    return x >= boardSize || x < 0 || y >= boardSize || y < 0;
}

}
```



```cpp
#include <emscripten.h>

static constexpr int boardSize = 4;

extern "C" {

EMSCRIPTEN_KEEPALIVE
bool isOutbounds(int x, int y)
{
    return x >= boardSize || x < 0 || y >= boardSize || y < 0;
}

}
```

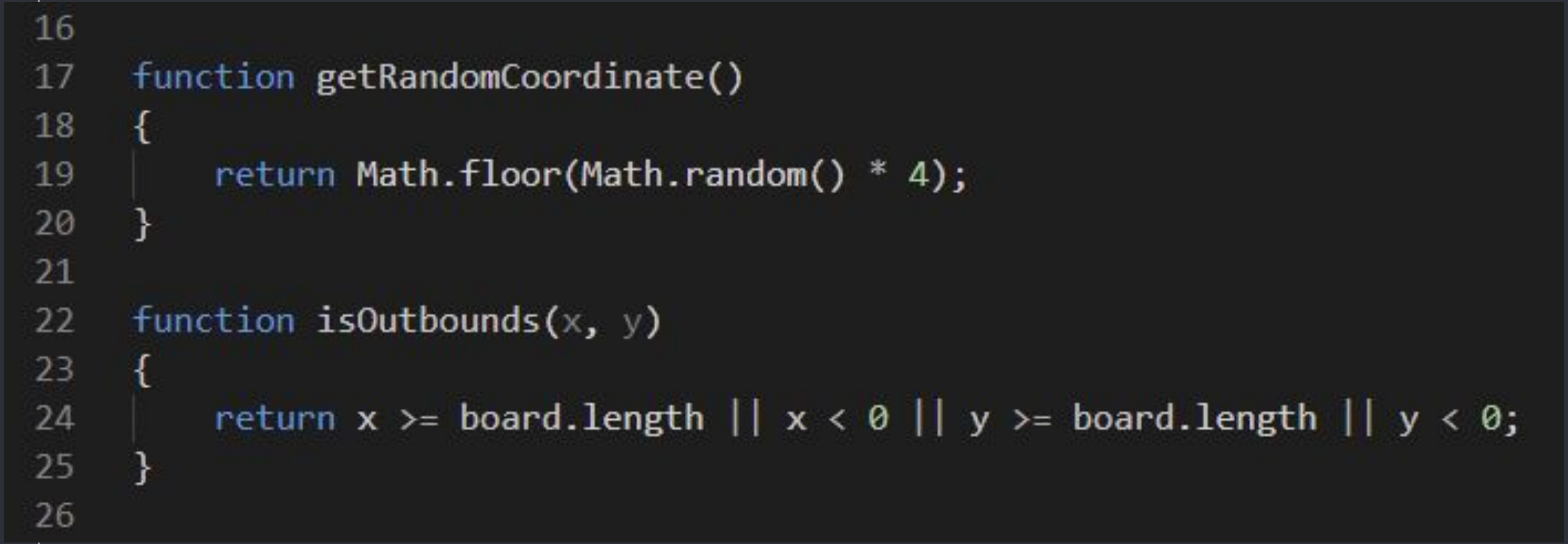


```
emcc -g -gsource-map --no-entry -s STANDALONE_WASM game_logic.cpp -o game_logic.html;
```



```
var importObject = {};
WebAssembly.instantiateStreaming(fetch('game_logic.wasm'), importObject)
.then((results) =>
{
    var isOutbounds = results.instance.exports.isOutbounds;
});
```

Great, onto the next function.



```

function getRandomCoordinate()
{
    return Math.floor(Math.random() * 4);
}

function isOutbounds(x, y)
{
    return x >= board.length || x < 0 || y >= board.length || y < 0;
}

```

2

WALL NUMBER 2

Where we’re going, there is no OS

SO RANDOM

With this standalone WASM file, there is no operating system level functionality.

SO RANDOM

With this standalone WASM file, there is no operating system level functionality.

You’re all on your own*

SO RANDOM

With this standalone WASM file, there is no operating system level functionality.

You’re all on your own*

So how do we solve this problem?

EMSCRIPTEN SAVIORS

Using random, calling timer functions and many other OS level functionality has to come from somewhere.

EMSCRIPTEN SAVIORS

Using random, calling timer functions and many other OS level functionality has to come from somewhere.

Thankfully there is a solution to this, where if you build a .js file in addition to your .wasm file, you will get many of these functionalities from the javascript side.

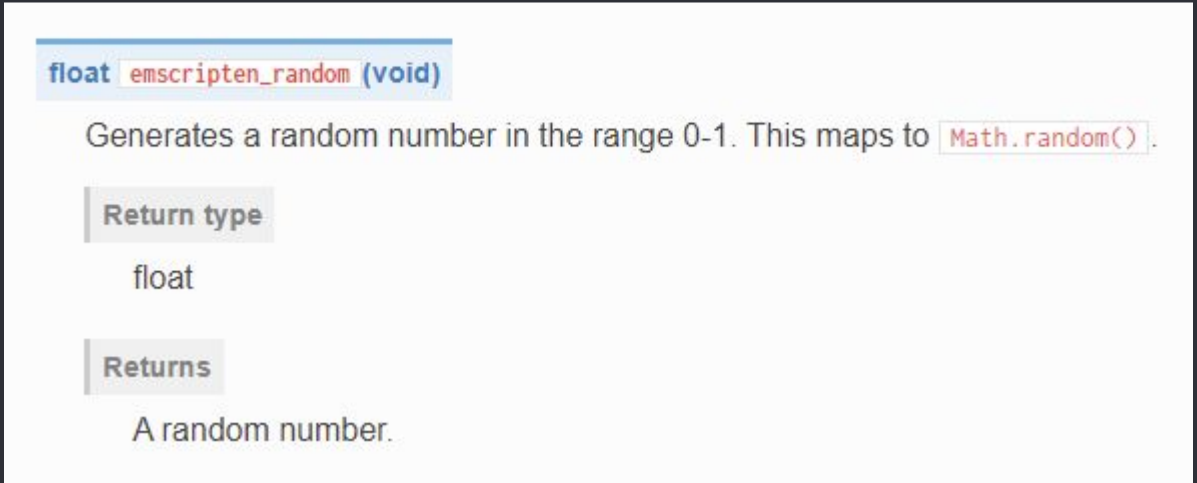
EMSCRIPTEN SAVIORS

Using random, calling timer functions and many other OS level functionality has to come from somewhere.

Thankfully there is a solution to this, where if you build a .js file in addition to your .wasm file, you will get many of these functionalities from the javascript side.

But how does it work? Can we do it ourselves?

EMSCRIPTEN RANDOM



float `emscripten_random` **(void)**

Generates a random number in the range 0-1. This maps to `Math.random()`.

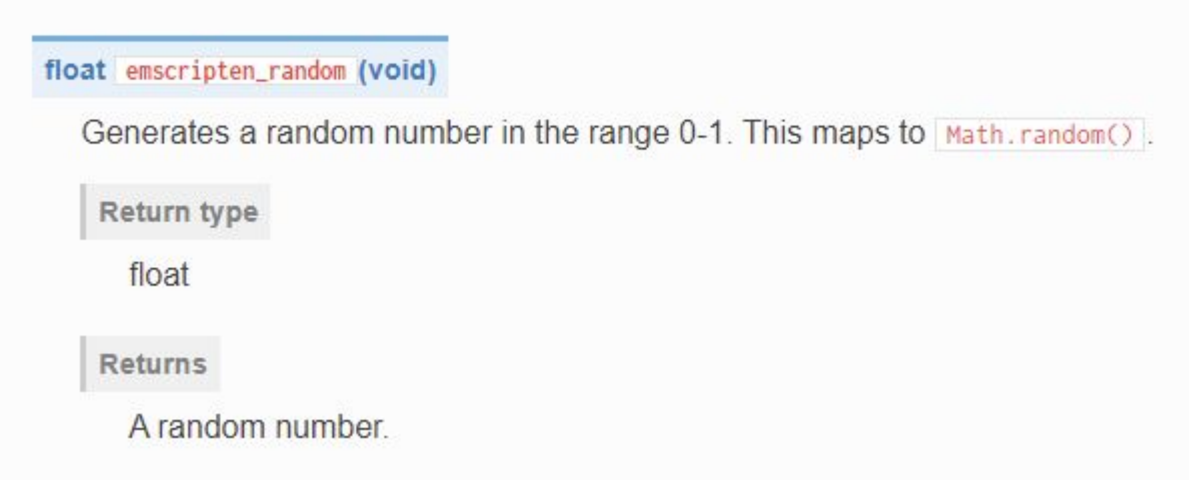
Return type

float

Returns

A random number.

EMSCRIPTEN RANDOM



float `emscripten_random` (void)

Generates a random number in the range 0-1. This maps to `Math.random()`.

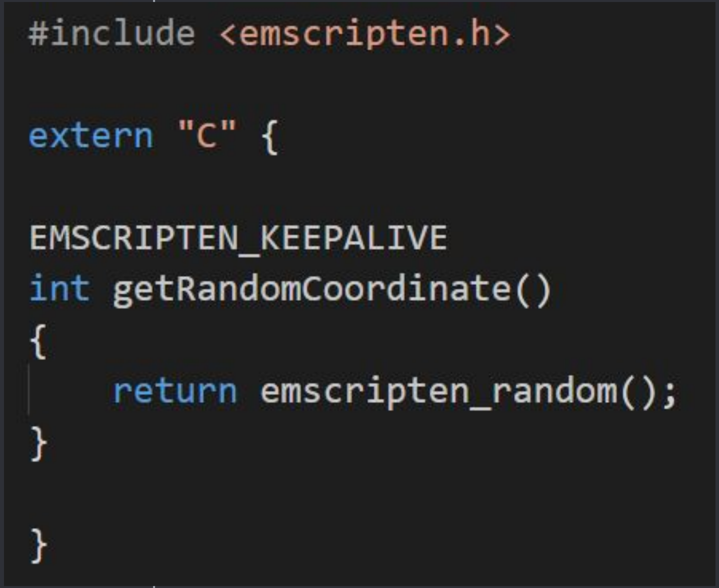
Return type

float

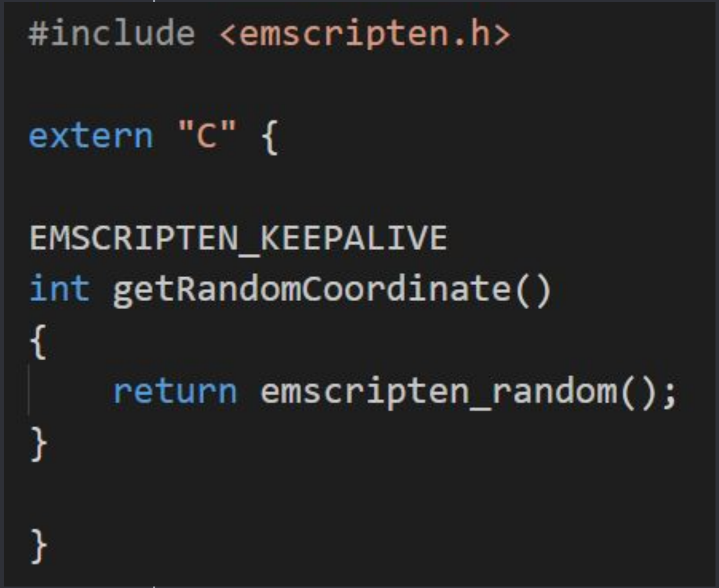
Returns

A random number.

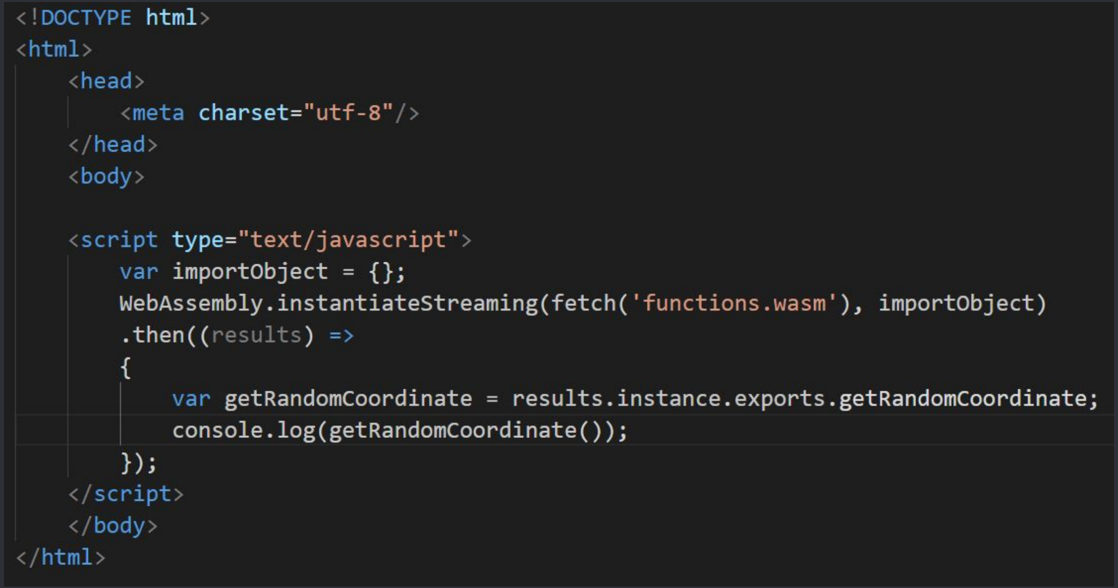
Looks great, but how do we use it?



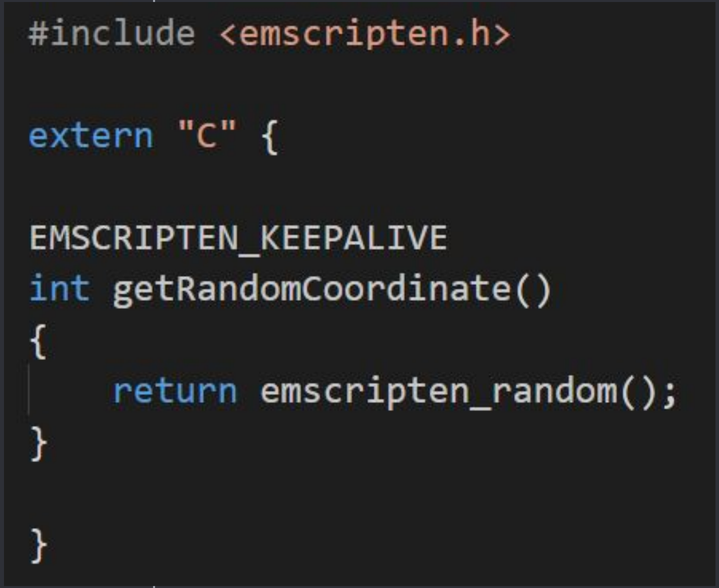
```
#include <emscripten.h>

extern "C" {

EMSCRIPTEN_KEEPALIVE
int getRandomCoordinate()
{
    return emscripten_random();
}

}
```



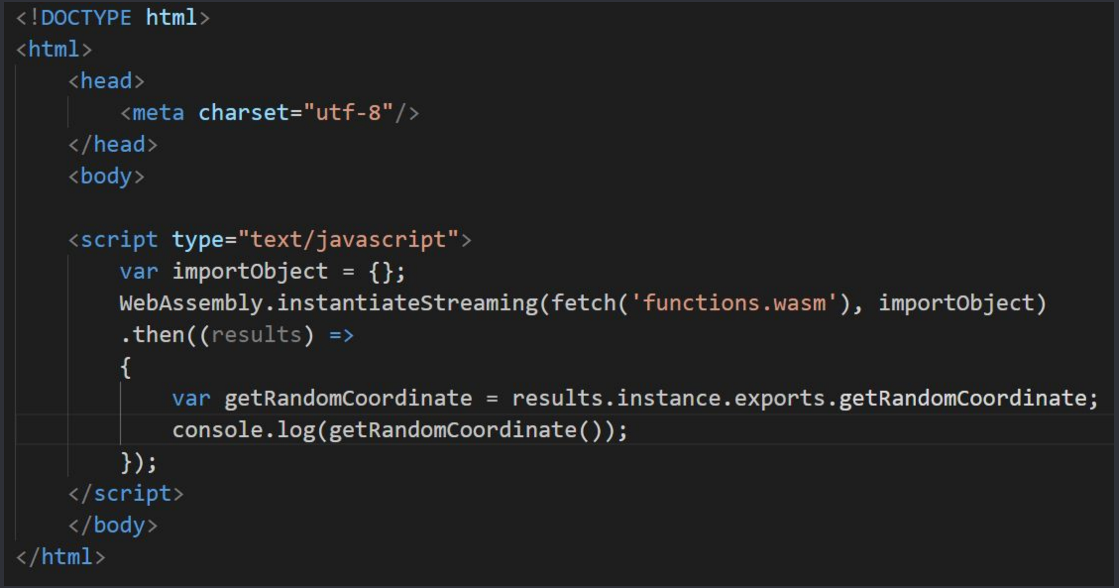
```c
#include <emscripten.h>

extern "C" {

EMSCRIPTEN_KEEPALIVE
int getRandomCoordinate()
{
    return emscripten_random();
}

}
```



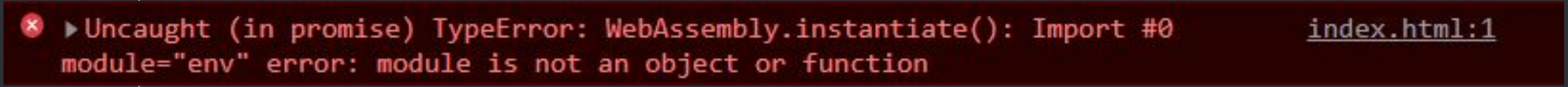
```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
    </head>
    <body>

    <script type="text/javascript">
        var importObject = {};
        WebAssembly.instantiateStreaming(fetch('functions.wasm'), importObject)
        .then((results) =>
        {
            var getRandomCoordinate = results.instance.exports.getRandomCoordinate;
            console.log(getRandomCoordinate());
        });
    </script>
    </body>
</html>
```



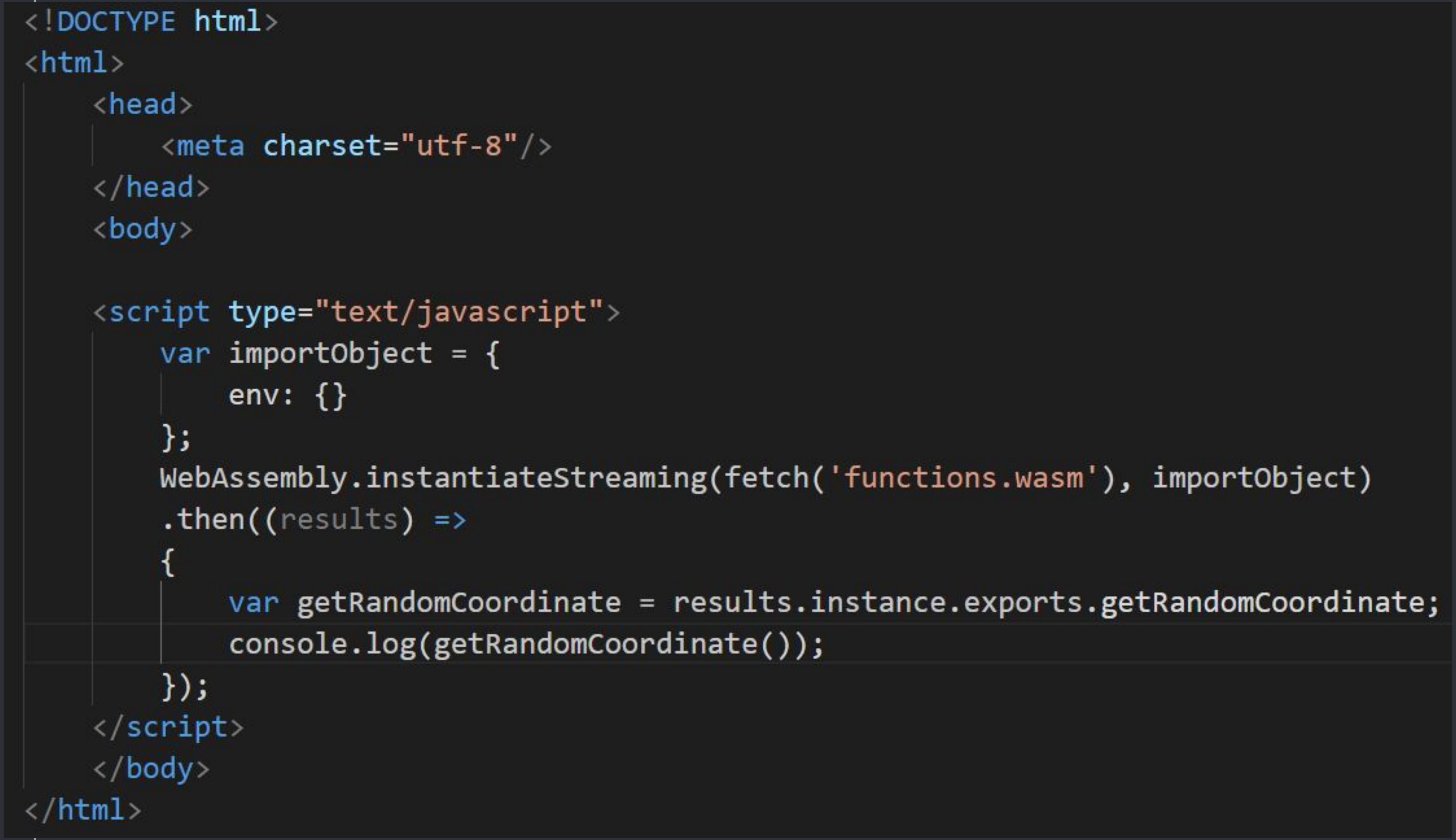
```
#include <emscripten.h>

extern "C" {

EMSCRIPTEN_KEEPALIVE
int getRandomCoordinate()
{
    return emscripten_random();
}

}
```



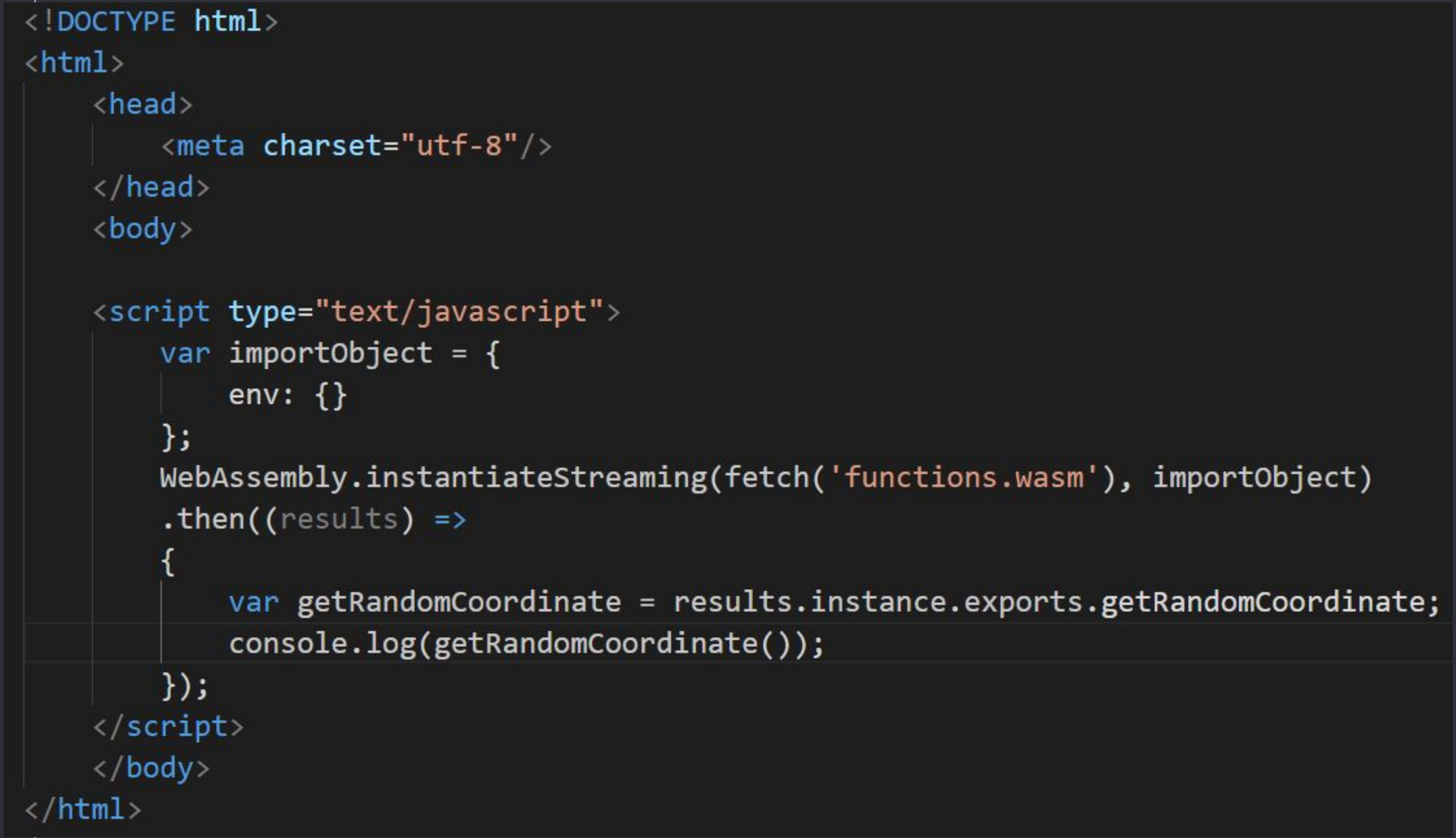
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
    </head>
    <body>

    <script type="text/javascript">
        var importObject = {};
        WebAssembly.instantiateStreaming(fetch('functions.wasm'), importObject)
        .then((results) =>
        {
            var getRandomCoordinate = results.instance.exports.getRandomCoordinate;
            console.log(getRandomCoordinate());
        });
    </script>
    </body>
</html>
```



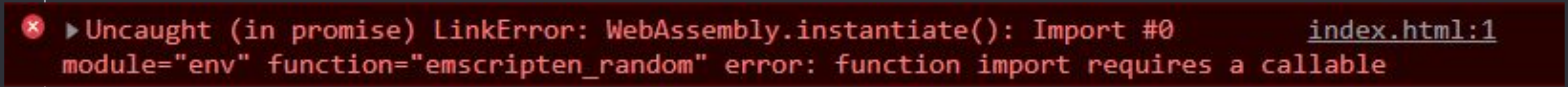
```
▶Uncaught (in promise) TypeError: WebAssembly.instantiate(): Import #0      index.html:1
module="env" error: module is not an object or function
```



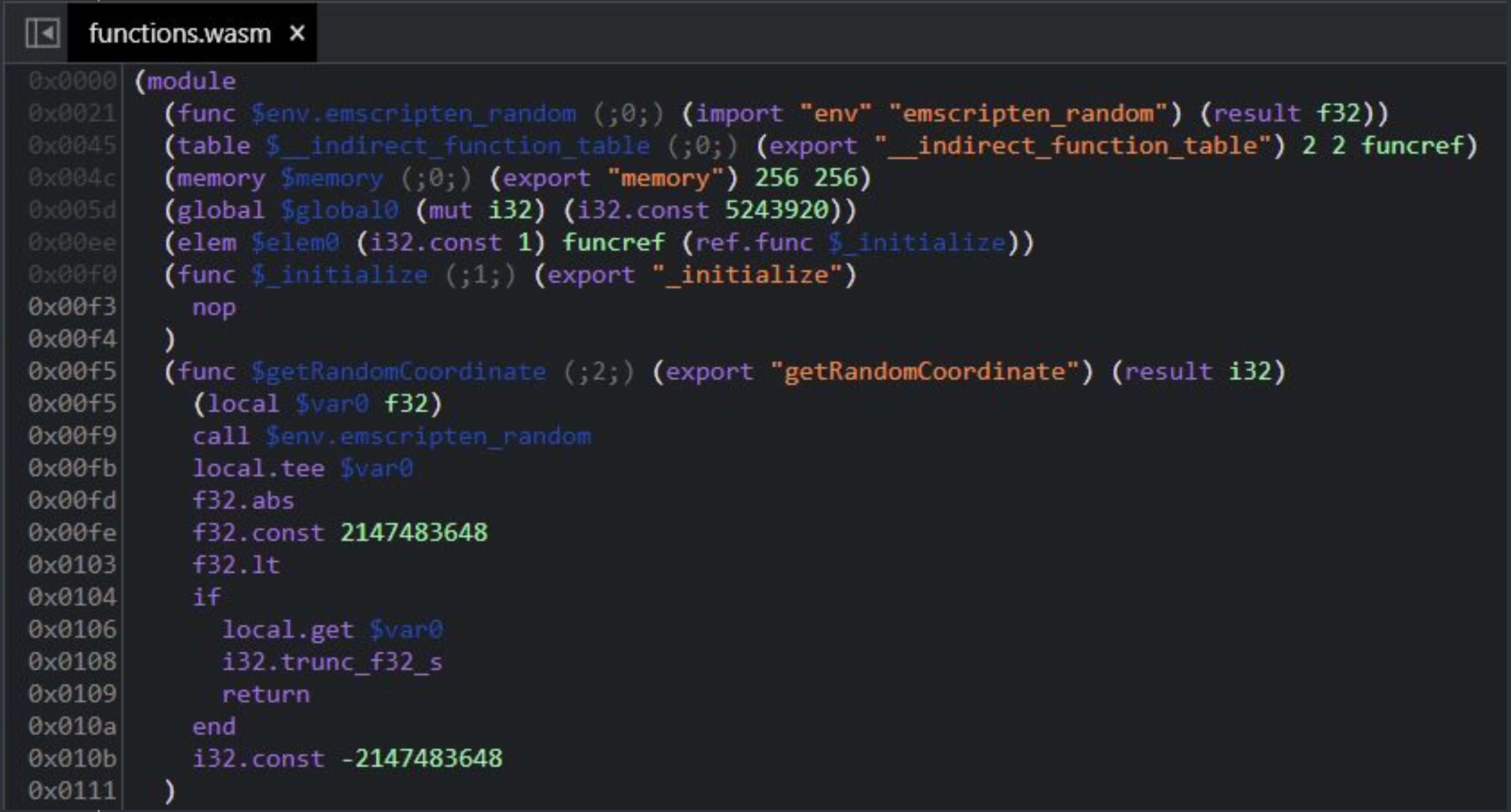
```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
    </head>
    <body>

    <script type="text/javascript">
        var importObject = {
            env: {}
        };
        WebAssembly.instantiateStreaming(fetch('functions.wasm'), importObject)
        .then((results) =>
        {
            var getRandomCoordinate = results.instance.exports.getRandomCoordinate;
            console.log(getRandomCoordinate());
        });
    </script>
    </body>
</html>
```



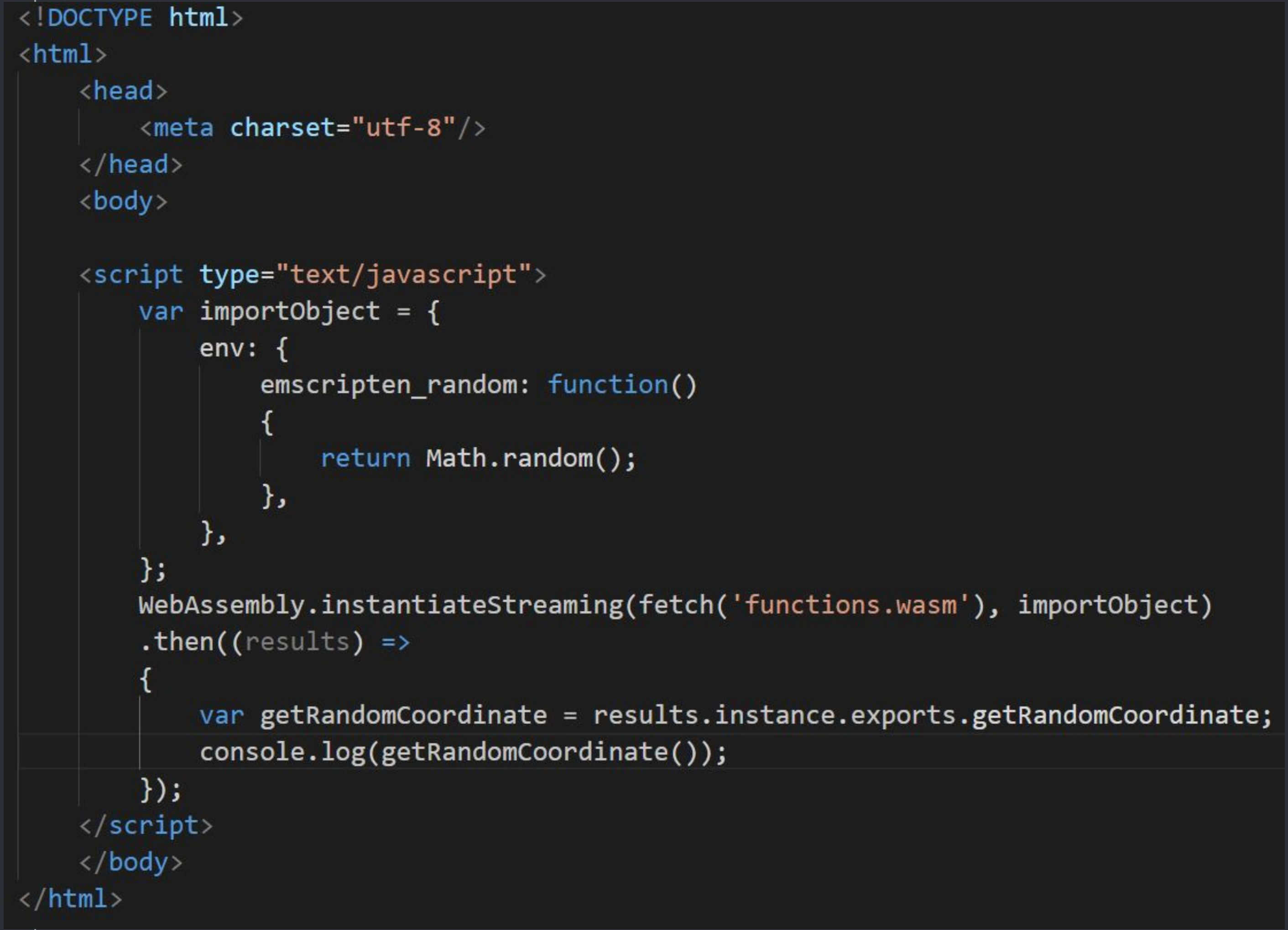
```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
    </head>
    <body>

    <script type="text/javascript">
        var importObject = {
            env: {}
        };
        WebAssembly.instantiateStreaming(fetch('functions.wasm'), importObject)
        .then((results) =>
        {
            var getRandomCoordinate = results.instance.exports.getRandomCoordinate;
            console.log(getRandomCoordinate());
        });
    </script>
    </body>
</html>
```



```
▶Uncaught (in promise) LinkError: WebAssembly.instantiate(): Import #0      index.html:1
module="env" function="emscripten_random" error: function import requires a callable
```



```
0x0000 (module
0x0021   (func $env.emscripten_random (;0;) (import "env" "emscripten_random") (result f32))
0x0045   (table $__indirect_function_table (;0;) (export "__indirect_function_table") 2 2 funcref)
0x004c   (memory $memory (;0;) (export "memory") 256 256)
0x005d   (global $global0 (mut i32) (i32.const 5243920))
0x00ee   (elem $elem0 (i32.const 1) funcref (ref.func $_initialize))
0x00f0   (func $_initialize (;1;) (export "_initialize")
0x00f3     nop
0x00f4   )
0x00f5   (func $getRandomCoordinate (;2;) (export "getRandomCoordinate") (result i32)
0x00f5     (local $var0 f32)
0x00f9     call $env.emscripten_random
0x00fb     local.tee $var0
0x00fd     f32.abs
0x00fe     f32.const 2147483648
0x0103     f32.lt
0x0104     if
0x0106       local.get $var0
0x0108       i32.trunc_f32_s
0x0109       return
0x010a     end
0x010b     i32.const -2147483648
0x0111   )
```



```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
    </head>
    <body>

    <script type="text/javascript">
        var importObject = {
            env: {
                emscripten_random: function()
                {
                    return Math.random();
                },
            },
        };
        WebAssembly.instantiateStreaming(fetch('functions.wasm'), importObject)
        .then((results) =>
        {
            var getRandomCoordinate = results.instance.exports.getRandomCoordinate;
            console.log(getRandomCoordinate());
        });
    </script>
    </body>
</html>
```

ONWARDS

Great, so now we can move over the rest of the game logic.

ONWARDS

Great, so now we can move over the rest of the game logic.

The board is an array of arrays of `Box` and the rest of the game logic is basically identical.

ONWARDS

Great, so now we can move over the rest of the game logic.

The board is an array of arrays of `Box` and the rest of the game logic is basically identical.

So now the gameplay can be simulated and called from JS, now we need to draw that data.

3

WALL NUMBER 3

Where’s the data?

I REMEMBER

We can communicate between C++ and JS using primitive types as you saw before, but as soon as things get a bit more complicated, we are in trouble.

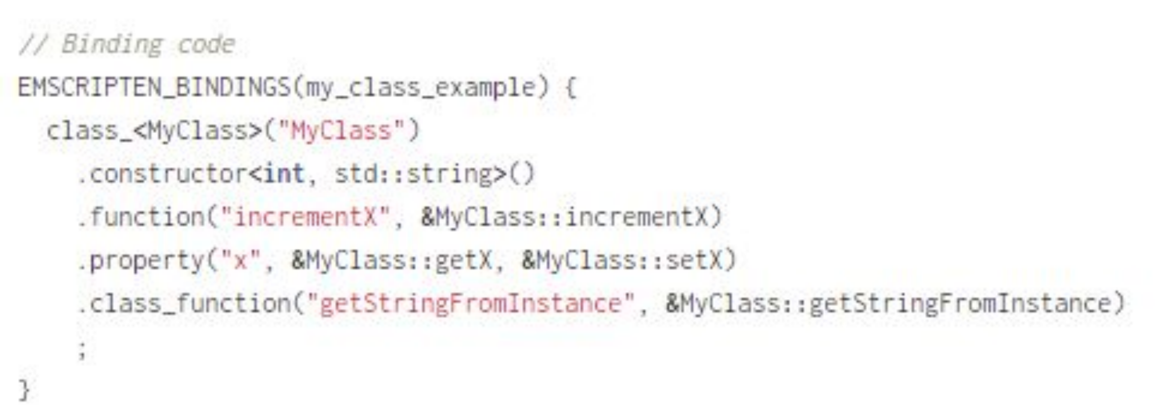
I REMEMBER

We can communicate between C++ and JS using primitive types as you saw before, but as soon as things get a bit more complicated, we are in trouble.

We could view the raw data of a std::vector within the memory of WebAssembly, but converting between a vector and a javascript list is not automatic

WE’RE IN A BIND

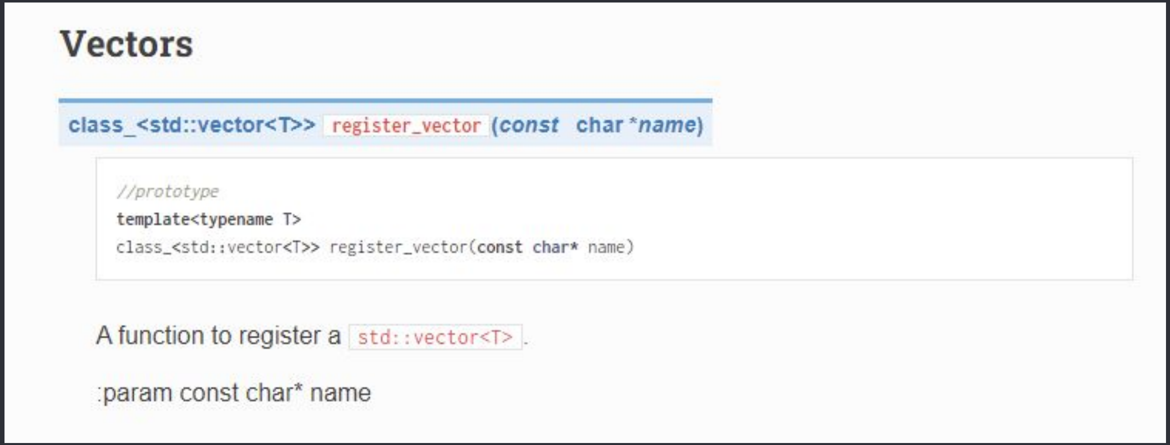
There is something called Embind that can help with passing more complex objects over to JS



```
// Binding code
EMSCRIPTEN_BINDINGS(my_class_example) {
  class_<MyClass>("MyClass")
    .constructor<int, std::string>()
    .function("incrementX", &MyClass::incrementX)
    .property("x", &MyClass::getX, &MyClass::setX)
    .class_function("getStringFromInstance", &MyClass::getStringFromInstance)
    ;
}
```

WE'RE IN A BIND

Embind even has helpers to bind common objects, like std::vector



Vectors

class_<std::vector<T>> register_vector (const char *name)

```
//prototype
template<typename T>
class_<std::vector<T>> register_vector(const char* name)
```

A function to register a `std::vector<T>`.

:param const char* name

I REMEMBER

You can even define a shared block of memory that can then be used by either JS or C++

I REMEMBER

You can even define a shared block of memory that can then be used by either JS or C++

Also there is the option to return a pointer to JS

I REMEMBER

You can even define a shared block of memory that can then be used by either JS or C++

Also there is the option to return a pointer to JS

But this is in the territory where you need to be a bit more careful with how each byte is used and represented.

WE DON’T NEED IT

Thankfully, I wrote the game logic to only use simple primitives, so we can finish converting all of the functions over to C++ and expose them to JS to use as needed.

WE DON’T NEED IT

Thankfully, I wrote the game logic to only use simple primitives, so we can finish converting all of the functions over to C++ and expose them to JS to use as needed.

Let’s look at this version of the implementation.

LET’S NOT STOP HERE!

Now we have basically everything except the rendering in the C++ version.

LET’S NOT STOP HERE!

Now we have basically everything except the rendering in the C++ version.

So let’s move that over as well.

LET’S NOT STOP HERE!

Now we have basically everything except the rendering in the C++ version.

So let’s move that over as well.

Thankfully Emscripten has great support for exactly what we need.

SDL1 and 2

Emscripten has built in support for SDL which is a cross platform library that provides among many things graphical rendering support.

SDL1 and 2

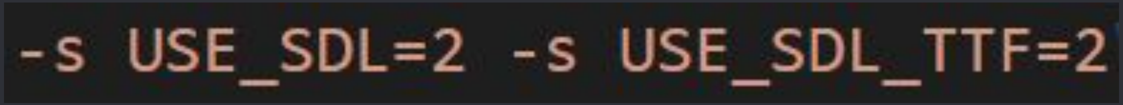
Emscripten has built in support for SDL which is a cross platform library that provides among many things graphical rendering support.

There is also support for SDL2 but it needs to be downloaded (which happens on first compile)

SDL1 and 2

Emscripten has built in support for SDL which is a cross platform library that provides among many things graphical rendering support.

There is also support for SDL2 but it needs to be downloaded (which happens on first compile)



```
-s USE_SDL=2 -s USE_SDL_TTF=2
```

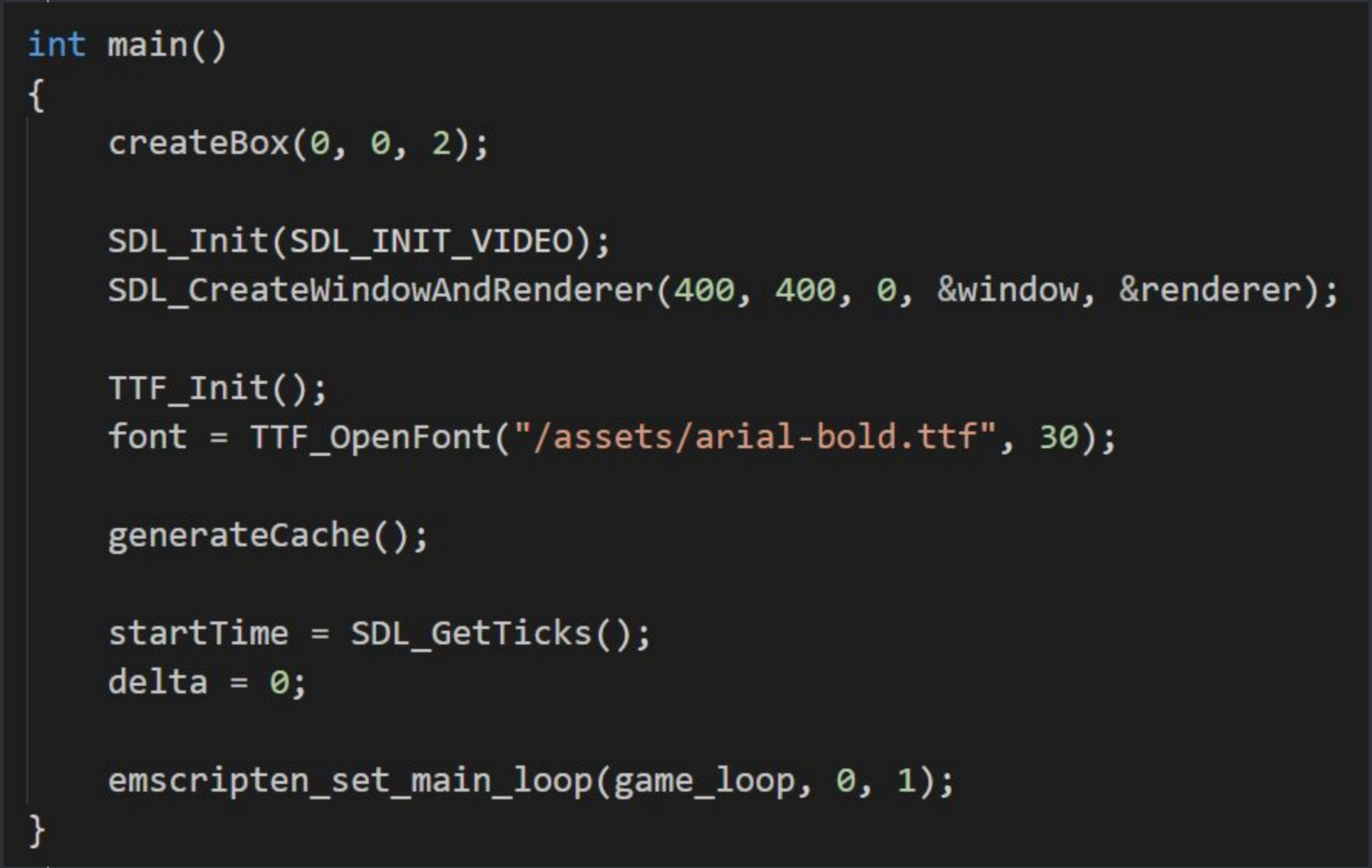
GLUE THAT CODE

Also since we will use SDL2 and other built in functionality, we will use the generated JS glue code.

GLUE THAT CODE

Also since we will use SDL2 and other built in functionality, we will use the generated JS glue code.

So instead of creating the importObject ourselves and implementing the functions that are needed, Emscripten has does this for us.



```c
int main()
{
    createBox(0, 0, 2);

    SDL_Init(SDL_INIT_VIDEO);
    SDL_CreateWindowAndRenderer(400, 400, 0, &window, &renderer);

    TTF_Init();
    font = TTF_OpenFont("/assets/arial-bold.ttf", 30);

    generateCache();

    startTime = SDL_GetTicks();
    delta = 0;

    emscripten_set_main_loop(game_loop, 0, 1);
}
```

RENDERING FUN

Now I port over the rendering code, which thankfully for this example is just a simple colored rectangle. (I wait with displaying the text for now)

RENDERING FUN

Now I port over the rendering code, which thankfully for this example is just a simple colored rectangle. (I wait with displaying the text for now)

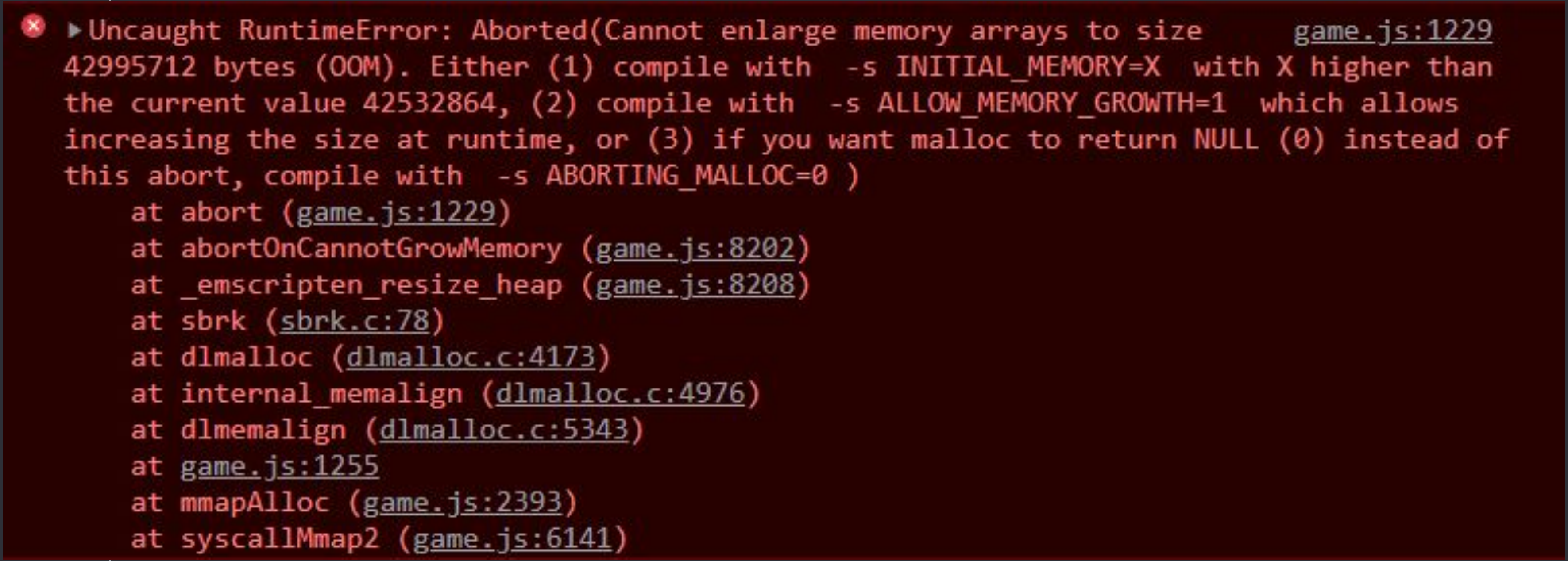
Everything compiles and looks like it should be.

RENDERING FUN

Now I port over the rendering code, which thankfully for this example is just a simple colored rectangle. (I wait with displaying the text for now)

Everything compiles and looks like it should be.

I run the code, I see the box and then…



```
▸Uncaught RuntimeError: Aborted(Cannot enlarge memory arrays to size      game.js:1229
42995712 bytes (OOM). Either (1) compile with  -s INITIAL_MEMORY=X  with X higher than
the current value 42532864, (2) compile with  -s ALLOW_MEMORY_GROWTH=1  which allows
increasing the size at runtime, or (3) if you want malloc to return NULL (0) instead of
this abort, compile with  -s ABORTING_MALLOC=0 )
    at abort (game.js:1229)
    at abortOnCannotGrowMemory (game.js:8202)
    at _emscripten_resize_heap (game.js:8208)
    at sbrk (sbrk.c:78)
    at dlmalloc (dlmalloc.c:4173)
    at internal_memalign (dlmalloc.c:4976)
    at dlmemalign (dlmalloc.c:5343)
    at game.js:1255
    at mmapAlloc (game.js:2393)
    at syscallMmap2 (game.js:6141)
```

4

WALL NUMBER 4

The sandbox isn’t infinite

MEMORY MANAGEMENT

Up to this point I have been using the default memory size and it has just happened to fit.

MEMORY MANAGEMENT

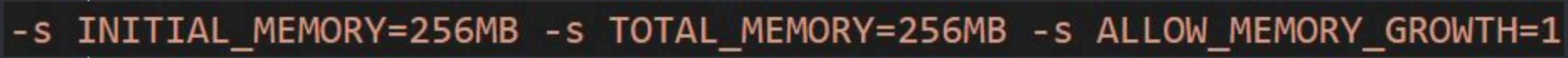
Up to this point I have been using the default memory size and it has just happened to fit.

But we need more memory now since SDL is involved.

MEMORY MANAGEMENT

Up to this point I have been using the default memory size and it has just happened to fit.

But we need more memory now since SDL is involved.



```
-s INITIAL_MEMORY=256MB -s TOTAL_MEMORY=256MB -s ALLOW_MEMORY_GROWTH=1
```

TEXT ADVENTURE

Great, this compiles and we see the box drawn in the canvas as before.

TEXT ADVENTURE

Great, this compiles and we see the box drawn in the canvas as before.

So let’s draw the text that should appear within the box.

5

WALL NUMBER 5

File not found

EMPTY SANDBOX

The environment we are in does not have much else outside of what we have given it.

EMPTY SANDBOX

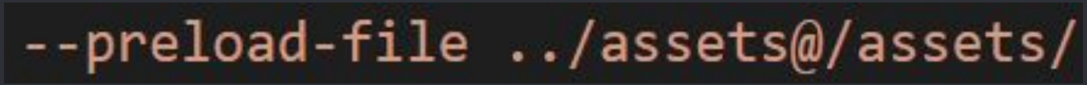
The environment we are in does not have much else outside of what we have given it.

So the font file we want to use does not exist, and the idea of a filesystem is different from what we expect. We have to provide the files.

EMPTY SANDBOX

The environment we are in does not have much else outside of what we have given it.

So the font file we want to use does not exist, and the idea of a filesystem is different from what we expect. We have to provide the files.

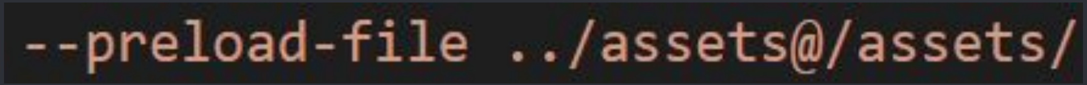


```
--preload-file ../assets@/assets/
```

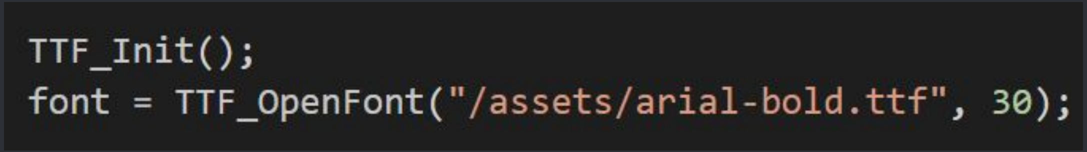
EMPTY SANDBOX

The environment we are in does not have much else outside of what we have given it.

So the font file we want to use does not exist, and the idea of a filesystem is different from what we expect. We have to provide the files.



```
--preload-file ../assets@/assets/
```



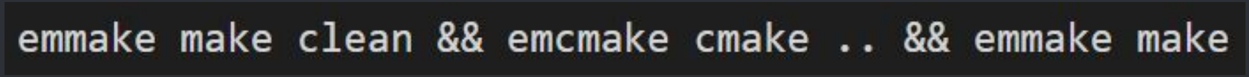
```
TTF_Init();
font = TTF_OpenFont("/assets/arial-bold.ttf", 30);
```

CMAKE

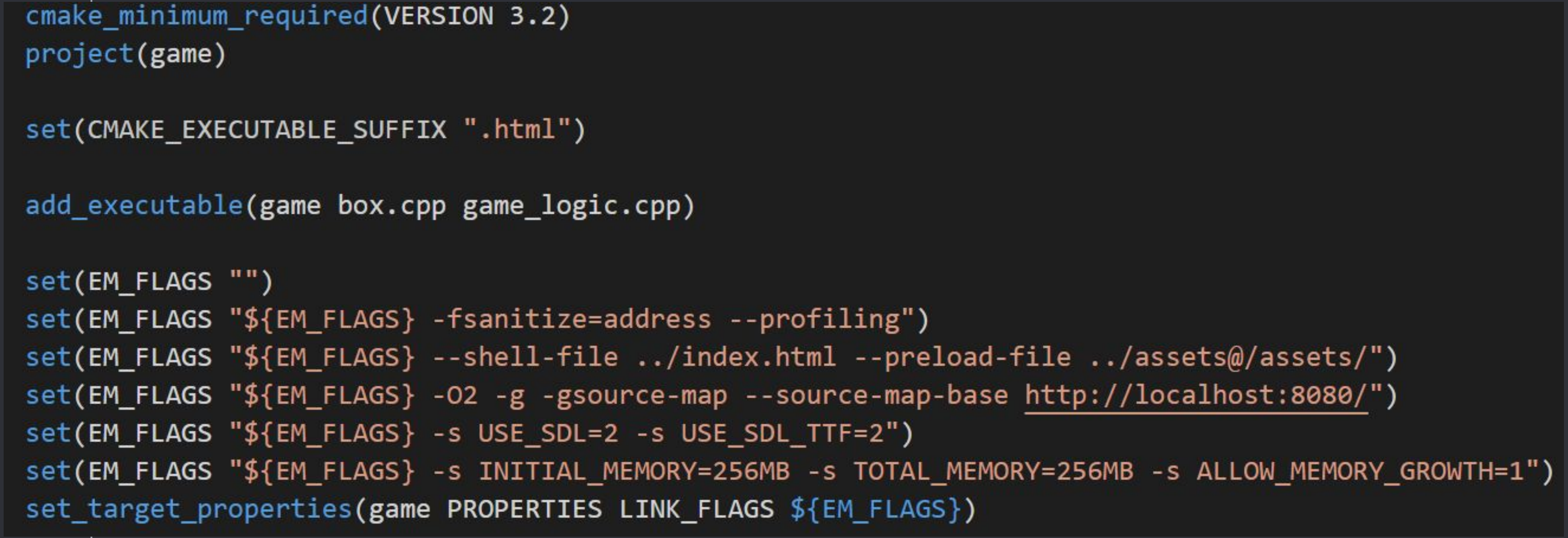
What Emscripten also provides is helper utilities to use common development tools like make and cmake. So I also wrote a simple CMake file for building the project.

CMAKE

What Emscripten also provides is helper utilities to use common development tools like make and cmake. So I also wrote a simple CMake file for building the project.



```
emmake make clean && emcmake cmake .. && emmake make
```

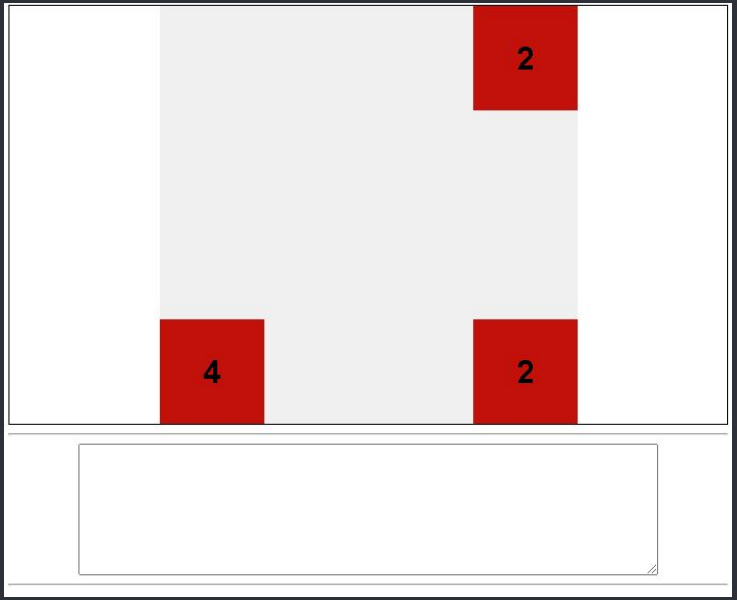


```
cmake_minimum_required(VERSION 3.2)
project(game)

set(CMAKE_EXECUTABLE_SUFFIX ".html")

add_executable(game box.cpp game_logic.cpp)

set(EM_FLAGS "")
set(EM_FLAGS "${EM_FLAGS} -fsanitize=address --profiling")
set(EM_FLAGS "${EM_FLAGS} --shell-file ../index.html --preload-file ../assets@/assets/")
set(EM_FLAGS "${EM_FLAGS} -O2 -g -gsource-map --source-map-base http://localhost:8080/")
set(EM_FLAGS "${EM_FLAGS} -s USE_SDL=2 -s USE_SDL_TTF=2")
set(EM_FLAGS "${EM_FLAGS} -s INITIAL_MEMORY=256MB -s TOTAL_MEMORY=256MB -s ALLOW_MEMORY_GROWTH=1")
set_target_properties(game PROPERTIES LINK_FLAGS ${EM_FLAGS})
```

IT'S RUNNING!

Great! So now we have everything running.

Let's look at it in action!



SUMMARY

Let’s summarize the walls we encountered.

SUMMARY

Let’s summarize the walls we encountered.

- Files need to be served while developing

SUMMARY

Let’s summarize the walls we encountered.

- Files need to be served while developing
- All functionality you depend on (ie. OS) needs to be implemented or given to you

SUMMARY

Let’s summarize the walls we encountered.

- Files need to be served while developing
- All functionality you depend on (ie. OS) needs to be implemented or given to you
- Data needs to be primitives or converted in some way before sending to JS

SUMMARY

Let’s summarize the walls we encountered.

- Files need to be served while developing
- All functionality you depend on (ie. OS) needs to be implemented or given to you
- Data needs to be primitives or converted in some way before sending to JS
- Memory size and growth needs to be thought about

SUMMARY

Let’s summarize the walls we encountered.

- Files need to be served while developing
- All functionality you depend on (ie. OS) needs to be implemented or given to you
- Data needs to be primitives or converted in some way before sending to JS
- Memory size and growth needs to be thought about
- Required files need to be embedded or preloaded with the output

Ólafur Waage

Senior Software Developer - TurtleSec AS
@olafurw on Twitter
https://github.com/olafurw/talk-cppp-webassembly