Bachelor's thesis

Degree Programme in Information and Communications Technology

2023

Yakub Kapri

RUNNING NATIVE CODE IN THE BROWSER USING WEBASSEMBLY

– Demonstration

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Degree Programme in Information and Communications Technology

2023 | 28

Yakub Kapri

RUNNING NATIVE CODE IN THE BROWSER USING WEBASSEMBLY

- Demonstration

This thesis aims to demonstrate the potential of WebAssembly as a powerful tool for developing web applications with native code. The thesis provides an overview of WebAssembly and its predecessor technologies and showcases the creation of a Pong game as a practical example. The logic of the game was created using C++ and compiled into WebAssembly using Emscripten.

The thesis also outlines the development workflow, including setting up the development environment, calling C++ functions from JavaScript, making changes to HTML, and the compilation process. The result of this endeavour was the successful execution of the Pong game on a web browser, utilizing WebAssembly for running native C++ code.

This thesis serves as an example for developers aiming to harness the capabilities of WebAssembly and leverage the advantage of native code to create compelling and robust web applications.

Keywords:

JavaScript, WebAssembly, Compilation target, Low-Level language, C/C++, Portability

Contents

Appendices

Pictures

List of Abbreviations

AOT	Ahead Of Time
CPU	Central Processing Unit
CSS	Cascading Style Sheets
HTML	Hyper Text Markup Language
JIT	Just In Time
MVP	Minimum Viable Product
NaCl	Native client
OS	Operating System
PNaCl	Portable Native Client
SDK	Software Development Kit
VM	Virtual Machine
W3C	World Wide Web Consortium
Wasm	WebAssembly
WAT	WebAssembly Text Format

1 Introduction

For a long time, JavaScript has been the only standard web scripting language. 98% of websites today use JavaScript as their client-side programming language (W3Techs, 2023). By design, JavaScript is a dynamically typed language, which has limitations and design faults as a language from the start. For example, JavaScript lacks high precision. It only supports 64-bit floating-point numeric, which limits its use for certain types of mathematical computation. JavaScript has improved with time and maintained its ambiguous presence on the web for a long time. Many powerful software are hard to write in JavaScript and gain native performance. There have been numerous attempts to run native code and near-native speed applications on the web. The promising technology in this field that solves the shortcomings of its predecessors is WebAssembly.

WebAssembly is a low-level assembly-like language with a binary instruction format for a stack-based virtual machine. It is designed to be portable (capable of running in different Operative Systems and architecture) and compilation target for higher languages such as C/C++, C# and Rust (WebAssembly, 2023).

This thesis focuses on web applications. For the demonstration part of the thesis, a simple pong game is created. The game features two paddles and a ball, where the objective is to score points by getting the ball beyond the opponent's paddle. The game is played by a single player against a computer-controlled paddle. The project also highlights the setup of a development environment.

2 Predecessors

Several attempts have been made in the past to support native code in browsers. The following section will discuss the most noticeable technology that tried to solve the problem.

2.1 ActiveX and Flash

In 1996 Microsoft announced ActiveX, which allowed developers to embed dynamic content on a web page (Microsoft, 1996). But it could not get acceptance on a large scale. Mostly tied to one specific platform (x86-based computer) and later deprecated (Microsoft Windows Blogs, 2015).

In 1995 another technology was released FutureSplash which changed its name to Flash in 1996. It was a platform for providing dynamic and multimedia content on web pages. Flash had a significant part on the web, but it suffered from critical security vulnerability over the years (Hoffmann, 2017) (Wressnegger et al., 2016).

With the rapid growth of the mobile platform, a plug-in model was inefficient for smartphones and tablets. Because these devices have limited memory and processing power compared to their desktop counterparts. It was Apple's dropping its support for Flash that triggered a decline in uses and the end of the plug-in era (Newstex, 2010).

2.2 Native Client (NaCl) and Portable Native Client (PNaCl)

In 2011 Google released NaCl which allowed users to run machine code in a browser by using sandboxing technology. NaCl does it by not interacting with other parts of the systems, applications, or platforms in which they run. NaCl performs at near-native speed by using the user's CPU capabilities. Because of these developers were able to write programs in C and C++, to use system services like sound and graphic hardware while maintaining the security in the

web. NaCl also gained the same service access as JavaScript (Donovan et al., 2010). Despite the fact NaCl was designed as a platform independent from the beginning, it required generating different executables ".nexe" modules for different hardware architecture (e.g., ARM, x86-64, or MIPS). And it could be distributed only through the Google Play store. These limitations were the main reasons for NaCl not being portable.

PNaCl was the improved version of NaCl considering the portability and suitability for the web. PNaCl was able to do that by generating single executable "pexe" modules, which were then translated into a native architecture of the client's machine. Host-specific executables were generated during the time of module loading and before executing code. But code portability and application portability are not the same. Applications require APIs, they rely on to be available, to function properly.

Google provided Pepper APIs for low-level services like audio playback, file access, graphics libraries and more (Google developers, 2021). While PNaCl could run in Chrome browsers on different platforms but Pepper APIs dependent applications could only run in browsers that provided implementations of the Pepper APIs.

In general (P)NaCl received mixed feedback from developers. Some argued and praised its security, performance, and possibility of applications on the web, alternative to JavaScript-written applications. While others criticized it, notably Mozilla and Opera, saying it is shifting the focus away from web platforms. Arguments were made on technical aspects concerning what the web should and should not be (Metz, 2011). In 2017 google deprecated (P)NaCl in favor of WebAssembly. The main reason was less use of (P)NaCl and the web community favored WebAssembly as a cross-browser solution (Nelson, 2017).

2.3 Emscripten and asm.js

Emscripten is a compiler from Low-Level Virtual Machine (LLVM) assembly to JavaScript or a subset of JavaScript “asm.js”(Zakai, 2011). Emscripten is a completely open-source WebAssembly compiler toolchain(Emscripten, 2021).

Luke Wagner, who works on the performance of JavaScript in Firefox, began collaborating with Zakai to modify the Emscripten to output the only patterns which performed well. This resulting subset of JavaScript became asm.js, which was a distinct language. asm.js was a highly modified subset of JavaScript. It was used for running process-intensive programs on the web platform. The common use case was to port the native C/C++ code to run on the browsers. Because of its design, it was not intended to be handwritten but to be used as a target language for compilers. Notable examples have been the Qt applications framework and game engines, Unity, and Unreal Engine (Wagner, 2017).

One of the distinct features of asm.js was Ahead-of-time (AOT) compilation, unlike standard JavaScript, which uses Just-in-Time (JIT) compilation. AOT compiling provides performance boosts by not having runtime type checks, absence of garbage collection, and efficient heap load and stores. However, code that fails to validate during AOT compilation falls back to standard methods, JIT compilation and regular interpretation (Herman et al., 2014).

asm.js was introduced in 2013 and developed within Mozilla. Firefox was the first browser to implement asm.js optimizations. Later Mozilla published the full specification of asm.js and encouraged other vendors to implement asm.js specific optimization. However, not all browsers implemented the optimizations for asm.js. The main problem was the lack of standards. Different browser vendors approached asm.js optimizations from different angles, in addition, asm.js also had limited features such as standard JavaScript, being its subset. This included features like 64-bit integer operations and threads (Wagner, 2017).

3 WebAssembly

WebAssembly (Wasm) is a low-level assembly-like language with a binary instruction format for a stack-based virtual machine. It is designed to be portable and compilation target for higher languages like C/C++, Rust, and many others, enabling deployment on the web for client and server applications with near-native speed. It is designed to complement and run alongside JavaScript by using the WebAssembly JavaScript APIs, WebAssembly on its own is not intended to be handwritten, but rather used as a compilation target (Watt, 2018). However, there is a textual representation of the WebAssembly to read and edit called WebAssembly text format (WAT) (Mozilla, 2023). The text format is the representation of the stack-based nature of the language through nested expressions. The primary goal of the text format is for debugging, while the binary format (.wasm file) is delivered to and compiled by browsers.

WebAssembly is the first W3C (World Wide Web Consortium) based open standard for running native code on the web and the fourth language to run natively in web browsers, HTML, CSS, and JavaScript being other languages. It was first officially announced in June 2015 (W3C Team, 2015). In November 2017, the WebAssembly Community Group consisting of four major vendors (Chrome, Edge, Firefox and WebKit) reached the initial Minimum Viable Product (MVP) design and WebAssembly 1.0 has shipped in major browsers engine (WebAssembly, 2023).

Design

WebAssembly design goal is to be a safe, fast, universal, compact, and portable language. It is designed to be independent of hardware and platform. WebAssembly is intended to work with or without JavaScript Virtual Machine (Haas et al., 2017).

Security

The web is the primary target for WebAssembly, so security consideration was included from the beginning of language design. On the web, codes are loaded from diverse sources and often they are untrusted sources. Protecting the host environment from such code is achieved by running them in a virtual machine (VM), like JavaScript. This technique is highly effective in preventing programs to compromise users' data and system state (WebAssembly, 2023).

Like JavaScript, WebAssembly runs in a sandbox and as a result, isolates the exploits by not allowing execution to escape the runtime environment. This does not mean it prevents safety bugs from compromising WebAssembly code or the data it uses. While code from C/C++ is compiled to WebAssembly, it inherits the classic memory safety vulnerabilities, such as buffer overflow and use-after-free. Also compiling into WebAssembly does not mitigate the vulnerabilities that propagate from the insecure source language. New processor features like tagged memory, fine-grain capabilities and pointer authentications can be detected and prevented with minimal overhead. WebAssembly's just-in-time compiler cannot take advantage of such features because pieces of information are lost while compiling into WebAssembly (Disselkoen et al., 2019).

In general, WebAssembly in its security design is safe with its sandboxed environment. This does not mean unsafe language will be safe after compiling to WebAssembly, which is not its design goal. With emerging modern technologies, there is always room for potential security vulnerabilities which need to be considered (Alladoum, 2018).

Performance

WebAssembly specification promises an execution speed as fast as or near-native code across different platforms. It is optimized ahead of time and compiled to machine code on the host (Haas et al., 2017).

Decoding WebAssembly is simpler and faster than parsing JS; furthermore, this decode/compile step can be split across multiple threads, and the entire process starts while the module is still downloading. This reduces the time it takes to download the application code and reach peak execution speed (Clark, 2018).

Portability and Compilation target

WebAssembly format does not restrict itself to any one browser, hardware, or operating systems type. Portability in the context of WebAssembly means that WebAssembly's binary format is a compact binary instruction format for stack-based virtual machines. It is specifically designed as a portable compilation target for higher-level languages like C, C++, C#, and Rust. This means WebAssembly can be executed on different operating systems and instruction set architectures, both on and off the web (WebAssembly, 2023).

4 Project Pong Game

4.1 Overview of Game

Most functions of the game are written in JavaScript, but a simple Intelligent function for controlling the computer paddle called “get_ai_move” is written in C++. The function automatically determines how the computer positions the paddle to bounce the ball back to the opponent player. The application aims to call the C++ function by JavaScript through WebAssembly.

C++ was chosen as the language to use for creating the function. This was because C++ is among the first languages which support compilation to Wasm. Note that due to lack of time, the styling of the user interface of the application was not a priority.

4.2 Setting up a Development Environment

We will need some applications and tooling to start developing WebAssembly.

This section will introduce the tools and technologies used in the project, and how to install and configure it on a local machine. After setting up the toolchain we will use the C++ compiler to generate WASM. After that, we will call the C++ function with JavaScript and finally, we will pass complex data with Embind.

Operating systems and hardware

- macOS Big Sur, version 11.6.2
- 2,4 GHz Dual-Core Intel Core i5
- 8 GB 1600 MHz DDR3

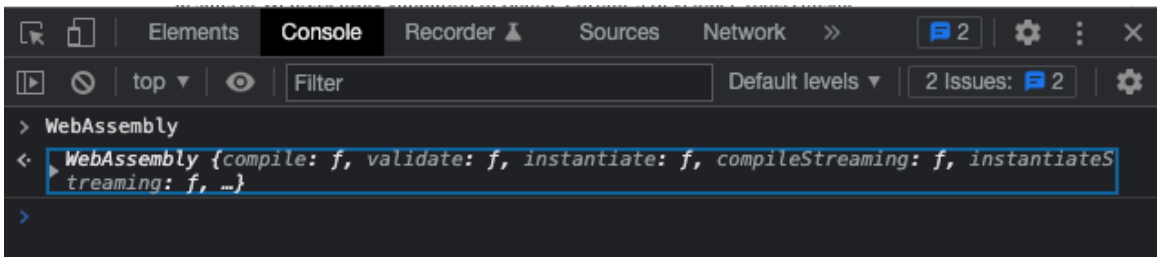
Text Editor

Visual Studio code is the text editor used for writing code, as the author is familiar with the software. A few extensions were configured to simplify the WebAssembly development process.

- C/C++ for visual studio code
- WebAssembly Toolkit for VSCode
- Live Server

Browsers

Google Chrome is used as a browser to run the application (Picture 1).



Picture 1. Results of WebAssembly validation in Chrome console.

Git

Git is a distributed version control system that allows tracking changes to files. Git will allow us to clone repositories from GitHub and is a prerequisite for the EMSDK. Git version 2.21.1 is used during the development of the project.

Emscripten

Emscripten is the source-to-source compiler. It is used as a build tool to generate the Wasm modules. Below are the steps for installing and setting up Emscripten on the local machine.

1. To install Emscripten we must clone from Git repo from the Emscripten project:
   git clone http://github.com/emscripten-core/emsdk.git

2. Enter the directory emsdk:
   cd emsdk

3. Fetch the latest version of the emsdk (not required for the first-time clone):
   git pull

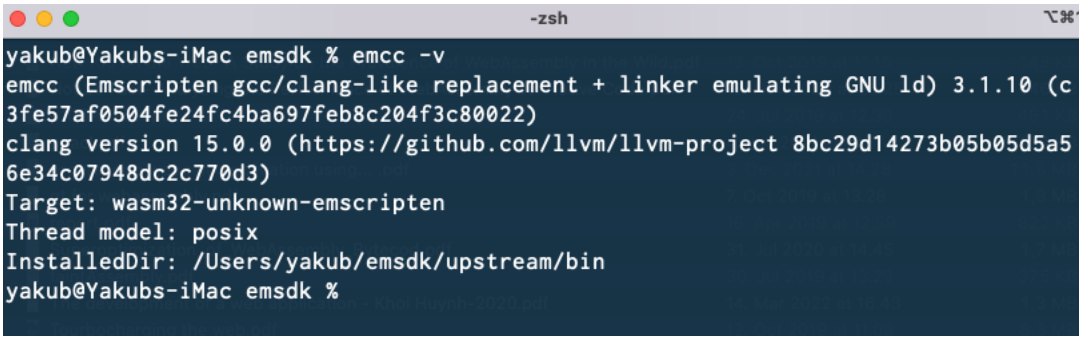
4. Download and install the latest SDK tools:
   ./emsdk install latest

5. Activate the latest version of Emscripten:
   ./emsdk activate latest

   The “activate” command sets up configuration files and environment variables. However, it does not add Emscripten to the system’s path. To add it, you need to run a shell script and source it.

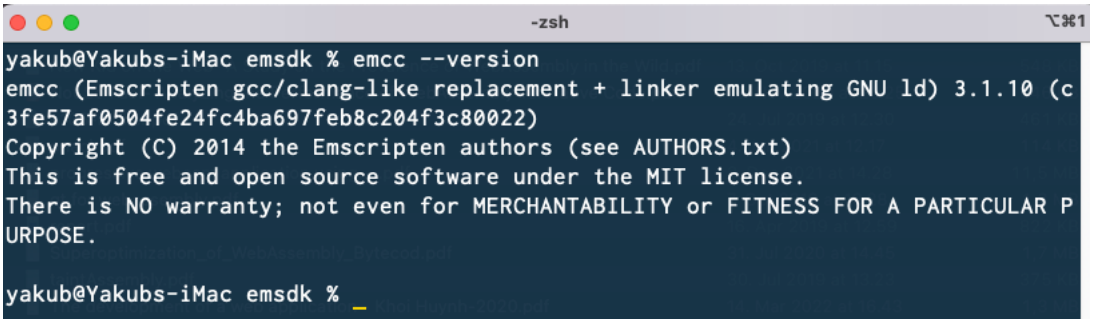
6. Bring the Emscripten toolchain into your path:
   source ./emsdk_env.sh

7. Test the above step (Picture 2):
   emcc -v



Picture 2. Environment setup.

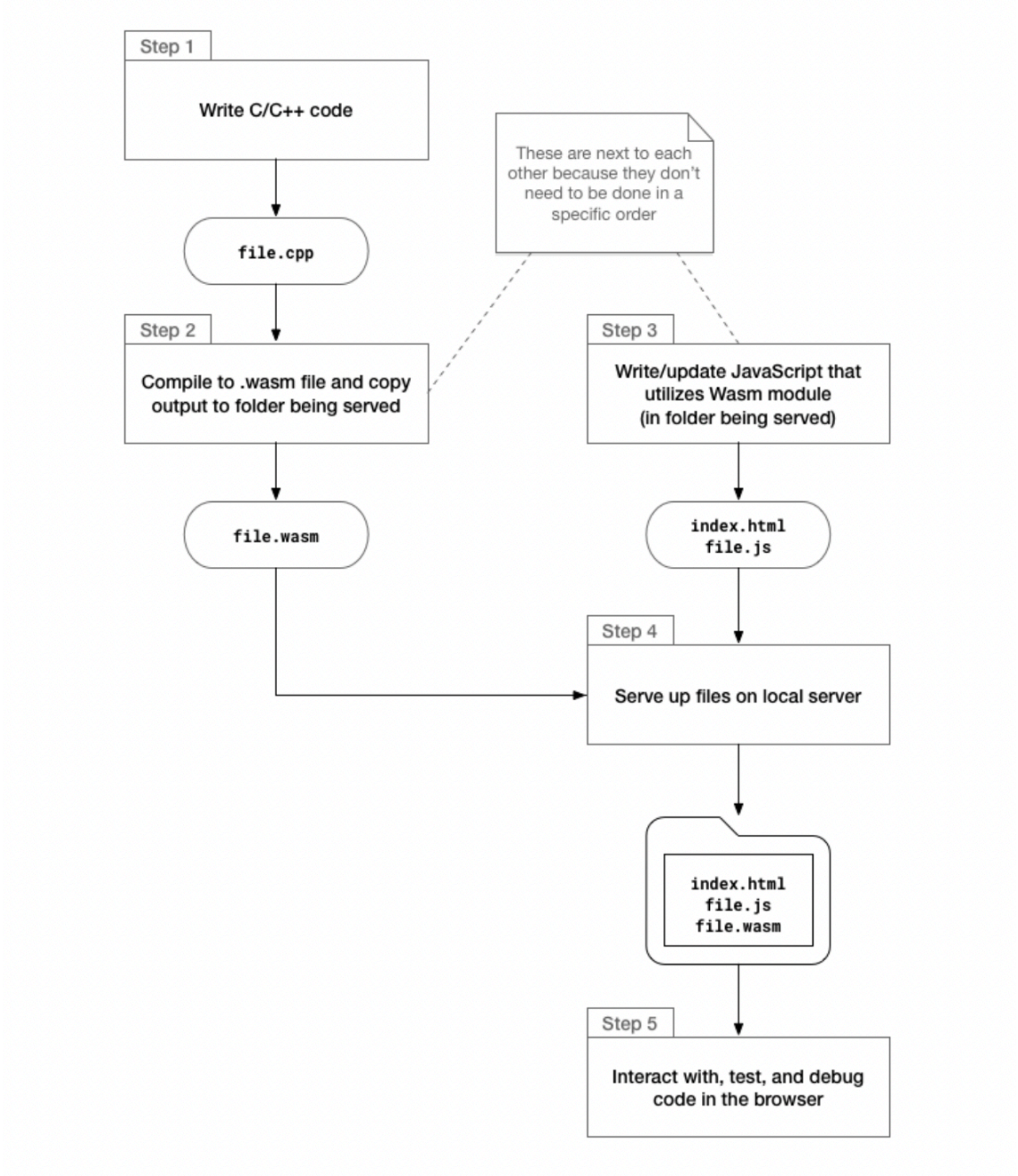
8. Check the version of Emscripten toolchain (Picture 3):
   emcc –version



Picture 3. Version of the Emscripten toolchain.

4.3 Development Workflow

For this pong game application, we will write the main function to control the computer player in C++ code and compile it down to the Wasm module, but the workflow (Picture 4) applies to other programming languages that compile to .wasm file.



Picture 4. Steps in the development workflow.

4.4 Calling C++ function from JavaScript

For this project, we will use a single C++ function called getAIMove (Picture 5) from JavaScript. For that, we are using a tool called Embind.

Embind: Embind is part of the Emscripten toolchain which allows C++ functions and classes to be called from JavaScript (Emscripten Contributors, 2015). We use **--bind** option when calling emcc to use embind while compiling.

```cpp
#include <emscripten.h>
#include <emscripten/bind.h>

enum Move {
    STATIONARY = 0,
    UP = 1,
    DOWN = 2
};


int getAIMove(float ball_xspeed, float ball_yspeed,
              float ball_xpos, float ball_ypos,
              int paddle_ypos) {
    int idealPosition = ball_ypos;

    if(ball_xspeed <= 0){
        auto turns = (ball_xpos - 50) / (-1 * ball_xspeed);
        idealPosition = ball_ypos + (ball_yspeed * turns);
    }

    if(idealPosition > paddle_ypos){
        return Move::DOWN;
    }
    if(idealPosition < paddle_ypos){
        return Move::UP;
    }
    return Move::STATIONARY;
}

EMSCRIPTEN_BINDINGS(pong) {
    emscripten::function("getAIMove", &getAIMove);
}
```

Picture 5. getAIMove function in C++.

The function file includes two headers emscripten.h and emscripten/bind.h which includes embind. Enumerator Move is defined with three constants, which hold the value for the computer player paddle. getAImove is the function to be exported which determines how a computer player should move its paddle. It takes the ball's x speed, y speed, x position, y position and paddle y position. These are floats and ints data types. Primitive data types were chosen compared to complex ones like classes and objects because they were easy to work with the bind.

getAIMove function will track the ball's position if the ball is moving away from the paddle. But if the ball moves towards the paddle i.e., ball_xpos is negative, it figures out the number of turns the ball will hit the paddle and calculates the ideal position. If the ideal position is greater than the paddle y position it will move down and if it is less than the paddle y position it will move up. The function is returning STATIONARY.

EMSCRIPTEN_BINDINGS is an Emscripten binding macro provided by Emscripten bind header which allows advertising the function which can be available to JavaScript. In Picture 5 from lines 30 to 32 we are saying that we want to create Emscripten function named "getAIMove" that we passed at the address of the function "getAIMove" which we have created above.

To make it work in a JavaScript file we have to use Module.getAIMove and pass the argument like shown in Picture 6. This is all needed to do in the JavaScript file.

```
93    const AIMove = Module.getAIMove(
94      ball.xspeed,
95      ball.yspeed,
96      ball.xpos,
97      ball.ypos,
98      leftPaddle.ypos
99    );
```

Picture 6. Calling code in JavaScript file that calls getAIMove function.

4.5 Changes in HTML

In the HTML file, we add the script, as shown in Picture 7. Where we are calling the render when the runtime has been initialized for WebAssembly. We have created a variable name Module and one of its properties is onRuntimeInitialized. When run time is initialized this function will get called and log in to the console after that render loop will execute. This way we can see webassembly has fully loaded before we do any invocation on it.

```
5     <script type="text/javascript">
6       let Module = {
7         onRuntimeInitialized: function () {
8           console.log('Module loaded: ', Module);
9           render();
10        },
11      };
12    </script>
```

Picture 7. Rendering in HTML file.

We are also pulling a file pong_wasm.js (Picture 8), which comes from Emscripten compile.

```
13      <script src="pong_wasm.js" type="text/javascript"></script>
```

Picture 8. Adding pong_wasm.js file.

4.6 Compiling

In the directory, there are three files pong.cpp, pong.html and pong.js (Pic 9). In the console, we write the following command.

emcc pong.cpp -o pong_wasm.js –bind

emcc is the command to compile. pong.cpp is the file we want to compile and pong_wasm.js is the name of the outcome file we want to be which we have included in the HTML file above (Picture 8). Finally, we pass the –bind flag, which is the one doing all the Emscripten bindings (Picture 9).

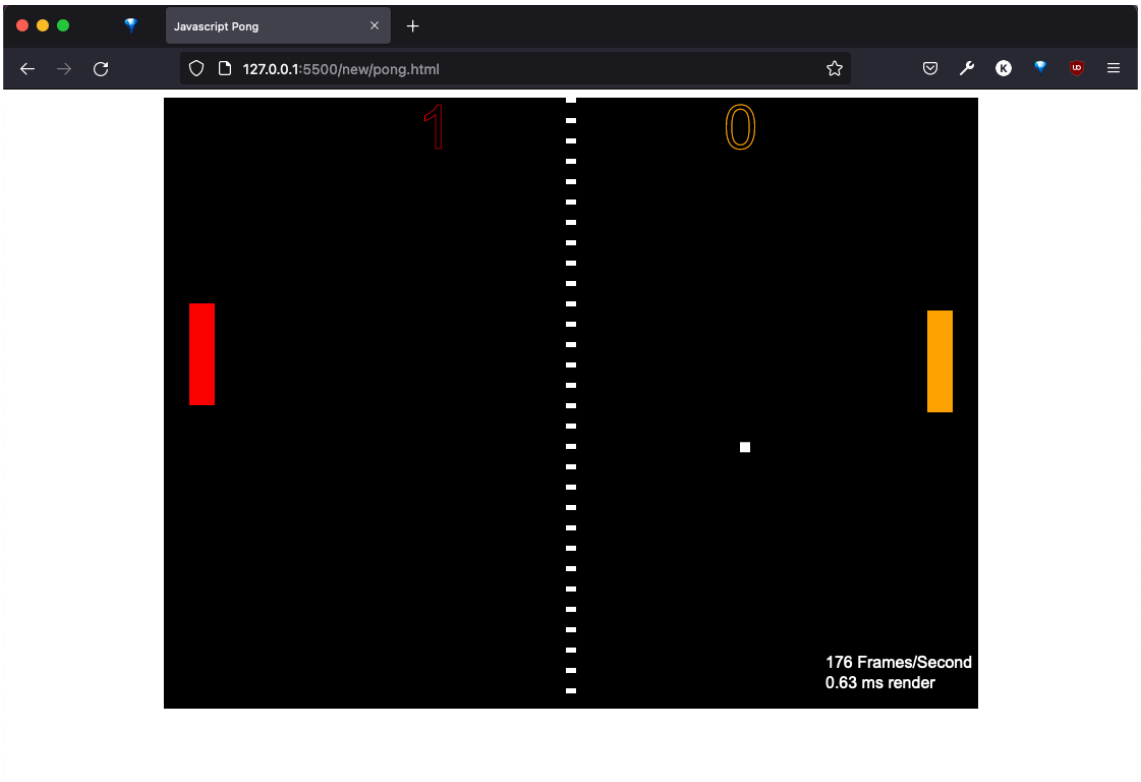
```
yakub@Yakubs-iMac new % ls
pong.cpp        pong.html       pong.js
yakub@Yakubs-iMac new % emcc pong.cpp -o pong_wasm.js --bind
yakub@Yakubs-iMac new % ls
pong.cpp        pong.js         pong_wasm.wasm
pong.html       pong_wasm.js
yakub@Yakubs-iMac new %
```

Picture 9. Commands to compile C++ file.

After compilation is completed two more files pong_wasm.js and pong_wasm.wasm files are added to the directory (Pic 9). The first compilation may take longer as Emscripten caches libraries for future use. Subsequent compilations will be faster.

4.7 Result

After opening the pong.html file, the browser will run a pong game (Picture 10). Where a player controls the orange paddle from the keyboard by pressing the up and down keys and the red paddle is controlled by a computer player through C++ in WebAssembly.



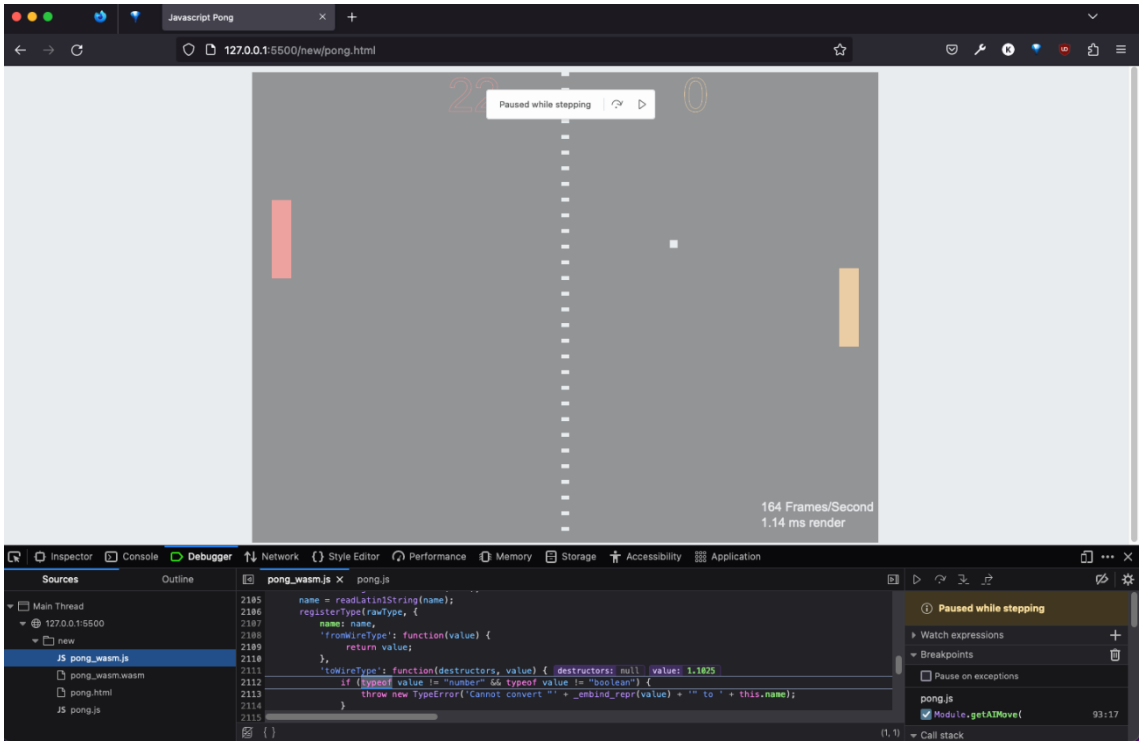
Picture 10. Pong game running in Browser.

To check if the WebAssembly code is working properly we can check in the dev console in the browser (Picture 11). To do so we can do the following steps

1. Go to the debugger
2. Go to the pong.js file in the source and put the breakpoint where the function getAIMove is called

3. Step-in in the breakpoint and it will lead to pong_wasm.js

This means WebAssembly is live, and we are calling C++ code from JavaScript.



Picture 11. Checking WebAssembly working in the debugger.

5 Conclusion

The goal of the thesis was to give an overview of WebAssembly and demonstrate its use case in web applications. To see the use case of WebAssembly, a simple pong game was developed. The game implements a C++ function for controlling the computer player's paddle. It was then compiled into WebAssembly and run in the browser through JavaScript. The successful execution of the game in the browser shows the ability to run the native code within a web application.

Concerning this thesis and its practical implementation, there are several ways of improving it. Firstly, the game could incorporate additional features to enhance its overall experience. Secondly, more advanced algorithms could be developed to enhance the gameplay. Lastly, User Experience could be improved to optimize user interaction with the game.

Overall, WebAssembly offers a promising solution for running high-performance native code on the web. It addresses the limitations of previous technologies and provides a standardized approach. As WebAssembly continues to evolve, it has the potential to revolutionize web development by enabling complex and performant applications that were previously challenging to implement in JavaScript alone.

References

Alladoum, C., 2018. Understanding WebAssembly An in-depth peek into the VM running in modern web browsers [WWW Document]. URL https://infocondb.org/con/shakacon/shakacon-x/web-disassembly-in-depth-peek-at-the-vm-running-inside-your-web-browser (accessed 5.31.23).

Clark, L., 2018. Making WebAssembly even faster: Firefox's new streaming and tiering compiler [WWW Document]. URL https://hacks.mozilla.org/2018/01/making-webassembly-even-faster-firefoxs-new-streaming-and-tiering-compiler/ (accessed 6.6.23).

Disselkoen, C., Renner, J., Watt, C., Garfinkel, T., Levy, A., Stefan, D., 2019. Position Paper, in: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy. ACM, New York, NY, USA, pp. 1–8. https://doi.org/10.1145/3337167.3337171

Donovan, A., Muth, R., Chen, B., Sehr, D., 2010. Portable native client executables. Google White Paper.

Emscripten, 2021. Emscripten documentation [WWW Document]. URL https://emscripten.org/docs/introducing_emscripten/about_emscripten.html (accessed 5.5.23).

Emscripten Contributors, 2015. Embind [WWW Document]. URL https://emscripten.org/docs/porting/connecting_cpp_and_javascript/embind.html (accessed 1.25.23).

Google Developers, 2021. Native Client and Portable Native Client documentation [WWW Document]. URL https://developer.chrome.com/docs/native-client/ (accessed 3.7.23).

Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.F., 2017. Bringing the web up to speed with WebAssembly. ACM SIGPLAN Notices 52, 185–200. https://doi.org/10.1145/3140587.3062363

Herman, D., Wagner, L., Zakai, A., 2014. asm.js specification [WWW Document]. URL http://asmjs.org/spec/latest/ (accessed 3.7.23).

Hoffmann, J., 2017. Flash And Its History On The Web [WWW Document]. URL https://thehistoryoftheweb.com/the-story-of-flash/ (accessed 5.29.23).

Mozilla, 2023. Understanding WebAssembly text format [WWW Document]. URL https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format (accessed 3.7.23).

Metz, C., 2011. Google Native Client: The web of the future - or the past? [WWW Document]. URL https://www.theregister.com/2011/09/12/google_native_client_from_all_sides/

Microsoft, 1996. Microsoft Announces ActiveX Technologies [WWW Document]. URL https://web.archive.org/web/20120201133255/http://www.microsoft.com/presspass/press/1996/mar96/activxpr.mspx (accessed 5.31.23).

Microsoft Windows Blogs, 2015. A break from the past, part 2: Saying goodbye to ActiveX, VBScript, attachEvent… [WWW Document]. URL https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/ (accessed 4.22.23).

Nelson, B., 2017. Goodbye PNaCl, Hello WebAssembly! [WWW Document]. URL https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html

Newstex, N.Y., 2010. Steve Jobs publishes some “thoughts on Flash”... many, many thoughts on Flash [WWW Document]. URL https://search.proquest.com/docview/189676933

W3C Team, 2015. WebAssembly Community Group [WWW Document]. URL https://www.w3.org/community/webassembly/ (accessed 5.20.23).

W3Techs, 2023. Usage statistics of JavaScript as client-side programming language on websites [WWW Document]. URL https://w3techs.com/technologies/details/cp-javascript (accessed 5.31.23).

Wagner, L., 2017. Turbocharging the web. IEEE Spectr 54, 48–53. https://doi.org/10.1109/MSPEC.2017.8118483

Watt, C., 2018. Mechanising and verifying the WebAssembly specification, in: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, New York, NY, USA, pp. 53–65. https://doi.org/10.1145/3167082

WebAssembly, 2023. WebAssembly [WWW Document]. URL https://webassembly.org/ (accessed 4.7.23).

Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K., 2016. Comprehensive Analysis and Detection of Flash-Based Malware, in: Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science. Springer International Publishing, Cham, pp. 101–121. https://doi.org/10.1007/978-3-319-40667-1_6

Zakai, A., 2011. Emscripten, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion - SPLASH ’11, OOPSLA ’11. ACM Press, New York, New York, USA, p. 301. https://doi.org/10.1145/2048147.2048224

Link to project source code

https://github.com/yakubkapri/pong-wasm