Compiling SDL Multiplayer Games to WebAssembly with Emscripten for use in Web Browsers

Kompilera SDL multiplayer spel till WebAssembly med Emscripten för användning i webbläsare

Oscar Falkmer
Martin Norrman

Examensarbete inom
Datateknik
Grundnivå, 15 hp
Handledare på KTH: Jonas Willén
Examinator: Ibrahim Orhan
TRITA-CBH-GRU-2022:054

KTH
Skolan för kemi, bioteknologi och hälsa
141 52 Huddinge, Sverige

Sammanfattning

Det kan vara svårt att samla in och distribuera onlinespel gjorda av oerfarna utvecklare. Detta är något som KTH (Kungliga Tekniska Högskolan) har problem med i en kurs som involverar SDL och SDL_Net programmering. En bra lösning på detta problem är att köra dessa spel på en webbsida. Ett lättanvänt sätt att kompilera och distribuera multiplayer-spel och spelservrar skrivna i C till webbapplikationer och webbservrar behövdes. Specifikt för spel skrivna i C med SDL och SDL_net biblioteken. Kompileringsverktyget Emscripten användes för att kompilera spel- och serverkod från C till WebAssembly, som sedan kunde användas genom de genererade JavaScript-funktionerna. Kommunikationen mellan klienten och servern sköttes av WebSockets. I största möjliga mån skulle Emscripten specifika funktioner döljas bakom C-bibliotek som emulerade formatet av SDL_Net. De färdiga lösningarna som emulerar formatet av SDL_Net bestod av två nya bibliotek, ett för servern och det andra för klienten. De emulerade framgångsrikt TCP-delarna av SDL_Net biblioteket. Webbläsarens händelseschemaläggare kräver att applikationer har möjligheten att återge kontroll till den. Detta gjorde att spelkodens oändligt loopande funktioner behövdes skrivas om för att kunna rendera i webbläsaren.

Nyckelord
Webassembly, Wasm, Emscripten, C, webbaplikation, webbserver, WebSocket, SDL, SDL_Net, kompilera.

Abstract

Collecting and deploying online games made by inexperienced developers can be hard. This is something KTH (Royal Institute of Technology) has a problem with pertaining to a course involving SDL and SDL_Net programming. A good solution to this problem is to host these games on a website. An easy-to-use way of compiling and deploying multiplayer games and game-servers written in C as web applications and web servers was needed. Specifically for games written in C using SDL and SDL_net libraries. The compiler toolchain Emscripten was used to compile game and server code from C to WebAssembly, that could then be used through the generated JavaScript functions. Communication between the client and the server was handled by WebSockets. As much of the Emscripten specific functions were to be hidden behind C libraries, emulating the format of SDL_Net. The finished solutions that emulated the format of SDL_Net, consisted of two new libraries, one for the server and the other for the client. The libraries successfully emulated the TCP parts of SDL_Net library. The browsers event scheduler necessitates applications to be able to return control back to it. This meant that the game codes endlessly looping functions had to be rewritten to enable rendering in the browser.

Keywords
Webassembly, WASM, Emscripten, C, web application, web server, WebSocket, SDL, SDL_Net, compiling.

Table of Contents

1 Introduction

This chapter contains the background to this thesis, the problem it sets out to solve and the goals set to be accomplished, as well as the limitations set in place.

1.1 Background

Students at KTH Royal Institute of Technology attending the course HI1038 “Projektkurs inom data- och nätverksteknik”, are tasked with making a multiplayer game capable of handling at least 4 simultaneous players. The programming language used to make these games is C, with the SDL and SDL_net libraries. The students usually have limited programming knowledge at the time of taking the course, so the games that are created usually end up tightly bound to the machines the students developed them on. This makes the games hard to share with others. Because of this, the course coordinator of HI1038 wanted a simple way of gathering these games in one place and being able to, as painlessly as possible, get them running and playable.

1.2 Problem Statement

An ideal way of gathering and making games easily available would be by hosting them on a website. Turning the games created in the HI1038 course into web applications would make the games a lot easier to share with others as a web application is accessed through a browser, which is used by a wide range of devices and operating systems.

While there are existing methods for compiling code written in C into a web application using WebAssembly, the problem arises as a result of the multiplayer aspect of the games. To get multiplayer functionality in a game compiled to run as a web application, developers must have prior experience in the area of web development and network communication. The students are not expected to have any knowledge about web development at this stage of their education. What this study aims to do is to try to develop an easy-to-use way of converting these games written in C with SDL and SDL_Net. into multiplayer web applications, with an easy and simple deployment method that doesn't require extensive prior knowledge in the area.

1.3 Goals

The goal of this thesis is to develop an easy-to-use way of deploying a game written in the C programming language as a web application, capable of handling multiple concurrent players on the same game instance. The deployment method should be simple enough to understand and use by people with limited programming knowledge and no prior experience of web application development. This method should work on the operating systems Windows, MacOS, and Linux. A simple prototype game is to be created using the method to test its functionality.

1.4 Limitations

This thesis is limited to a ten-week work period with two people contributing. It will not contain a complete explanation of the Emscripten toolchain and its functionality as it is too big to be covered in its entirety.

2 Theory and Background

This chapter covers the necessary underlying theories and technologies used in this thesis as well as prior works (2.1) that were related to the stated goals. The SDL-libraries, WebAssembly and Emscripten are explained in section 2.2, 2.3 and 2.4 respectively as they are central to this work. Furthermore, in section 2.5 comes an explanation of web servers as they are important to enable web browser hosting, specifically Node.js as a web server solution. Section 2.6 describes network protocols used in this study for data transferring between client and server.

2.1 Prior Works

There exist several projects that use WebAssembly and Emscripten for porting C code to the browser. Most of these applications do not need networking support. For example, a port to desktop browsers made by James Mackenzie of the classic game “Commander Keen” [1], originally created by id software. Another example is a simple snake game written in C by John Sharp as part of a tutorial on Emscripten [2]. Both of these games were created by generating WebAssembly code from the original source code written in C and C++. The website webassemblygames.com also provides examples of games using WebAssembly [3].

An example of previous work with network interactions written in C and compiled with Emscripten is “Monster Madness Online” by Trendy Entertainment & Nom Nom Games [4]. Since the article was written in 2013 and WebAssembly had yet to be announced, Monster Madness Online used asm.js instead of WebAssembly. WebAssembly has more or less rendered asm.js obsolete and has become the new standard for running C code in the browser and is up to 20 times faster than JavaScript [5]. In the article about Monster madness the process of compiling a game using Unreal Engine 3 to a web browser version of that game is described. They write about trying to utilize WebRTC as the networking solution before settling on using WebSockets. WebRTC is a peer-to-peer API that provides browser-to-browser communication, something that was not the desired architecture for Monster Madness Online.

No previous work was found during the literature study pertaining to compiling SDL_Net code to run in the browser. There existed however a port that could be used by the Emscripten compiler, more about this in section 3.3.1.

2.2 Simple Direct Media Layer

Simple DirectMedia Layer or SDL for short, is an open-source software development library written in C under the zlib License [6]. SDL provides low level access to audio, video and input devices such as keyboard and mouse. SDL supports 2D graphics and via OpenGL-, Direct3D-, Vulkan- or Metal-API also supports 3D graphics [7]. In addition, the SDL library is also capable of File input/output abstraction, threads, timers, CPU feature detection and power management [7].

The library is cross platform with support for Windows, Mac OS X, Linux, iOS, and Android [7]. Natively SDL supports the programming languages C and C++. It also has so called bindings for other languages including but not limited to C#, Python, Go and Rust [8]. A language binding in programming and software development is an API that allows a programming language to use a foreign library. All this combined makes it possible to write code on one operating system and compile it on another with very little change to the actual code.

Figure 1 shows a simple SDL game where you can move a rectangle up and down. The code for this program is included in appendix 3.

Figure 1: Simple SDL game window showing a red bar that can move up and down.

2.2.1 SDL_Net

In addition to basic SDL, there are also separate official libraries that extend SDL to provide additional functions. The most commonly used libraries are SDL_Image (images), SDL_mixer (audio), SDL_ttf (fonts), SDL_Net (network), SDL_rtf (RTF documents). There is one of these official extra libraries that is of significance to this study, which is SDL_Net. It is a networking library that simplifies connection handling and data transfering [9]. It has cross platform support, which means a programmer doesn't have to code separate solutions depending on platform. It uses socket connections for both TCP and UDP to provide support for client-server solutions.

2.3 WebAssembly

Webassembly also known as Wasm was first released in 2017 with the aim to improve performance of web applications that were traditionally written in JavaScript (JS). To do this, programs written in WebAssembly are delivered in pre-compiled binary and can therefore be loaded faster than JavaScript which has to be compiled at runtime. WebAssembly programs usually don't consist of handwritten code like JavaScript programs but are instead generated from programs written in higher level languages like C, C++ or Rust [10]. WebAssembly is currently supported by most major web browsers which also makes it possible to be used on all major operating systems.

The way WebAssembly code is generated from higher level languages is by using WebAssembly compilers like Emscripten and Cheerp or other tools like Blazor or SwiftWasm, depending on what higher level language is to be converted. It provides safe, fast and portable low-level code that can be executed in the browser [11].

Even though WebAssembly is a binary language it is presented with syntax that makes it easier for a human to understand. It has modules that define functions, globals, tables and memory. These modules can be both imported and exported. An instance of a module is equal to a dynamic representation of a program and its operations are provided by an embedder, such as an operating system or virtual machine. When making an instance of a module all imports must have a definition [11].

Each module consists of functions that can take parameter values and return a value of its defined function type. Functions can call each other recursively but can't be nested within one and other. Functions consist in turn of instructions that use an operand stack to manipulate values. Some instructions may initiate traps that will stop the current execution and initiate a JavaScript exception to be thrown [11][12].

WebAssembly uses linear memory which consists of a large byte array. A module can define a single memory and share it with other modules via import or export. The memory size is dynamic to fit the modules needed. It is also disjoint from code space, stack and engine data structures which limits the program to only affecting its own memory and not the environment it runs in, thus making the program more secure to run [11].

2.4 Emscripten

Emscripten is an open source WebAssembly toolchain that can be used to compile C and C++ code into Wasm. It supports most portable C/C++ code and has a clear niche for game code conversion.

2.4.1 Emscripten Compiler and Library Ports

Emscripten provides, through its SDK, a compiler known as emcc that works as a drop-in replacement of a standard C compiler, like gcc or clang. Emcc utilizes LLVMs backend stage to compile programs from C and C++ into WebAssembly and JavaScript [13].

When compiling with emcc two files are generated based on the C code. The first is the Wasm file that contains the generated functions. The second one is a JavaScript glue file that makes it possible to call the functions in the Wasm file from JavaScript files [14]. Besides these two files the option is given to generate a HTML page for testing of the generated code [14][15]. This page can be accessed through a local file server and will look as shown in figure 2.

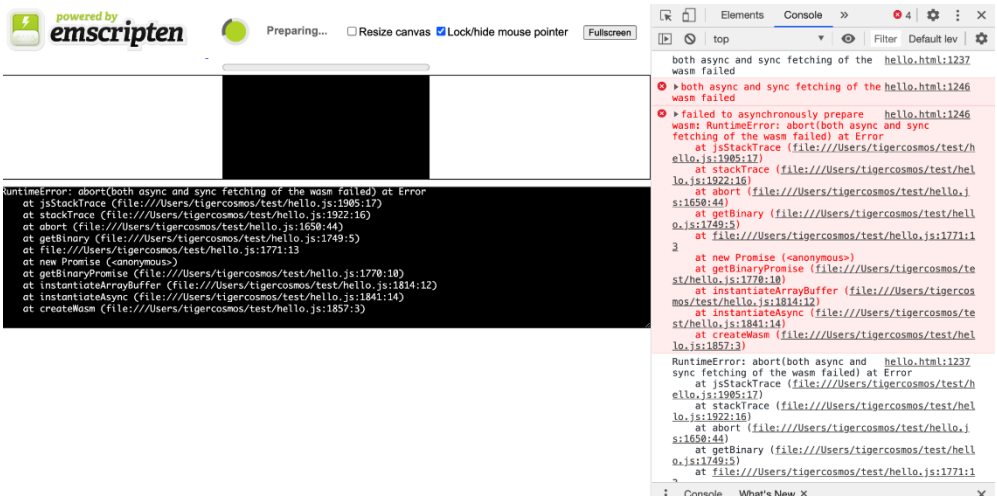

Figure 2: Emscripten generated HTML page running in a browser.

To complement the built-in libraries in Emscripten there exist several extra libraries that have been ported to Emscripten by the Emscripten community [16]. These ports reside on a GitHub and can be loaded when requested by emcc, shown in figure 3. After a library is fetched it is built and linked

with the project [16]. All available ports can be seen by running “emcc –show-ports”. Existing libraries of relevance to this thesis are the SDL2 ports and the SDL2_Net, SDL2_image, SDL2_mixer, SDL2_ttf ports [17].

```
emcc foo.c -sUSE_SDL=2 -o game.js
```

Figure 3: Compiling the C file foo.c with SDL2 port.

2.4.2 JavaScript to Wasm Calls

When not using the Emscripten provided HTML file, other means of calling functions in Wasm from JS are needed. Emscripten has two functions, ‘ccall’ and ‘cwrap’, that fill this role in slightly different ways [18]. ‘ccall’ makes a call to a specific function with parameters and returns the result. ‘cwrap’ on the other hand wraps the function and returns it as a callable JavaScript function. This is useful when wanting to call the function several times [18]. Figure 4 shows usage of ‘ccall’ and ‘cwrap’.

```
// name of C function, return type, [argument types], arguments
var result = Module.ccall('int_sqrt', 'number', ['number'], [28]);
int_sqrt = Module.cwrap('int_sqrt', 'number', ['number'])
int_sqrt(28)
```

Figure 4: Use of ‘ccall’ and ‘cwrap’ to get the square root of 28.

To enable the usage of ‘ccall’ and ‘cwrap’ they have to be declared as exported runtime methods when compiling. In a similar manner the functions to be used by ‘ccall’ or ‘cwrap’ need to be declared as exported functions. When declaring a function as exported the name of the C function must be prepended with an underscore [18]. Export of functions and runtime methods can be seen in figure 5.

```
-s EXPORTED_FUNCTIONS='["_foo", "_bar"]'
-s EXPORTED_RUNTIME_METHODS='["ccall", "cwrap"]'
```

Figure 5: Declaring exported functions and exported runtime methods when compiling.

2.4.3 C & C++ to JavaScript Calls

Emscripten also provides several different means of communicating out of the C or C++ code into JavaScript [19]. A simple but somewhat slow function is ‘emscripten_run_script’ that enables calls to JS functions [43]. Emscripten supplies ways of directly writing JS code inside the C or C++ file. Two functions, ‘EM_JS’ and ‘EM_ASM’, can be used for this purpose and are both faster than ‘emscripten_run_script’ [20]. The use of these three functions can be seen in figure 6.

‘EM_JS’ is used to declare a function that can later be called within the C code just like any other function in C [21]. In the declaration of the function the return type, function name to be called and the function parameters are specified, as well as the JavaScript code itself. The return type and parameters make use of C data types.

'EM_ASM' is used to run the written JS code directly inside another function and is similar to 'emscripten_run_script' [21]. Plain 'EM_ASM' lacks return values, but this is mitigated by two other versions. 'EM_ASM_INT' and 'EM_ASM_DOUBLE' can return values but require the need of casting to translate the returned values into the requested data type.

```
EM_JS(void, printNum, (int a),{
  console.log("Number: " + a);
});

void foo(int a){
  printNum(a);
  emscripten_run_script("console.log('Number: ' + a);");
  EM_ASM({ console.log("Number: " + a); }, a);
  //If a = 5 all three will write "Number: 5" to console.
}
```

Figure 6: Usage of 'EM_JS', 'emscripten_run_script' and 'EM_ASM' to print the text "Number: 5" to the console.

When using functions like 'EM_JS' and 'EM_ASM_INT' with strings as return type or as parameters, conversion of the string is needed [22]. Conversion between C strings and JS strings can be done with the use of the Emscripten 'UTF8ToString' and 'StringToUTF8' functions. When returning a string from JS to C, memory needs to be allocated for a new string which will be a converted copy of the JS string. Emscripten enables this with two functions. 'lenghthBytesUTF8' that calculates the JS string length in UTF-8, and '_malloc' that is used to allocate the needed memory on the Wasm heap.

2.4.4 Emscripten Main Loop

Games written in C typically have a main loop that is constantly running and updating the game state and images shown on screen. This program structure was problematic for the browser, as the process schedulers are using a cooperative multitasking model. This means each event has to give control back to the scheduler for the next process to run [23]. Thus, an infinite loop in a program will never give back control and locks up the browser.

```
emscripten_set_main_loop_arg(loop_handler, &ctx, -1, 1);
```

Figure 7: Usage of 'emscripten_set_main_loop_arg' to start an infinite loop with the function 'loop_handler'.

To get around this problem Emscripten has provided a set of functions that emulates a main loop called 'emscripten_set_main_loop' and 'emscripten_set_main_loop_arg'. These call a given function that works just like the regular main loop. This main loop is run a specified number of times per second, while also giving up control to the scheduler periodically. The functions work largely the same but differ slightly in the arguments passed. Both take a pointer to the function that will act as the main loop, an integer for the desired frames per second and an integer to decide if it should act as an infinite loop or not [24]. The 'emscripten_set_main_loop_arg' function also takes a void pointer used to pass user-defined data to the function acting as the main loop. Figure 7 shows the usage of 'emscripten_set_main_loop_arg'.

2.4.5 Emscripten Network Support

Emscripten provides several different ways of supporting network communication to a server [48]. The Emscripten WebSocket API is a C/C++ library that enables WebSocket communication. It utilizes callback functions to handle incoming messages and is capable of sending data in binary and UTF-8 text format [25].

For conversion of existing POSIX TCP sockets there exists a non-completed solution that emulates the POSIX socket over a WebSocket. Besides the limitation of only using TCP there is also the problem with its implementation as Emscriptens documentation highlights that an out of the box implementation will run into problems [25].

A third method is the usage of a WebSocket proxy server that allows full POSIX socket API access from the browser. Calls from the POSIX socket will pass via WebSockets to the proxy server that in turn handles the native TCP and UDP calls. This solution is recommended for testing infrastructure and debugging as it is somewhat slow [25].

By default, the browser does not support direct UDP communication. Technologies like netcode.io and WebRTC can however provide UDP and UDP-like communication [25][26]. Currently Emscripten does not provide any C or C++ API for interacting with any of these [25].

2.4.6 Emscripten Pthreads

Emscripten supports multithreading in browsers though the use of SharedArrayBuffer. This enables sharing of memory between the main thread and worker threads. It also provides synchronization that enables Pthreads (POSIX threads) API [44].

2.4.7 Hello World in Emscripten

The simple SDL game shown in chapter 2.2 was rewritten to be compiled by Emscripten. The port flags used when compiling to the HTML file were '-O2' and '-s USE_SDL=2'. Figure 8 shows a screenshot from how a chrome browser looked when the resulting HTML file was hosted on a python file server. Code for this version of the game is included in appendix 4.

Figure 8: Simple SDL game compiled with Emscripten to run in a browser.

2.5 Web Servers

Web servers can be categorized as static or dynamic depending on what service they provide. A static web server provides only its hosted files, whilst a dynamic server also provides extra software in the

form of an application server or database communication [27]. The web server's main goal is to handle requests made by the client. These requests can be for data, execution of a computation or interaction with a database. If no data exists, an error occurred or if the user has provided incorrect credentials, an error message will be returned to the client. The returned data or error will be handled and displayed by the client browser [27].

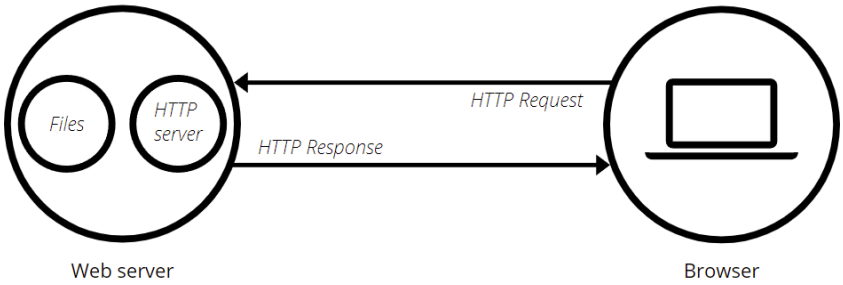

Figure 9: Web server and browser communicating via HTTP [27].

2.5.1 Web Applications

A Web application shares many similarities with a normal website. The main difference is the web applications focus on interactivity, authentication, and integration of different components [28]. The basis for this is the concept known as Software as a Service, SaaS, and the goal of a web application is to provide software to the client without the need of having to download and install it on the client computer [29][30].

Instead, the software is run in the browser which has several benefits. The software is more accessible as the browser eliminates the need to make different versions for different operating systems and the need for specific hardware. Updates and maintenance don't need to be handled by the client and are instead done by the application provider [29]. There is also an increase in security as the software is harder to copy and subject to piracy. These features come with the downside that is the constant need for an internet connection and the risks that come with having resources exposed to the internet [29][30].

2.5.2 Node.js

Node.js is a free and open-source cross platform JavaScript runtime environment using the V8 JavaScript engine. Thus, it is not a web server per say but can be used to create web servers using modules like the HTTP module or frameworks like Express.js amongst many others. Node runs all processes in a single thread and I/O is handled asynchronously to avoid blocking of the thread. This allows node to avoid the burden of managing thread concurrency [31].

The non-blocking single thread solution is a distinct aspect of Node.js. This is not to say that blocking methods don't exist, they do, but are distinguished by their names ending in "Sync" [32]. Node also has the potential to use more than one thread for specific tasks, but the thread pool, also known as the work pool, is not very big. Thus, the usage of the work threads should be handled with care as to not block them in vain [33]. It is also important to note that CPU heavy operations that don't perform I/O aren't considered to be blocking, which can be a problem if the threads become too busy to handle new requests [32].

An important note is that error handling in Node.js is implemented differently for synchronous and asynchronous operations. A synchronous operation will throw an error and will crash if the error is

not caught, whilst with an asynchronous operation it is up to the programmer to decide if an error should be thrown or not in the first place [32].

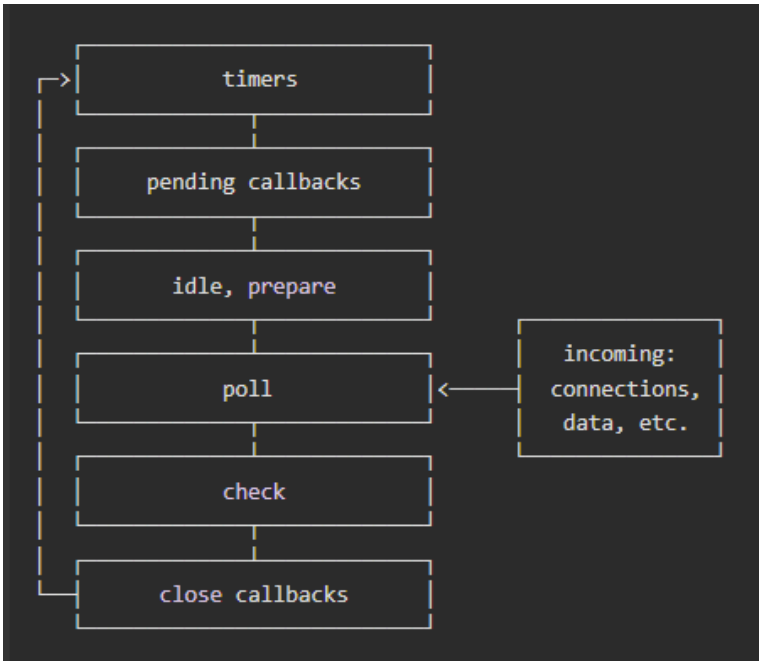

Figure 10: Node loop phases.

Node's ability to perform non-blocking I/O operations is based on its event looper [34]. The loop is initiated at Node.js startup and loops through a series of phases as depicted in figure 10. Each phase has its own queue of callback executions following the principle of first in first out. When in a phase operation specific to that phase are performed and callbacks from the queue are performed until the queue is either empty or a maximum number of callbacks have been reached. The loop will then enter the next phase [34].

2.6 Communication

Communication over the net can be done in a large variety of ways through many different protocols. The following segments will handle HTTP and the WebSocket API.

2.6.1 HTTP (Hypertext Transfer Protocol)

The way a browser communicates with a web server is through the use of HTTP, which is an application-level protocol for transporting hypermedia documents. HTTP relies on TCP and before HTTP communication can begin a TCP connection must be established [35]. HTTP is stateless and the server does not remember any data between requests [36]. This can be mitigated by sending cookies with session data between the browser and the server allowing for requests to happen within the same context [35].

The HTTP headers are an important part of the communication capabilities of HTTP and are often grouped by context into requests, response, representation, or payload headers. The purpose of the headers are to allow for additional information about the client, the resource or other metadata about the message or the underlying network. The headers are composed by a name and a value separated by a colon [37].

Cross-Origin Resource Sharing, also known as CORS, is based on the HTTP headers and allows the server to know the origin of a message and if it is to be trusted with resources. A preflight request is made by the browser to the server to check if it will be allowed access to a resource. In the preflight request indications of the method and headers that the real request will have, are appended. The server evaluates the preflight request and returns a message either allowing or denying the real request [38]. In the case of a request being declined the infamous CORS error is returned. The reason for the error will not be revealed in detail to the browser out of server security concerns [39].

2.6.2 WebSockets

WebSockets consist of the WebSocket protocol and the WebSocketAPI and is used to communicate between two endpoints known as sockets. This enables two-way communication between a browser and a server. WebSockets is based on TCP and eliminates the need for multiple HTTP connections such as constant polling, XMLHttpRequest or iframes [40].

WebSockets provide full duplex communication and thus allow for data to be sent both ways simultaneously. A websocket connection is established by first using a HTTP request/response to connect the client to the server. The HTTP header is upgraded to switch from a HTTP connection to a WebSocket [41][42]. The new WebSocket is established through a handshake over TCP. The handshake provides both parties with necessary information about each other to establish communication. Except for in this connection establishing phase, WebSockets do not utilize the request and response architecture. After a connection is established, it is left open for communication that can be initiated or terminated from both sides [41][42][43].

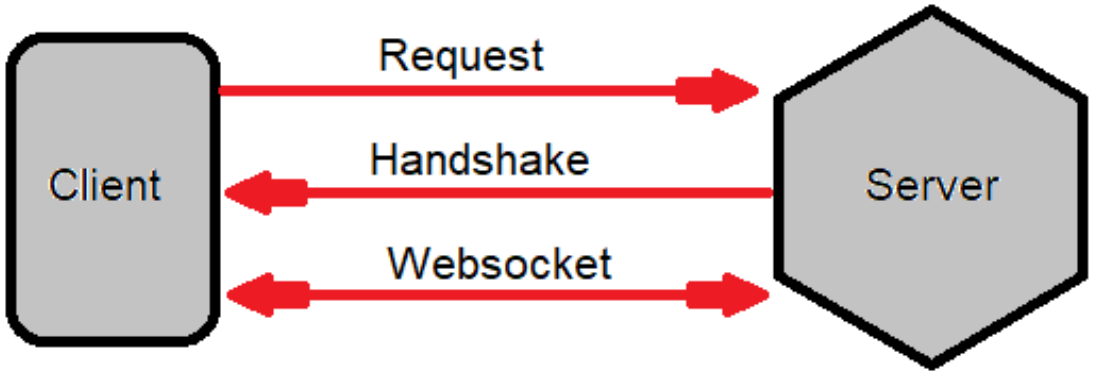

Figure 11: Shows the connection phase with HTTP, handshake and final WebSocket connection.

When using the WebSocket protocol, uniform resource identifiers are needed with the “ws:" or “wss:” scheme. Similarities can be drawn to HTTP URLs that will use the “http:” and “https:” scheme [42].

WebSockets can be used for a wide array of different purposes. Some popular usages are in real time web applications, games, chat clients, news tickers and stock tickers [41]. Despite this there are still some areas where you might want to use HTTP instead. These use cases consist of when you want to get data only once, don’t want or need constant updates or if you want to get older data [43].

3 Methodology and Results

In this chapter the academic methodology (3.1) and the tools and frameworks (3.2) that were implemented in the solution will be presented. This is followed by a brief description of two early attempt in section 3.3. The final methodology is described in section 3.4 and finally the result of said methodology can be found in section 3.5.

3.1 Academic Methodology

An initial literature study was conducted to gather information of possible tools and frameworks to be used. To find information on the subject preliminary searches after similar works involving Emscripten compiled games with multiplayer were done. These results gave information to what areas were of interest for the study to cover and worked as a baseline for the following searches with the intent of finding credible sources that could be used, such as white papers, scientific articles, and other thesis. For a list of search engines and search terms, see appendix 5. Based on this study a method was formulated and a prototype game was used to test it. The functionality of the solution was then tested and evaluated. Tests can be found in appendix 6. If errors or unsatisfactory behavior arose, additional research was made on the topic to formulate a solution to the issue. The literature study was then updated with the new information and information that had over time proved to be irrelevant was removed.

3.2 Tools and Frameworks

As the original game code was written in the C programming language with SDL2. The Webassembly compilation toolchain Emscripten was chosen as it had support for SDL2 libraries through easy-to-use ports when compiling. Emscripten also provided a large amount of documentation and functionality for communicating between JavaScript and WebAssembly files. From the prior works listed in section 2.1 Emscripten seemed like a well-suited option when compiling games to be used in browsers.

On the client side a simple file server was used and on the server side Node.js was used to host the game server. A file server and Node.js were simple to use and were powerful enough to suit the needs of this thesis without requiring a large amount of time setting up the servers. The goal of the project was to keep it as simple as possible, and these choices reflected that.

3.3 Early Attempts

Two unsuccessful methods for implementing network communications were tested prior to the construction of the final method. Both of these earlier attempts acted as steppingstones towards our solution presented in section 3.4.

3.3.1 Compile SDL_Net

An initial attempt was made at compiling code that straight up implemented the SDL_Net library. Emscripten provided a library port for SDL_Net in the same way that it provided ports for the other SDL libraries. A simple TCP echo server was created using SDL_Net which was then compiled with the Emscripten compiler. The Wasm code was then run with Node.js using ‘ccall’ as mentioned in section 2.3.1. No issues arose when compiling but when running the server it generated an error, “unsupported syscall setsockopt”. Solving the problem of getting the SDL_Net socket to function with Emscripten was deemed to take more time than what was available, so another method had to be used. More about this will be discussed in section 4.1.2.

3.3.2 Non Emscripten WebSocket

The goal of the second attempt was to implement WebSockets outside of the Wasm code and instead make use of the flexibility of the multitude of JavaScript networking libraries. As can be seen in section 3.4, WebSockets became a part of the end solution, and this first attempt with WebSockets greatly affected the way the solution would look in the end.

A simple WebSocket echo server was implemented with Node.js using the 'ws' library. No Wasm code was implemented on the server side as it simply would echo back what the client sent. The client on the other hand did use Wasm code that was implemented and run from HTML in the same way as in the end solution, see section 3.4.2.1 for more detail. The client made use of the 'EM_JS' functions, explained in section 2.4.3, to make calls to a JS file that implemented the 'ws' library. Problems arose as the JS file being called to by Wasm on the client side was unable to load or recognize the 'ws' library, resulting in crashes. This problem led to the use of the Emscripten WebSocket library on the client side, see section 3.4.2.3. Discussion of the type of solution this second attempt tried to utilize can be found in section 4.1.2.

3.4 Solution

Here the final method will be presented for converting SDL_Net games. This method was created based on the experiences from the earlier attempts. First there will be a broader overview (3.4.1) that presents the general code flow on both the client and server side. This will be followed by a more in-depth explanation of the client method (3.4.2) and server method (3.4.3) separately. And finally, a look at the use of the Makefile (3.4.4) on both client and server side. The complete documentation of the SDL_Net_B and SDL_Net_BS library can be found in appendix 1 and 2.

3.4.1 Overview

The main goal of this thesis was to create a method for deploying a multiplayer game, written in C using SDL libraries and SDL_Net, to the browser. The games server was to be deployed as a web server.

The overall conversion of the C code, both on the client and server side, was done through the use of the Emscripten compiler. Emscripten compiled the original C code into a Wasm file and a JavaScript file that acted as glue code. For Emscripten to make a functional game that could run in the browser and communicate with the server, some changes needed to be done to the client and server code. These changes mostly involved the use of the emscripten provided C libraries that allowed for looping of the game code without blocking other functionality.

To handle communication between the client and the server WebSockets were used. Emscripten provided a library for easy handling of WebSockets that could be used by the client whilst the servercode used the 'ws' library running on Node.js. To emulate the SDL_Net library while still using WebSockets, two separate libraries were created. SDL_Net_B was made to abstract the WebSockets on the client side, while SDL_Net_BS was used on the server side. The need for a client and server specific library was apparent due to the large differences in WebSocket implementation, which is explained in section 3.4.2 and 3.4.3.

As WebSockets are asynchronous whilst SDL_Net is not, some sort of in-between storage was required for incoming communication to emulate the polling nature of certain SDL_Net functions.

The following figure (12) gives a broad overview of the general coad flow and structure of the client and server, as well as to how the SDL_Net_B and SDL_Net_BS libraries fit into the code.

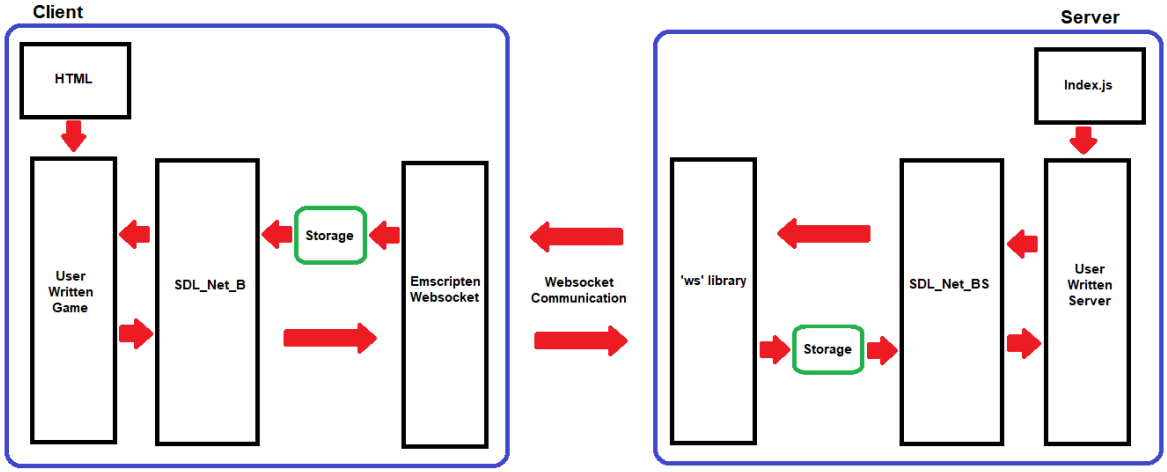

Figure 12: A broad overview of the code flow in the solution. Index and HTML are the starting points for the server and client respectively.

3.4.2 Client-side

This part will, in the order given, focus on how the Wasm game code was implemented in the HTML file, what modifications were used to get the game to function and render in the browser environment, and finally a rundown of the SDL_Net_B library. For information concerning the Makefile see section

3.4.2.1 Implementation in HTML

As mentioned in section 2.4.1 the Emscripten compiler can generate a HTML file along with the Wasm file and its JavaScript glue file. But due to the lack of user friendliness of said HTML file, a separate HTML file largely based on one created by John Sharp [10] was used. Thus, only the Wasm file and the JS glue file were created with the emscripten compiler. The client code was loaded and appended via the JS glue file to the HTML, see figure below.

```
var script = document.createElement('script');
script.src = "game.js";
document.body.appendChild(script);
```

Figure 13: Importing the JavaScript glue code to be used in HTML.

To enable the game to show up on screen a global 'Module' object was created at startup with a 'canvas' function that returned a canvas element defined in the HTML. This can be seen in figure 14. The 'Module' object was used by the Wasm code to get access to the canvas element and render the game at different points in its execution.

```
var Module = {
    canvas: (function() {
        var canvas = document.getElementById('canvas');
        return canvas;
        })()
};
```

Figure 14: Module object with function returning HTML canvas element.

As shown in figure 15, the game was started by calling the ‘mainf’ function with the use of the ‘ccall’ function mentioned in section 2.4.2. The ‘mainf’ function was intended as a hook for starting the game code in Wasm. To avoid editing in the JS file, all games to be run had to have a function called ‘mainf’ that returns void and takes no arguments. How ‘mainf’ works and its role in the game code will be the subject of section 3.4.2.2.

```
Module.ccall('mainf', null, null);
```

Figure 15: The use of ‘ccall’ to call the ‘mainf’ function to start the game.

3.4.2.2 Mainf() and Emscripten Main Loop

As mentioned in the previous segment the ‘mainf’ function was used to simplify the deployment of the Wasm code. It was implemented as a starting point in the code and acted as a hook to be used by ‘ccall’ in the HTML file. In ‘mainf’ all initialisations of the SDL libraries and SDL_Net_B library took place.

As mentioned in section 2.4.4, code that requires rendering has to use the emscripten main_loop functions to yield to the DOM. The first loop to be used had to be set up in ’mainf’, along with a context struct containing any other structs or variables that were needed in the code which was to act as the loop. For the structs created inside the “looping” function memory allocation was needed, as the looping function in reality is a repeated function call.

3.4.2.3 SDL_Net_B

WebSockets were used for communicating with the server. But to simplify the conversion of games using the SDL_Net library, a new library was created. The goal was to make the interaction with the new library to be as similar to the original SDL_Net library as possible. The SDL_Net_B library that was created acted as an abstraction of the Emscripten provided WebSocket library. The WebSocket library used callback functions to handle messages, whilst the original SDL_Net library used polling. This led to the need of an in-between storage for messages that could be polled for entries.

The storage was composed of a linked list of structs called nodes. Each node struct contained a pointer to the following node and a constant char pointer to a message that had arrived. Access to the nodes was possible with an index struct. Each socket had its own index struct containing a pointer to the first and last node in the linked list. When a message arrived, a new node was created and appended as the last node in the list. To get a message the first node in the list was accessed and the message it held was retrieved. The index then removed the first node and rebound its pointer to the following node.

As mentioned, the WebSocket library provided by Emscripten used callback functions to handle incoming messages. These callbacks were defined and initialized in SDL_Net_B. This meant that a user

would be none the wiser of their existence. The initialization of each callback took place in the 'SDL_Net_B_Open' function, which also created and started a new socket.

3.4.3 Server-side

The following segments will describe the method used to run Wasm code with node and how looping code was handled (3.4.3.1) and finally how the SDL_Net_BS library works and communicates via WebSockets (3.4.3.2).

3.4.3.1 Run Wasm in Node.js

The server was executed using Node.js and implemented WebSockets through the 'ws' library. The JavaScript glue file was imported into an JS file using 'require'. This file would then be used as the starting point for the server program. As seen in figure 16, an instance of the glue code had to be created to use the 'mainf' function with the use of the 'ccall' function mentioned in section 2.4.2. The function 'mainf' served as an entry point into the server code in the same manner that it did for the client code, see section 3.4.2.2 for client details.

```
const wasm = require("./server.js");
wasm().then((instance) => {
    instance.ccall('mainf', null);
  })
```

Figure 16: Making use of 'ccall' in Node.js.

A major difference between the clients 'mainf' function and the servers was that the server could not use the Emscripten loop functions as they were largely integrated with the browsers rendering mechanics, see section 2.4.4. As the server lacked any need for rendering it could instead rely on ordinary while loops.

This however occupied the Node.js event loop, making it unable to perform the asynchronous tasks needed for the WebSocket communication. Emscripten provided a sleep function that was placed in the beginning of the while loop. The time of the sleep function was set to 1ms to limit the amount of lag it would create. This sleep function allowed Node.js to handle the other asynchronous tasks associated with the WebSocket API before returning to the while loop. The use of the sleep function is seen in figure 17.

```
while(running){
  emscripten_sleep(1);
```

Figure 17: Suitable placement of sleep function in server while loop.

3.4.3.2 SDL_Net_BS

As mentioned in section 2.3.1, Emscripten does not provide any server-side library for handling WebSockets. Instead, the WebSocket library "ws" in Node.js was used. To communicate with the JS file containing the WebSocket server object the 'EM_JS' function was used as explained in section 2.4.3.

As with the client the WebSocket server worked asynchronously and a middle storage to handle messages and connections was needed. Two arrays were created in the JS portion of the project, as this was simpler than doing it in C.

One array kept track of all connections to the server and consisted of connection objects containing the actual socket object given by ‘ws’ on connection and a UUID (Universally Unique Identifier) for identifying each connection, as well as a client’s port and address. This array was paired with an integer that kept track of what index in the array was to be checked by the server to retrieve a new connection. All connections were handled in the order of arrival, which was also the order in the array.

A second array was used to store messages until handled by the user. When a message arrived the ‘ws’ socket object was compared to the once in the connection array, and if a connection existed the socket's UUID was retrieved. The message was then stored in the array with its UUID. The user then retrieves the first message in the array with the same identifier and then removes it. The array was then compressed to not leave dead space behind. If a socket was closed all messages with a matching UUID were also removed.

This still meant that any number of clients could connect to the WebSocket server, but not all were handled by the user’s server code. To not drain the server’s memory, messages originating from unhandled clients were blocked from being saved in the message storage. To find and remove unhandled connections automatically, a time to live and a handled marker was added to the connection object. When a new message arrived, the connection was checked to have been handled or not. If not, another check was made to see if it's time to live had expired. If that was the case the connection was closed.

When a connection was closed the connection object was removed and the array was compressed. As all connections got handled in order of arrival the compression meant that all handled connections were at the beginning of the array. Thus, the index for the next unhandled connection could be calculated from the number of handled connections in the array.

3.4.4 Makefile and Emscripten Ports

As mentioned in section 2.4.1 Emscripten was provided with external libraries by the use of ports when compiling. To simplify the process of compiling the C code makefiles where used for both the client and server side. With the makefile the code could easily be run with the ‘emmake make’ command, shown in figure 18.

Figure 18: Using the make file with the Emscripten emmake command.

Both the client and the server implemented ‘-sALLOW_MEMORY_GROWTH’ to enable usage of dynamic memory.

On the client side the ‘mainf’ and ‘ccall’ functions were exported as specified in section 2.4.2. The flags ‘-O3’, ‘-sASYNCIFY’, ‘-lwebsockets.js’ were used to enable the use of the Emscripten WebSocket library. SDL2 libraries were added with ‘-sUSE_SDL=2’, ‘-sUSE_SDL_IMAGE=2’, ‘-sUSE_SDL_MIXER=2’ and ‘-sUSE_SDL_TTF=2’. To allow certain image formats used by SDL2_image ‘-sSDL2_IMAGE_FORMATS='["png"]'” was used. Finally, to enable the use of resources to be called from Wasm, such as images or other files, ‘--preload-file <path>’ had to be implemented.

On the server side less flags were needed as there was no need for the SDL2 ports. Just like with the client side the ‘mainf’ and ‘ccall’ functions were exported as specified in section 2.4.2, as well as ‘lengthBytesUTF8’, ‘stringToUTF8’ and ‘UTF32ToString’ to enable their use, see 2.4.3. The ‘emscripten_sleep’ function mentioned in section 3.4.3.1 required ‘-O3’ and ‘-sASYNCIFY’. Finally to be able to load the compiled code to Node.js using ‘required’ as shown in section 3.4.3.1, the ‘-sMODULARIZE’ flag was needed.

If a project consisted of more than one C file, they could be included in the Makefile as shown in figure 19. All files would be compiled into a single Wasm file.

```
all: server.c
    $(CC) bar.c foo.c SDL_Net_BS.c server.c
```

Figure 19: Compiling “server.c” that is dependent on “foo.c” and “bar.c” several C files with Emscripten.

3.4.5 Prototype Game

To test the two libraries that were created in this study, a small game was developed utilizing both SDL_Net_B and SDL_Net_BS. It was made with SDL2 and Emscripten based on John Sharp's “Move Owl” code [23]. The original game involved simply moving a sprite around with the arrow keys on the keyboard. The game was remade to consist of 4 separate clients moving around simultaneously and being able to see each other in the browser game window. The network communication was set up so that each client connected to an echo server. When a client moved it would send a message with the player's new coordinates to the echo server that in turn would forward that message to all other connected clients that would update the corresponding sprite accordingly. The network code for the client side was done with the help of the SDL_Net_B library and the echo server with the SDL_Net_BS library. The resulting game together with a readme describing how to get it working correctly was put up in a repository on GitHub, making it accessible to the public.

3.5 Results

Two C libraries have been created called SDL_Net_B and SDL_Net_BS. The libraries in combination with the Emscripten compiler and a pre-made HTML file, made it possible to write code for an online game in a similar way to how you would write it with normal SDL and SDL_Net code. But with the ability to have the game running in a browser. The libraries are limited to emulating the TCP half of the SDL_Net library. This is because WebSocket are used to handle the communication between the browser and the server. During development tests were performed and can be found in appendix 6.

The functions in these new libraries mimic the usage of the TCP functions in the SDL_Net library. A comparison of the original SDL_net library and the SDL_Net_B and BS librarys can be seen in table 1. Because of this, if the user is already familiar with writing code for SDL_Net, the new libraries should require no further knowledge. Except that the use of Emscripten requires the main game loop of the game to be written in a specific way, as web browsers can’t support an infinite loop in a program.

A simple game was also created to prove that the libraries were working as intended and serve as a base for students and others wanting to create SDL games running in the web browser. A repository of this game is available to people with access to the GitHub via the following address, https://github.com/manorrm/MoveFourGuys_Emscripten. In this repository there is also a readme with instructions to getting it working. The game supported 4 simultaneous players together with an

echo server. SDL_Net_B was used in writing the client-side code and SDL_Net_BS was used when writing the echo server.

Table 1: Comparison of SDL_net and SDL_Net_B/BS libraries.

SDL_Net library	**SDL_Net_B/BS library**
const SDL_version *__SDLNet_Linked_Version__()	Not implemented
int __SDLNet_Init__()	int __SDLNet_B_init__()
void __SDLNet_Quit__()	void __SDLNet_B_Quit()__
void __SDLNet_Write16__(Uint16 *value*, void **area*)	Not implemented
void __SDLNet_Write32__(Uint32 *value*, void **area*)	Not implemented
Uint16 __SDLNet_Read16(__void **area*)	Not implemented
Uint16 __SDLNet_Read32__(void **area*)	Not implemented
int __SDLNet_ResolveHost__(IPaddress **address*, const char **host*, Uint16 *port*)	int __SDLNet_B_ResolveHost__(IPaddress **address*, const char* *host*, int *port*)
const char *__SDLNet_ResolveIP__(IPaddress **address*)	const char *__SDLNet_B_ResolveIP__(IPaddress **address*)
TCPsocket __SDLNet_TCP_Open__(IPaddress **ip*)	Socket __SDLNet_B_TCP_Open__(IPaddress **address*)
void __SDLNet_TCP_Close__(TCPsocket *sock*)	void __SDLNet_B_TCP_Close__(Socket *sock*)
TCPsocket __SDLNet_TCP_Accept__(TCPsocket *server*)	Socket __SDLNet_BS_TCP_Accept__(); *Only implemented in the SDL_Net_BS library.*
IPaddress *__SDLNet_TCP_GetPeerAddress__(TCPsocket *sock*)	IPaddress *__SDLNet_B_TCP_GetPeerAddress__(Socket sock)
int __SDLNet_TCP_Send__(TCPsocket *sock*, const void **data*, int *len*)	int __SDLNet_B_TCP_Send__(Socket *sock*, const char* *data*)
int __SDLNet_TCP_Recv__(TCPsocket *sock*, void **data*, int *maxlen*)	int __SDLNet_B_TCP_Recv__(Socket *sock*, char* *data*, int *maxlen*)
Any UPD functionallity	Not implemented
Any Socket Sets functionallity	Not implemented

4 Analysis and Discussion

In this chapter the results and methods used in this study are analyzed and evaluated. Recommendations for future work to be done in this area are also discussed in this chapter, as well as potential broader impacts on social and economic aspects.

4.1 Technical Solution

In this section the choices behind our technical solution and other alternative methods will be discussed. As well as how the solutions relate to the goals stated in chapter 1.

4.1.1 Solution in Relation to Goals

In relation to the goals, the final method does provide a way to use SDL_Net like syntax to create multiplayer games that run in the browser, as well as game servers deployed as web servers. The only part of the initial goal that was not fulfilled was the lack of UDP support which the original SDL_Net library provided. The cause of this missing UDP support is discussed in section 4.1.2. Overall, the nature of the solution being compatible with web browsers allows it to be used on several operating systems, such as Windows, MacOS and Linux.

4.1.2 Network Communication

The final WebSocket solution was easy to implement and work with. It was provided as a complete and fully functioning library on the client side by Emscripten and by Node.js on the server side. WebSocket was also a well proven technology as its full duplex communication channels were well suited for online games. The downside with WebSockets is the lack of a UDP like communication as it is based on TCP. UDP communication was also something that the original SDL_Net library provided and something this study set out with as a goal to implement. Given our time frame and struggles with other network solutions we chose to focus on WebSockets to make sure this study would have a working solution even though UDP would not be supported. Functionality that emulated UDP communication using WebSockets might have been possible given a slightly larger time frame.

Another possible solution would be to not use the Emscripten WebSocket library, but instead communicate with a JavaScript file that implements other forms of communication. As the JS code wouldn't need to be compiled or supported by Emscripten it would allow for a multitude of other communication methods, for example WebRTC and Netcode.io for emulating a UDP connection.

As mentioned in section 2.4.1 a port of SDL_Net was available, but the compiled code failed to create a successful connection. The amount of information given to us in the form of error messages was very small and we were unable to find any conclusive indication on how to fix it. As we had a slim time frame, we decided to abandon this method. It might have been possible to use the SDL_Net port in combination with the full WebSocket proxy server mentioned in section 2.4.5.

4.1.3 Middle Storages

To mimic the behavior of the SDL_Net library, internal storages had to be implemented to change the way data was asynchronously provided by callback functions into a polling style system. JavaScript arrays and a linked list were used as they could grow and shrink dynamically. In this study we have not tested how these storage systems would hold up under a heavy load of network traffic, and if that

would lead to a large usage of memory. We know that this solution works for four simultaneous clients, but we don't know if there is an inherent limit placed on our network libraries. Further testing is needed to examine this aspect.

4.1.4 Server Sleep

The SDL_Net_BS library used when writing server-side code, requires a very short sleep to be called somewhere in the server loop. This is because the current implementation is single threaded and can't handle the asynchronous tasks related to the WebSocket without this sleep. This is not a very elegant solution to the problem and has a negative impact on performance.

A potential future improvement would be to multithread the Node.js server. This would enable the asynchronous WebSocket tasks to be run on a seperate thread from the server loop. Doing this would eliminate the need for that loop to give up control by putting it to sleep. As mentioned in section 2.4.6, Emscripten supports the Pthreads API and thus it is possible to implement POSIX threads. In our limited testing of using this API we concluded that it would require a lot of effort and time to get working correctly, time we did not have in this project.

4.1.5 Ease of Use

A goal of this project was to keep the resulting method of porting a SDL game to the web as simple as possible. The functions in the SDL_Net_B and SDL_Net_BS mimic the use cases of the corresponding SDL_Net functions and thus should be easy to use without any experience in JavaScript and very little Emscripten knowledge. The biggest difference from the usual way of writing game code with SDL, and using the method described in this study, is that the main game loop has to be formatted in a specific way and use a specific Emscripten function to work. This results in that the user trying to port a game will be required to have some knowledge about Emscripten, but still only limited to how to structure the main game loop. With the goal of needing as little prior knowledge as possible it is of course not ideal that the main loop have to be rewritten to be able to port a game, but the way resource sharing works for browsers, this step seemed unavoidable and not something we could work around.

The lack of UDP support also limits some of the usability of this solution. This should not be something users of the libraries can't work around.

4.2 The Thesis Impacts on Broader Societal Aspects.

This thesis's potential impact on societal, economic, ecological, and ethical aspects will be discussed in this section.

4.2.1 Social Impact

This thesis could enable less experienced programmers to create both games and other programs for the browser with a server. This makes the medium more approachable and could enable people new to programming to want to realize their programming idées. The ability to simply create a working program can have a positive impact on an individual's association with programming, and in the long run this might prove to increase understanding and interest in programming and computer science as a whole.

4.2.2 Economic Impact

The technologies explored in this thesis could have some economic impact as the time and effort to rewrite existing programs and libraries are decreased or possibly even removed entirely. The expression that “time is money” holds very much true in today's software industry, and a lot of effort is put

into technology that can speed up the software production and deployment process. The software is also more accessible as the browser eliminates the need to make different versions for different operating systems, the need for specific hardware as well as updates and maintenance don't need to be handled by the client and are instead done by the application provider as previously stated in section 2.4.1.

4.2.3 Ecological Impact

The technologies discussed in this thesis have a very limited ecological impact. We can see that the decrease of time spent could have a small ecological benefit of decreasing energy consumption in the production process. The decrease in hardware dependency can also lower the negative impact on the environment.

4.2.4 Ethical Impact

We see no ethical implications as a result of the thesis.

5 Conclusions

This thesis presented a method for developing multiplayer online games running in a web browser with syntax very similar to that of SDL_Net. This was in line with the goals of developing an easy-to-use way of deploying games written in C as web applications capable of handling multiple concurrent players. The method involved the use of the Emscripten compiler together with two new libraries that were developed as a part of this thesis. The SDL_Net_B and SDL_Net_BS libraries mimic the TCP functions of the original SDL_Net library, but with network communication using the Emscripten WebSocket API. Users familiar with the SDL_Net syntax will easily be able to implement and utilize these libraries.

This solution was compatible with web browsers allowing it to be used on several operating systems, such as Windows, MacOS and Linux. As part of this thesis a simple prototype game was created using the new libraries to test their functionality. This prototype was made available on GitHub with the intention to be used as a foundation for a simple multiplayer game. This in combination with the syntax should make it very approachable for first year students at KTH.

5.1 Future Work

As a future work the UDP functionality could be implemented into the SDL_Net_B and SDL_Net_BS libraries. The current limitation of the solution lies in that while the original SDL_Net library had both UDP and TCP support, the B and BS libraries only support TCP.

Another area of interest for the future is the implementation of threading on the server. Emscripten provides some threading support in the shape of worker thread communication and memory sharing. Threading would improve the performance of the server as the sleep can be removed.

References

[1] Mackenzie J, Sitemap Emscripten [Internet] 2019 [cited2022 April 27]. Available from: https://www.jamesfmackenzie.com/sitemap/#emscripten

[2] Sharp J. emscripten-and-sdl-2-tutorial [Internet] 2016 [cited 2022 March 27] Available from: https://lyceum-allotments.github.io/2016/06/emscripten-and-sdl-2-tutorial-part-1/

[3] Webassemblygames. List of WebAssembly Games [Internet]. 2017. [cited 2022 April 27] Available from: https://www.webassemblygames.com/

[4] Stieglitz J, Nyman R. Monster Madness – creating games on the web with Emscripten [Internet] 2013 [cited 2022 April 25]. Available from:https://hacks.mozilla.org/2013/12/monster-madness-creating-games-on-the-web-with-emscripten/

[5] Webassembly, FAQ [Internet] [cited 2022 April 25]. Available from:https://webassembly.org/docs/faq/

[6] SDL Wiki [Internet]. SDL Community; 2021. Simple DirectMedia Layer; [updated 2022 Jan 2; cited 2022 April 9]. Available from: http://wiki.libsdl.org/FrontPage

[7] SDL Wiki [Internet]. SDL Community; 2021. Introduction to SDL 2.0; [updated 2022 Jan 2; cited 2022 April 9]. Available from: https://wiki.libsdl.org/Introduction

[8] libsdl. SDL Language Bindings [Internet] 2002. [cited 2022 April 9]. Available from: https://www.libsdl.org/languages.php

[9] libsdl. SDL_net Documentation [Internet] 2002.[cited 2022 April 9]. Available from: https://www.libsdl.org/projects/SDL_net/docs/index.html

[10] Yan Y, Tu T, Zhao L , Zhou Y, Wang W. Understanding the performance of webassembly applications. IMC '21: Proceedings of the 21st ACM Internet Measurement Conference. 2021 Nov 02:533–549 Available from: https://dl.acm.org/doi/pdf/10.1145/3487552.3487827?casa_token=UiYa2FWirj8AAAAA:yyOlEAoRq1ZDAXBdivtFvhJPdcYgQhng4YWXFi-oXy8JIzlDV0R1EhbztXvzLVEIGNMUrASTrozdKRg

[11] Haas A, Rossberg A, Schuff D, Titzer B, Holman M, Gohman D, et al. “Bringing the Web up to Speed with WebAssembly” from “PLDI 2017: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation”. June 18 - 23, 2017: 185 - 200. Available from: https://css.csail.mit.edu/6.858/2022/readings/wasm.pdf

[12] Rossberg A, WebAssembly Specification [Internet] 2022 [cited 2022 April 12] Available from: https://webassembly.github.io/spec/core/intro/overview.html

[13] Emscripten [Internet] Emscripten Compiler Frontend (emcc) [cited 2022 May 12] Available from: https://emscripten.org/docs/tools_reference/emcc.html

[14] Emscripten [Internet] Building to WebAssembly [cited 2022 May 12] Available from: https://emscripten.org/docs/compiling/WebAssembly.html

[15] Emscripten [Internet] Running Emscripten [cited 2022 May 12] Available from: https://emscripten.org/docs/getting_started/Tutorial.html#running-emscripten

[16] Emscripten [Internet] Emscripten Ports [cited 2022 May 12] Available from: https://emscripten.org/docs/compiling/Building-Projects.html#emscripten-ports

[17] Emscripten [Internet] emscripten-ports [cited 2022 May 12] Available from: https://github.com/emscripten-ports

[18] Emscripten [Internet] Calling compiled C functions from JavaScript using ccall/cwrap [cited 2022 May 12] Available from: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-ccall-cwrap

[19] Emscripten [Internet] Calling JavaScript from C/C++ [cited 2022 May 12] Available from: https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-call-javascript-from-native

[20] Emscripten [Internet] emscripten_run_script [cited 2022 May 12] Available from: https://emscripten.org/docs/api_reference/emscripten.h.html#c.emscripten_run_script

[21] Emscripten [Internet] Inline assembly/JavaScript [cited 2022 May 12] Available from: https://emscripten.org/docs/api_reference/emscripten.h.html#inline-assembly-javascript

[22] Emscripten [Internet] Conversion functions — strings, pointers and arrays [cited 2022 May 12] Available from: https://emscripten.org/docs/api_reference/preamble.js.html#conversion-functions-strings-pointers-and-arrays

[23] Sharp J. Emscripten and SDL2 Tutorial Part 5: Move, Owl [Internet] 2016 [cited 2022 March 27] Available from: https://lyceum-allotments.github.io/2016/06/emscripten-and-sdl2-tutorial-part-5-move-owl/

[24] Emscripten [Internet] Browser Execution Environment [cited 2022 May 12] Available from: https://emscripten.org/docs/api_reference/emscripten.h.html#c.emscripten_set_main_loop_arg

[25] Emscripten [Internet] Networking [cited 2022 May 12] Available from: https://emscripten.org/docs/porting/networking.html

[26] Fiedler G, Why can't I send UDP packets from a browser? [Internet] gafferongames.com [cited 2022 April 20] Available from: https://gafferongames.com/post/why_cant_i_send_udp_packets_from_a_browser/

[27] Mozilla Corporation, What is a web server? [Internet] [Updated 2022 Mar 5, cited 2022 March 30] Available from: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server

[28] Vadim Belsky, “Web applications vs. website: finally answered”, Scnsoft; 2017 [Updated 2017 November 9, cited 2022 Mars 31], Available from: https://www.scnsoft.com/blog/web-application-vs-website-finally-answered

[29] Brian Turner, “What is SaaS? Everything you need to know about Software as a Service”[Internet], Techradar; 2020 [Updated 2020 September 17, cited 2022 Mars 31], Available from: https://www.techradar.com/news/what-is-saas

[30] Simon P, “The Next Wave Of Technologies”, John Wiley & Sons, Inc., Hoboken, New Jersey:2021, Available from: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.6315&rep=rep1&type=pdf

[31] OpenJS Foundation, Introduction to Node.js [Internet] [cited 2022 March 23] Available from: https://nodejs.dev/learn/introduction-to-nodejs

[32] OpenJS Foundation, Overview of Blocking vs Non-Blocking [Internet] [cited 2022 March 23] Available from: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/

[33] OpenJS Foundation, Don't Block the Event Loop [Internet] [cited 2022 March 23] Available from: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

[34] OpenJS Foundation, The Node.js Event Loop, Timers, and process.nextTick() [Internet] [cited 2022 March 23] Available from: https://nodejs.org/en/docs/guides/event-loop-timers-and-next-tick/

[35] Mozilla Corporation, An overview of HTTP [Internet] [Updated 2022 Mar 5, cited 2022 March 22] Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[36] Mozilla Corporation, HTTP [Internet] [Updated 2022 Mar 8, cited 2022 March 22] Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP

[37] Mozilla Corporation, HTTP headers [Internet] [Updated 2022 Mar 5, cited 2022 March 22] Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers#websockets

[38] Mozilla Corporation, Cross-Origin Resource Sharing (CORS) [Internet] [Updated 2022 Mar 5, cited 2022 March 22] Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[39] Mozilla Corporation, CORS errors [Internet] [Updated 2022 Mar 5, cited 2022 March 22] Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS/Errors

[40] Fette I. Google, Inc. Melnikov A, Isode Ltd. The WebSocket Protocol [Internet] [Updated 2011 Dec, cited 2022 April 4] Available from: https://datatracker.ietf.org/doc/html/rfc6455

[41] Rasheed R, What is WebSocket? [Internet] [cited 2022 April 4] Available from: https://www.educative.io/edpresso/what-is-websocket

[42] PubNub Inc, What Are WebSockets, and When Should You Use Them? [Internet] [cited 2022 April 4] Available from: https://www.pubnub.com/guides/what-are-websockets-and-when-should-you-use-them/

[43] Geeksforgeeks. What is web socket and how it is different from the HTTP? [Internet] [Updated 2022 Feb 21, cited 2022 April 4] Available from: https://www.geeksforgeeks.org/what-is-web-socket-and-how-it-is-different-from-the-http/

[44] Emscripten [Internet] Pthreads support [cited 2022 May 14] Available from: https://emscripten.org/docs/porting/pthreads.html

Appendix

Appendix 1: SDL_Net_B documentation

Covers the structs and functions of the library.

Structs

IPaddress

Emulates the IPaddress struct from SDL_Net. It contains host as a string and port as an integer.

B_Socket

B_Socket struct is not to be confused with the socket handles used by Emscripten WebSocket library. Socket is a pointer to a struct containing a socket handle and a pointer to an IPaddress struct. It emulates the TCP_Socket provided by the SDL_Net library.

MessageStorage

Consists of an array of Index pointers and an integer keeping track of the number of elements in the array. If a client is connected to more than one server, each server's messages will be reachable via its Index pointer in the array.

Socket_message_index

A struct containing a pointer to the B_Socket of the server and two Node pointers. One for the first and last Node making up a linked list for storing incoming messages.

Node

Contains an incoming message and a pointer to the next node in the linked list.

Functions

int SDLNet_B_init()

Abstracts ‘emscripten_websocket_is_supported’ that checks if websockets are supported in the browser. Returns 1 if supported and 0 if not. Init also allocates memory for the message storage, more details under TCP_Recv.

void SDLNet_B_Quit()

Abstracts the ‘emscripten_websocket_deinitialize’ function and shuts down all existing websockets.

int SDLNet_B_ResolveHost(IPaddress *address, const char* host, int port)

Adds a given hostname and port to an IPaddress struct.

const char *SDLNet_B_ResolveIP(IPaddress *address)

Simply returns hostname from provided IPaddress struct.

Socket SDLNet_B_TCP_Open(IPaddress *address)

Takes an IPaddress struct and uses its content to create a new Websocket with the Emscripten Websocket library. Callback methods used by the Websocket are set. A B_Socket struct is created, containing the newly created sockets handle and a pointer to the IPaddress struct. A new socket_message_index struct for the server connection is created. Returns a B_Socket struct of new socket.

void SDLNet_B_TCP_Close(Socket sock)

Uses ‘emscripten_websocket_close’ and ‘emscripten_websocket_delete’ to close a socket.

IPaddress *SDLNet_B_TCP_GetPeerAddress(Socket sock)

Returns a pointer to the given Sockets IPaddress struct.

int SDLNet_B_TCP_Send(Socket sock, const char* data)

Sends data in UTF-8 text format and returns length of sent data or -1 at failure.

int SDLNet_B_TCP_Recv(Socket sock, char* data, int maxlen)

Accesses MessageStorage and gets the message from the first node in the linked list belonging to the socket provided. Filles given char array with retrieved data from node if length is less than maxlen. Returns -1 if connection is closed.

Appendix 2: SDL_Net_BS documentation

Covers the structs and functions of the library.

Structs

IPaddress

Emulates the IPaddress struct from SDL_Net. It contains host as a string and port as an integer.

BS_Socket

BS_Socket struct contains an id variable in the form of a UUID as a string and a pointer to an IPaddress struct. It emulates the TCP_Socket provided by the SDL_Net library.

Functions

void SDLNet_BS_Quit()

Closes all connected sockets and finally closes itself to new connections.

int SDLNet_BS_ResolveHost(IPaddress *address, const char* host, int port)

Adds a given hostname and port to an IPaddress struct.

const char *SDLNet_BS_ResolveIP(IPaddress *address)

Simply returns hostname from provided IPaddress struct.

Socket SDLNet_BS_TCP_Open(IPaddress *address)

Takes an IPaddress struct and uses its content to start the WebSocket server. Returns a pointer to a BS_Socket with the provided IPaddress and the id “SERVER”.

void SDLNet_BS_TCP_Close(Socket sock)

Closes the socket and removes the connected client and all unread incoming messages.

Socket SDLNet_BS_TCP_Accept()

Checks the list of connected clients for unhandled connections. If a new client has connected to the server, its information is retrieved. A new BS_Socket with the clients address and UUID is created and returned to the user in the server code.

IPaddress *SDLNet_BS_TCP_GetPeerAddress(Socket sock)

Returns a pointer to the given Sockets IPaddress struct.

int SDLNet_BS_TCP_Send(Socket sock, const char* data)

Passes UUID of provided socket and message to be sent to Websocket file. WebSocket is retrieved from the connection list by id and used to send the message in UTF-8 format. Returns length of message. If function returns -1 message was not sent and socket should be closed.

int SDLNet_BS_TCP_Recv(Socket sock, char* data, int maxlen)

Checks the message array for the first available message with the same id as the provided socket. Message is removed from the array and returned to the user by copying content of message to provided data char pointer. If function returns 0 no message was found and if it returns -1 no client was found. The latter warrants the need to close the socket.

Appendix 3: Simple SDL Program

Let's you move a red square up and down

```
#include <SDL2/SDL.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define WINDOWWIDTH 800
#define WINDOWHEIGHT 600

typedef struct {
    float playerPosX;
    float playerPosY;
    int playerWidth;
    int playerLenght;
    int playerMove;
} player;
#define INIT_PLAYER(X, Y) player X = {.playerPosX = Y, .playerPosY = (WIN-
DOWHEIGHT/2),.playerWidth = 10, .playerLenght = 60, .playerMove = 5}

SDL_Rect structToRectPlayerP(player *player){
    SDL_Rect rect;
    rect.x = (int)player->playerPosX;
    rect.y = (int)player->playerPosY;
    rect.w = player->playerWidth;
    rect.h= player->playerLenght;
    return rect;
}

void renderPlayers(SDL_Renderer *renderer, player *playerArray[], int play-
erArrayLength){
    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 0);
    for(int i = 0; i<playerArrayLength; i++){
        SDL_Rect playerRect = structToRectPlayerP((playerArray[i]));
        SDL_RenderFillRect(renderer,&playerRect);
    }
}

int main(int argv, char** args)
```

```
{
    SDL_Init(SDL_INIT_EVERYTHING);

    SDL_Window *window = SDL_CreateWindow("HelloSDL", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, WINDOWWIDTH, WINDOWHEIGHT, 0);
    SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, 0);

    int isRunning = 1;

    INIT_PLAYER(player1, 100);

    player *playerArray[1];
    playerArray[0] = &player1;

    running = 1:
    while(running){
        SDL_Event event;
        while (SDL_PollEvent(&event)){
            switch(event.type){
                case SDL_QUIT:
                    running = 0;
                    SDL_DestroyRenderer(renderer);
                    SDL_DestroyWindow(window);
                    SDL_Quit();
                    return 0;

                case SDL_KEYDOWN:
                    if (event.key.keysym.sym == SDLK_ESCAPE){
                        running = 0;
                        SDL_DestroyRenderer(renderer);
                        SDL_DestroyWindow(window);
                        SDL_Quit();
                        return 0;

                } else if(event.key.keysym.sym == SDLK_DOWN){
                    if(playerArray[0]->playerPosY   +   playerArray[0]->play-
erMove > WINDOWHEIGHT - playerArray[0]->playerLenght){
                        playerArray[0]->playerPosY = WINDOWHEIGHT - playerAr-
ray[0]->playerLenght;
                    }else{
```

```
                        playerArray[0]->playerPosY   =   playerArray[0]->play-
erPosY + playerArray[0]->playerMove;
                    }
                } else if(event.key.keysym.sym == SDLK_UP){
                    if(playerArray[0]->playerPosY   -   playerArray[0]->play-
erMove < 0){
                        playerArray[0]->playerPosY = 0;
                    }else{
                        playerArray[0]->playerPosY  =   playerArray[0]->play-
erPosY - playerArray[0]->playerMove;
                    }
                }
            }
        }

        SDL_RenderClear(renderer);
        renderPlayers(renderer, playerArray, 1);
        SDL_SetRenderDrawColor(renderer,0, 0, 0, 0);
        SDL_RenderPresent(renderer);
    }

    return 0;
}
```

Appendix 4: Code modified from appendix 3 to work with Emscripten

Code modified from appendix 3 to work with emscripten, showing the game in a web browser.

```
#include <SDL2/SDL.h>

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <emscripten.h>

#define WINDOWWIDTH 800
#define WINDOWHEIGHT 600


typedef struct {
    float playerPosX;
    float playerPosY;
    int playerWidth;
    int playerLenght;
    int playerMove;
} player;
#define INIT_PLAYER(X, Y) player X = {.playerPosX = Y, .playerPosY = (WIN-
DOWHEIGHT/2), .playerWidth = 10, .playerLenght = 60, .playerMove = 5}

struct context{
    int playerArrayLength;
    SDL_Renderer *renderer;
    SDL_Window *window;
    player *playerArray[4];
};

SDL_Rect structToRectPlayerP(player *player){
    SDL_Rect rect;
    rect.x = (int)player->playerPosX;
    rect.y = (int)player->playerPosY;
    rect.w = player->playerWidth;
    rect.h = player->playerLenght;
    return rect;
}
```

```
void renderPlayers(SDL_Renderer *renderer, player *playerArray[], int playerArrayLength){
    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 0);
    for(int i = 0; i<playerArrayLength; i++){
        SDL_Rect playerRect = structToRectPlayerP((playerArray[i]));
        SDL_RenderFillRect(renderer, &playerRect);
    }
}

void game_loop(void *arg){
    struct context *ctx = arg;

    SDL_Event event;
    while (SDL_PollEvent(&event)){
        switch (event.type){
        case SDL_QUIT:
            SDL_DestroyRenderer(ctx->renderer);
            SDL_DestroyWindow(ctx->window);
            SDL_Quit();
            return;

        case SDL_KEYDOWN:
            if (event.key.keysym.sym == SDLK_ESCAPE){
                SDL_DestroyRenderer(ctx->renderer);
                SDL_DestroyWindow(ctx->window);
                SDL_Quit();
                return;

            } else if(event.key.keysym.sym == SDLK_DOWN){
                if(ctx->playerArray[0]->playerPosY + ctx->playerArray[0]->playerMove > WINDOWHEIGHT - ctx->playerArray[0]->playerLenght){
                    ctx->playerArray[0]->playerPosY = WINDOWHEIGHT - ctx->playerArray[0]->playerLenght;
                }else{
                    ctx->playerArray[0]->playerPosY = ctx->playerArray[0]->playerPosY + ctx->playerArray[0]->playerMove;
                }
            } else if(event.key.keysym.sym == SDLK_UP){
```

```
                if(ctx->playerArray[0]->playerPosY - ctx->playerArray[0]-
>playerMove < 0){
                    ctx->playerArray[0]->playerPosY = 0;
                }else{
                    ctx->playerArray[0]->playerPosY   =     ctx->playerAr-
ray[0]->playerPosY - ctx->playerArray[0]->playerMove;
                }
            }
        }
    }

    SDL_RenderClear(ctx->renderer);
    renderPlayers(ctx->renderer,    ctx->playerArray,    ctx->playerArray-
Length);
    SDL_SetRenderDrawColor(ctx->renderer, 0, 0, 0, 0);
    SDL_RenderPresent(ctx->renderer);
}



int main(int argv, char** args)
{
    SDL_Init(SDL_INIT_EVERYTHING);

    SDL_Window *window = SDL_CreateWindow("Hello SDL", SDL_WINDOWPOS_CEN-
TERED, SDL_WINDOWPOS_CENTERED, WINDOWWIDTH, WINDOWHEIGHT, 0);
    SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, 0);


    INIT_PLAYER(player1, 100);
    struct context ctx;
    ctx.playerArray[0] = &player1;
    ctx.playerArrayLength = 1;
    ctx.renderer = renderer;
    ctx.window= window;

    emscripten_set_main_loop_arg(game_loop, &ctx, -1, 1);

    return 0;
}
```

Appendix 5: Search Engines and Terms

Search engines used in literary study followed by some of the search terms used.

Engines

- **Google**
- **Google scholar**
- **Duck duck go**

Terms

- **Automated server deployment methods**
- **Apache and webassembly**
- **Benefits of web application deployment**
- **Containers vs virtual machines**
- **Ccall emscripten**
- **DevOps**
- **Emscripten tutorial**
- **Emscripten games**
- **Emscripten**
- **Emcc**
- **Emscripten compiler**
- **Emscripten threads**
- **Emscripten websockets**
- **Game server deployment**
- **Gcc flags**
- **How apache web server works**
- **JS to Wasm**
- **Node.js vs nginx**
- **Node.js vs nginx vs apache**
- **Node js and Webassembly**
- **Node vs apache vs nginx**
- **Run c code in ngnix**
- **Run c code in apache server**
- **Run c code in nodejs**
- **Run SDL game in browser**
- **SDL**
- **SDL net**
- **SDL Documentation**
- **SDL net TCP**
- **SDL net UDP**
- **SDL libraries**
- **Simple python server**
- **Simple nodejs server**
- **Simple server deployment**
- **simple apache deployment**
- **Saas concept**

- **UDP in the browser**
- **Vue and Wasm**
- **Wasm**
- **Wasm to JS**
- **Webassembly games**
- **Webassembly on nginx**
- **Web server popularity**
- **Websockets**
- **Webassembly**
- **WebRTC**
- **Web application deployment methods**
- **What is saas**
- **Web apps vs website**
- **Write a server in c**
- **What is a web server**

Appendix 6: Tests

All the following tests require that you understand the SDL_Net_B and BS library and their functions, and how to run code that's been generated via Emscripten. See our GitHub for more instructions.

ID	1.1
Header	Single client – Connect, send, and receive
Goal	Connect a client to the server, send data to server that will then give response message.
Prerequisites	Have a running server that in a loop poll for a new connection. When a connection is retrieved it polls the connected socket for a received message. If a message is found it prints it to console and sends a response to client.
Anticipated result	A connection is established, and the server can check for incoming messages. If message arrives it sends back a simple response that is received by the client. The message received by both the client and the server is the same as the one sent by the other party.
Execution	1. Client code initialized the SDL_Net_B library on startup and then attempts to create a new connection. On success a socket is returned. 2. Emscripten main loop is started. 3. Returned socket is then used to send a single message in string format. 4. Socket is polled for received message. 5. If message is received it is printed in console.
Result	Connection was established with the server. A message was sent, and the response was received. The message was the same as the one sent by the other party.
Approved	Yes
Unwanted result	None

ID	1.2
Header	Single client – Disconnected by client
Goal	Disconnect from server and make socket unable to send or receive data. Server will get indication of connection being closed.
Prerequi-sites	The same server as in 1.1. with modification that receive function return -1 leads to a send of response message and then calling close on socket. Client code from 1.1 is used as a base with modifications seen under execution below.
Anticipated result	A message to be sent and received by client. After socket close it won't send or receive message to and from the server.
Execution	1. After client received message, the socket is closed. 2. After socket closed try and send new message. 3. Client then checks for incoming messages on socket and prints if any. 4. Check server console for message received. 5. Check client console for message received.
Result	When client closed server got notification via it receive function returning -1. This prompted server to send a single message to client before closing socket. Message was undelivered as socket was already closed by client.
Approved	Yes
Unwanted result	None

ID	1.3
Header	Single client – Disconnected by server and close server for new connections
Goal	Disconnect a client from server and make socket unable to send or receive data. If server disconnect client will get indication of connection being closed.
Prerequisites	Same server as 1.1 with modifications seen under execution below. Client sends message awaits response and then sends one more message and awaits response. All received messages on client are printed to console. If receive returns -1 connection was closed and client will try a reconnect and send a message.
Anticipated result	First message is received by server and client gets response. Second message is unable to be sent by client as server has closed. Client will attempt reconnect and fail.
Execution	1. When server receives message print message and send response message. 2. Then close socket and call quit function.
Result	First message is received by server and client gets response. Second message is unable to be sent by client as server has closed. Client will attempt reconnect and fail resulting in a crash.
Approved	Yes
Unwanted result	The crash on the client side because of the reconnect attempt to a server that is closed.

ID	1.4
Header	Single client – Get server info on client
Goal	Get access to the IPaddress struct from the socket struct.
Prerequi-sites	A connected socket to a server.
Anticipated result	The correct IPaddress struct is returned.
Execution	1. Use function to get peer address and give the connected socket as argument. 2. Use result to print address and port to console.
Result	Correct address and port was able to print.
Approved	Yes
Unwanted result	None

ID	2.1
Header	Multiple clients – Connection, send, and receive
Goal	Connect multiple clients to a server.
Prerequi-sites	Use the move4guys code provided on GitHub. Have server running.
Anticipated result	Four clients can connect to the server and send data about their movements. This data will be provided to the other clients and used to update the other players on screen.
Execution	1. Start client on four different file servers. 2. Connect each client to the server. 3. Move the player character on every client instance. 4. Check that the movements are shown on the other clients.
Result	The four clients could connect, and their movement was shared and updated to the other clients.
Approved	Yes
Unwanted result	None

ID	2.2
Header	Multiple clients – Disconnect clients and reconnect
Goal	Be able to disconnect clients and for the other clients to function as normal. Be able to connect to the server again.
Prerequisites	Four connected clients as in 2.1.
Anticipated result	Disconnecting a client does not affect the other client's functionality. New clients can be connected without disrupting the existing client's functionality. Newly connected clients have the same functionality as the other clients.
Execution	1. Refresh one client browser page to close that connection. 2. Move the characters in the still connected client windows. 3. Check each connected client for updates on the character movements. 4. Disconnect a second client as in step 1. 5. Repeat step 2 and 3. 6. Try and connect the two unconnected clients. 7. Repeat step 2 and 3.
Result	Disconnecting and reconnecting clients did not affect the other client's functionality.
Approved	Yes
Unwanted result	None

ID	2.3
Header	Multiple clients – Disconnect client by server and reconnect.
Goal	For the server to be able to disconnect a single client without affecting other client's functionality.
Prerequi-sites	Same as in 2.1 with the added modifications given in executions below.
Anticipated result	The server can disconnect clients without affecting other client functionality.
Execution	1. After four clients are connected select two of them and close their connection. 2. Do step 2 and 3 from test 2.2. 3. Refresh one of the disconnected clients and try to reconnect. 4. Repeat step 2.
Result	Client connections being closed did not affect other client's functionality. Reconnection to server was possible with no effect on other client's functionality.
Approved	Yes
Unwanted result	None

ID	2.4
Header	Multiple clients – Quit server with clients
Goal	To be able to close all existing connections and make server unavailable.
Prerequi-sites	Four connected clients as in 2.1. With added modifications described in executions below.
Anticipated result	All client connections are closed and new connections to server are prohibited.
Execution	1. When the fourth client connects call quit on server. 2. Check all clients if they received on close message. 3. Refresh client and try to reconnect.
Result	All clients disconnected and no new connections were possible.
Approved	Yes
Unwanted result	None

ID	2.5
Header	Multiple clients – Get client info on server
Goal	Get info of client address and port for each client
Prerequi-sites	Four connected clients as in 2.1. With added modifications described in executions below.
Anticipated result	The correct client's info can be retrieved.
Execution	1. When message arrives from client get peer address. 2. Use returned IPaddress struct to print address and port of client to console.
Result	The correct clients address, and port was retrieved.
Approved	Yes
Unwanted result	None