TogetherTacToe Project Documentation

Fraunhofer FOKUS, Technische Universität Berlin; supervised by Dr. Louay BASSBOUSS

Jörn STRÖMSDÖRFER
Technische Universität Berlin
Berlin, Germany
joern.stroemsdoerfer@campus.tu-berlin.de

Marcin BOSK
Technische Universität Berlin
Berlin, Germany
marcin.l.bosk@campus.tu-berlin.de

***Abstract*—The vast majority of people nowadays carries a very powerful computing device with them, namely a smartphone. It is used to access information and capabilities on demand, mostly through numerous applications installed on the device. One such application is the internet browser. It allows access to websites which provide their content using traditional and modern web technologies. With introduction of multiple new and flexible frameworks, the creation of immersive web applications is more accessible to creators than ever. This documentation outlines the project TogetherTacToe which combines the use of numerous cutting-edge frameworks from various technological fields in order to deliver entertaining content to users. It is a showcase game that integrates multiple advanced web technologies to provide immersive single- and multiplayer virtual reality gameplay available straight in the browser on a handheld device of the user's choice.**

***Index Terms*—VR, WebVR, WebXR, A-Frame, C++, JavaScript, WebAssembly, MEAN Stack, mongoDB, node.js, Socket.io, TicTacToe, TogetherTacToe, web application, web game, web technologies.**

I. TOGETHERTACTOE: AN INTRODUCTION

In the vast realm of technology the field of web technologies is among the most frequently changing and rapidly advancing ones. It seems that almost on a daily basis new frameworks and APIs are provided to the worldwide community of web developers.

For this enticing reason we enrolled in the "Advanced Web Technologies" project at the Fraunhofer FOKUS with great motivation. Of the many project options provided we chose "WebVR", i.e. projects utilizing the cutting edge API for delivering virtual and even augmented reality content in the native web browser of VR devices (e.g. smartphones). While already being thereby on the forefront of modern technology, we decided to go further and set an ambitious task for ourselves: building a software product that is fun and intuitive to interact with and that incorporates an entire array of modern technologies while at the same time utilizing classical web standards.

The solution we came up with bears the name **TogetherTacToe** and it is a single- & multiplayer VR game. Here is an overview of its usage:

Generally the procedural flow through the game is that of choosing whether to play against an artificial opponent (i.e. singleplayer mode) or against a random human opponent (i.e. multiplayer mode). In the former, the player is presented with further options of choosing a difficulty level and which symbol to play with (i.e. X or O) before being able to launch into the game. In the latter, matching up of players will occur automatically according to the time of them launching a game. From there on the game follows familiar scheme of Tic Tac Toe, where on a 3x3 playing board each of the two opposing players tries to get three fields in a row. As mentioned before this entirely virtual reality game is played simply by navigating to a web address in a web browser.

Fig. 1. Screenshots of the menu and playing field of **TogetherTacToe**

With this workflow through the game described above, we are certain to have created an intuitive and foolproof usage paradigm. What is of greater interest to this paper however is a look behind the curtain at the inner workings of the solution. In the coming chapters we will give general overviews of used technologies (see *II. Overview of used Technologies*) and an in-depth look into the architecture of the solution (see *III. Project Architecture*). We will then include an insight into our development process and therein milestones we set, challenges we faced and lessons we learned (see *IV. Development Process*) before concluding with a summary of this project (see *V. Outlook*).

II. Overview of used Technologies

This chapter will in a general manner introduce the technologies used for this project and in some cases showcase possible alternatives and then outline considerations for the choice for one of these alternatives.

A. WebVR

WebVR is a JavaScript API that provides support for virtual reality devices and therein interfaces to their respective components necessary for VR experiences. It is an open standard that makes it possible to experience virtual reality in a web browser and therefore with a wide range of devices, from the heads-up display VR devices (e.g. Oculus Rift) to smartphones.
There are several frameworks based upon this standard, that enable fast-paced development of virtual reality applications that are then widely and easily accessible through a web browser. [22]

1) React VR: React VR is a VR framework that allows the development of VR apps by using only JavaScript. It uses the same design as React, which is a declarative and component-based JavaScript library for building user interfaces.
Like A-Frame, React VR is based upon the JavaScript 3D library three.js. [13]

2) babylon.js: babylon.js is an open source 3D engine directly based off of the JavaScript library WebGL. It allows the creation of 3D games and experiences in the realm of virtual reality with HTML5, WebVR, and WebGL/Web Audio. The latter is designed to render interactive 3D computer graphics and 2D graphics within any compatible web browser, without the use of any plug-ins. This signifies far reaching compatibility for babylon.js. [6]

3) A-Frame: A-Frame is a WebVR framework which was started by the Mozilla VR team with the aim of giving web developers a low-threshold ability of building 3D and VR worlds. Since the framework can be used from HTML it is accessible to everyone. It therefore has extensive cross-platform performance.
A-Frame is open-source and its three dimensional elements are based upon objects of the JavaScript 3D library three.js. [1]

For reasons of platform independence, ease-of-use and development time (i.e. time-to-market) we chose A-Frame as our framework on top of the WebVR JavaScript API. Beyond that the prior experience in three.js and WebGL development existent in our group made this framework an ideal choice for us.

B. WebAssembly

"WebAssembly is a portable binary instruction format. It is compiled from high-level languages (e.g. C/C++) and intended for deployment to the web. Currently it is supported by all four major browser engines (i.e. Chrome, Edge, Firefox, and Safari)." [23]
With WebAssembly a developer writes her code in high-level languages such as C++ as usual with just very minor changes. Therefore she has at her disposal the entire range of the language's capabilities, e.g. direct access to the memory and 64-bit integers. This code is then compiled to a Wasm module, deployed to a website and launched via JavaScript. Afterwards the binary format of WebAssembly is natively decoded and executed in the JavaScript Virtual Machine that is included in every browser. This means that the familiar security of the sandboxed browser remains.

Comparison: JavaScript & WebAssembly
WebAssembly is not intended to substitute, but rather compliment JavaScript on the web. JavaScript was not designed for its use in today's web landscape as a language for high performance computations and large applications. Wasm aims to close that gap.

Loading Speed
Although the Wasm binary format is executed in the Virtual Machine within the web browser as JavaScript is, there are significant gains in terms of parsing performance. This is due to the fact that JavaScript has to initially do the interleaved tasks of parsing, compiling + optimizing and re-optimizing before execution. [11] WebAssembly's binary format on the other hand is simply decoded natively and after a faster compilation much more quickly executed. Therefore the cold-load user experience is critically improved.
Speed comparisons done by several individuals and companies illustrate that by switching from JavaScript to WebAssembly their load times improved by 3x up to 20x. [9] [24]

Size
The improved load time of Wasm is also owed to the compact size of Wasm modules needed to be transferred over the wire to the client. When directly comparing equivalent algorithms in firstly gzipped minified JavaScript code and secondly a Wasm module the binary format of WebAssembly is still around 10-20% smaller. [3] [2]

Runtime Speed
There are significant advantages in execution speeds beyond those arising from the aforementioned initial code implementation in C++ (i.e. 64-bit integers (operations 4x faster), direct memory access, various CPU instructions, no garbage collection, ...).
WebAssembly is intended as compiler target, i.e. designed for compilers to generate and not for humans to write. Therefore it can provide a more ideal set of instructions for machines. This results in a 10% to 800% speed improvement over JavaScript code that might even have been coded by a developer highly knowledgeable about the optimizations of JavaScript's just-in-time compiler. [11]
Direct practical speed comparisons show an overall 8x to 15x improvement for execution from a Wasm module instead of JavaScript code. [10] [7] [21]

It is however at this point worth noting that there are cases which do not warrant the choice of WebAssembly, since

there would in fact be a decrease in runtime speed. Mainly this is when the software solution regularly passes large and complex objects from the JavaScript code, whose main task should be manipulating the DOM and capturing UI events, to the Wasm module.

C. MEAN stack

MEAN stack is a collection of open source JavaScript elements which consists of: MongoDB, Node.js, Express.js and Angular.js (briefly discussed in the following subsections, except for Angular.js which is not used in this project). Its aim is to simplify development of dynamic web applications by providing all necessary components, such as a database, back-end, front-end and JavaScript runtime environment within the stack. [12] [8]
Its uses are not limited to pure web applications. Within the research community it has recently been evaluated as a presentation platform development tool for data gathered by IIoT devices. [17]

1) MongoDB: MongoDB is a free and open source NoSQL-database with a simple, easy to use document model. It supports multiple programming languages including Python, Java, C++, C# and JavaScript. It has a highly adjustable data model where the data is stored within constructs similar to JSON elements mapped to objects on the application level code. It enables effortless data analysis by supporting real-time aggregation together with ad-hoc queries and indexing. From the design perspective, it is a distributed database with tools allowing fast queries, indexing and real time data aggregation. [14]

2) Express.js: Express.js is a framework that provides a collection of additional features for web and mobile development. It is built on top of Node.js with its lightweight construction not influencing the performance of applications that run using its parent framework. It provides additional features to Node.js such as HTTP routing and GET/POST/DELETE operations. Express.js is also a basis for multiple frameworks based on Node, such as: Feathers, KeystoneJS or Blueprint. [17] [18] [19]

3) Node.js: Node.js (also called Node) is a server-side JavaScript environment used for network (web) applications, in construction similar to the Twisted library for Python. It handles web requests asynchronously using callbacks called on connections and incoming traffic. Its design is based on events and in contrast to many modern solutions does not use multi threading and resource-locking for operation. This allows for better scalability and finer-grade control over the application execution. Node is mainly designed for low latency and streaming applications that run over HTTP protocol. It is often used in conjunction with Express.js. [16] [20]

D. Socket.io

Socket.io is a JavaScript framework that enables real-time text or binary communication between a server and client. It is event-based and provides straightforward connection establishment and session management with automatic re-connection when the server becomes unavailable for a short period of time. The framework also has support for "namespaces" which are channels/groups of sockets that can communicate with one-another over the server using broadcast, multicast or unicast messages. It is widely used, with its most prominent applications being Microsoft Office, Yammer, Zendest and Trello. [5] [4]

III. Project Architecture

The task of supplying a smooth and entertaining experience to the end user requires a well designed application architecture. For that reason the functionality of the solution is split into two main parts - the server and client. This division allows distinguishing between front/back-end development and testing of the solutions separately without them interfering with one another. It also made the final integration of the visual and underlying aspects of the game much smoother and allowed avoiding complicated relations between the code running in the user's browser and on the server.

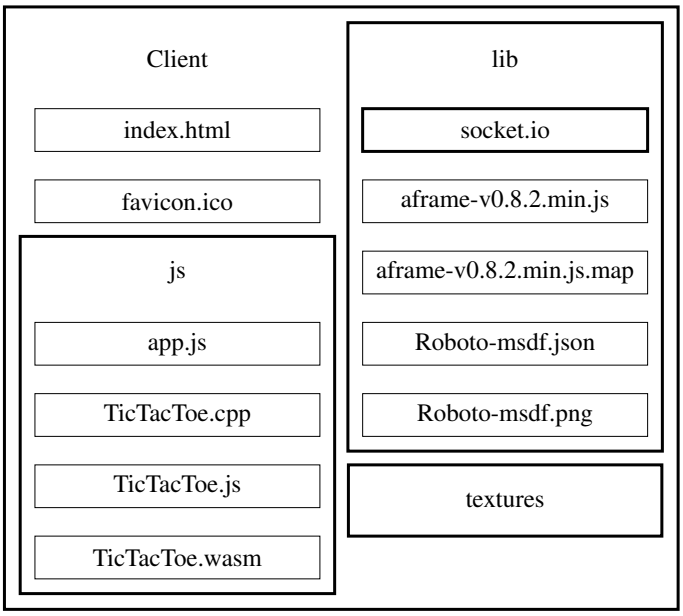

Fig. 2. File Structure for Client

That clear separation can be seen on multiple levels of the application, but it is mostly noticeable in the introduced file structure. There one can clearly see the distinction into a client file structure which can be seen in *Figure 2* as well as a server one presented in *Figure 3*. Thicker lines in these figures represent folders.

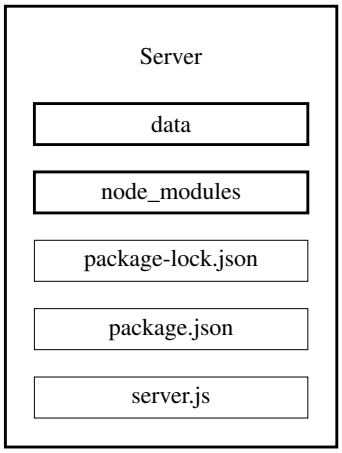

Fig. 3. File Structure for Server

A. Singleplayer

Whereas the multiplayer mode utilizes for its functionality components from both client and server (see *III.B.*), the singleplayer mode is exclusively a client side application. The comprised components from *Figure 2* are:

- index.html
- js/app.js
- js/TicTacToe.js
- js/TicTacToe.wasm
- lib/aframe-v0.8.2.min.*
- lib/Roboto-msdf.*
- textures/*

Roughly explained, the graphics components are all contained in the HTML-file and work in accord with the A-Frame library files while all playing functionality is implemented in the WebAssembly module, i.e. the WASM-file and its functions are accessible via the glue file TicTacToe.js. All remaining files are pictures, textures and fonts necessary for a polished graphical user experience.

First part of this subsection will be an in-depth look at the playing functionlity: As mentioned before, all functionality necessary to play in singleplayer mode against a bot is contained within the WASM-file. The most important functions exposed to JavaScript-file js/app.js via the glue file js/TicTacToe.js are `void _SetMove(int field, int player)`, `int _GetMove()` and `int _isGameOver()`. This glue file together with the WASM-file are created from js/TicTacToe.cpp with the compiler tool EMSCRIPTEN. What one can see here, is that the communication between JavaScript and WebAssembly is lightweight (i.e. only integers are passed back and forth) and minimal. Internally, WebAssembly keeps its own 9 fields long array that represents the current game status by integers denoting which player - if any - has taken a field.

The human player plays a move by passing her own ID and the field ID (both integers) to WebAssembly. Then the game checks if the game has been won and if so returns the winning player's ID back to JavaScript. If the game is still afoot, then the application fetches the bot's next move from the C++ module and the check whether current game is over is performed again.

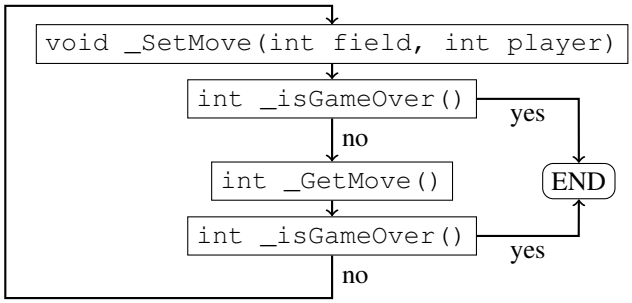

Fig. 4. Workflow of the singleplayer mode within WebAssembly

In the original CPP-file two algorithms have been implemented. The first one, which user can select by choosing difficulty level EASY (see *Figure 1*) is a straightforward implementation of the MinMax algorithm. [15] The second option is an improved version of it that is able to look ahead further into the game and therefore calculate the bot's next move more perfectly by examining every possible outcome. It can be selected by choosing difficulty level DIFFICULT. In this version of the algorithm a player can achieve a tie at most, but never win against it.

The second part of this subsection will be a description of the inner workings of the graphical side of **TogetherTacToe**: As described in *II.A* the VR framework used is based upon three.js and therefore many capabilities for graphical display are provided to the developer. First and foremost this is the creation and manipulation of 3D objects, i.e. the playing field, boxes within fields, the scope and textures wrapped onto boxes. The used mechanism for selecting buttons and boxes through movement of the VR device is called a Raycaster and all 3D objects that would respond to their selection via the Raycaster are combined into a "cursor-listener" group.

In order to make sure no play move data was at any point during the game lost or corrupt a coherent scheme was imlemented, wherein the WebAssembly module would be passed the play move data first and the graphical interface would be changed after. The latter was achieved by firstly only having boxes shootable which were in a designated "shootable"-group and secondly by replacing them, once shot, with new 3D objects and wrapping them respectively with a texture of the player's or bot's symbol (i.e. 'X' or 'O'). Thereby the graphical representation visible to the user would always match the internal game status array of the WebAssembly module.

If after any move (i.e. either player's or bot's move) the integer returned from `_isGameOver()` function indicated a winner, no objects would be shootable anymore and with a designed time delay the user would be informed of the game's outcome. Once at this point, the user has an option to

reset the web application and play another game.

B. Multiplayer

The multiplayer functionality of **TogetherTacToe** is strictly based on the singleplayer version, however it required some additional components to be built. Most importantly a server had to be introduced into the mix and the client needed an extension which allows it to communicate with the server. To achieve the multiplayer functionality the same files as mentioned in the singleplayer section *III.A* are utilized, with the addition of files from the Server/ folder (see *Figure 3*):

- server.js - being the actual server
- node_modules/ - node.js related packages
- package-lock.json and package.json - also node.js related
- data/ - containing the mongoDB database

Extensions to the client functionality have been implemented in Client/js/app.js.

1) Server Side: In order to properly serve the clients a server had to be prepared from the ground up. Since the beginning a few requirements had to fulfilled in order to provide the full **TogetherTacToe** functionality without any problems. The server needed to

- provide the whole website to requesting client,
- have a database to manage the multiplayer sessions and
- have the ability to exchange information with clients even after the first page request has been handled.

To achieve that a few new technologies were introduced into our project, mainly the MEAN stack (in our scope MongoDB, Express.js, Node.js, see *II.C*) and Socket.io (see *II.D*).

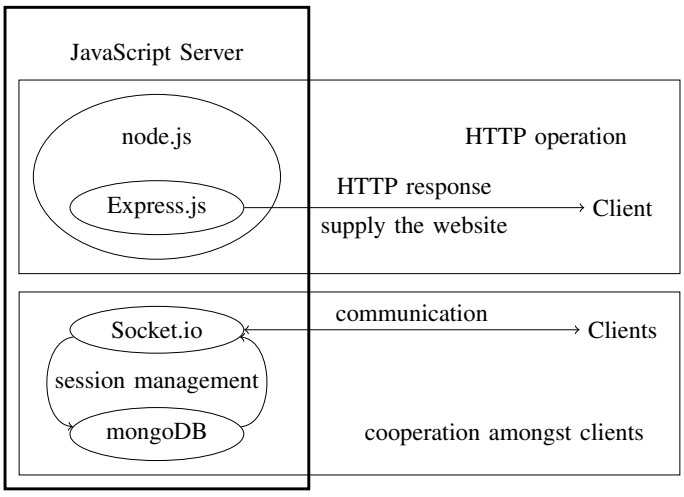

Fig. 5. Interoperation of various elements within the server

These components have been put together in a particular way to fulfill the requirements put against our server. The correlations are presented as per *Figure 5*. As one can see, the JavaScript server consists of two parts. The first one allows for HTTP operation, that is supplying the website to clients. It is achieved using the Express.js framework in cooperation with node.js in a straightforward way, where simply the whole Client/ folder is provided on request using the `express.static` function.

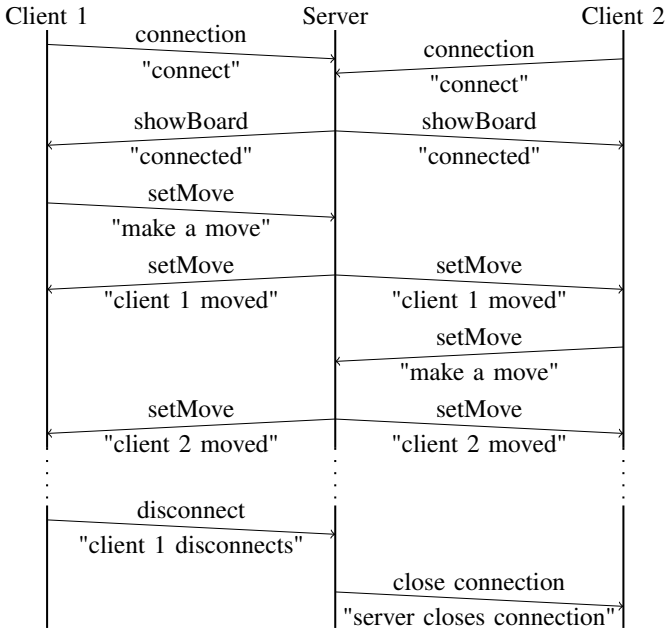

Fig. 6. Exemplary communication scheme between the clients and server

The second part is significantly more involving, being responsible for communication with the clients and session management in multiplayer games. Each of the sessions is saved within the mongoDB database in a collection named "sessions". Each entry contains a JSON string: `{id: entry_ID, c1:ID of 1st client, c2:ID of 2nd client}`. Sessions are identified based on the id of an entry. *c1* and *c2* identify the first and second client that joined the session respectively. An entry is firstly created when the first client of the pair joins using a Socket.io connection message, as per *Figure 6*. The second client joins in a similar way and is added to the session entry. In the next step, the server sends a "showBoard" message to both clients, which indicates that the game is started and both clients can play. Further on the server is simply forwarding messages from the client that has just made a move to both of the clients which only upon reception of a message update their graphical playing boards and can make further moves respectively.
The server does not know when the game ends (see *2) Client Side* for further information on this). A session is deleted when one of the clients disconnects (what happens by simply closing the web browser or refreshing the page/going back to the main menu) and simultaneously the server closes the connection to the other client.
By using the mongoDB for session management, it is ensured that the solution is scalable and can handle a tremendous amount of parallel running multiplayer games, limited only by the storage on the server. The alternative surely highly possible approach of handling session in the server code and therefore purely in memory would not provide this advantage.

2) Client Side: As the client has been designed to be later on extended for the multiplayer mode from the very beginning, integration of it did not require as much effort as implementation on the server side. All of the original functionality was maintained and additionally only minor adjustments were introduced.

On the visual side, an additional DOM element was added: a "Start Multiplayer" button (see texititFigure 1) which gives players another choice on the home screen, namely to play against one another.

More interesting are changes made to the underlying functionality of the client. Firstly, the A-Frame cursor-listener was adjusted in a way that it can be used in both the single- and multiplayer modes. To achieve that, the existing functions are simply split into two cases - one for singleplayer (which remained unchanged) and one for the multiplayer mode (which functionality had to be significantly adjusted). For this differentiation a variable is used that is set upon a choice of multiplayer vs singleplayer modes.

The shooting function was, in a sense, simplified for multiplayer. On the cue of a player making a move the only things that happen are: a change of the current player to the other one (after making a move the player cannot make another move immediately) at the client and a message sent to the server which contains information on the chosen field.

The most significant change at the client concerns the way in which boxes are replaced after a move is made. In contrast to the singleplayer version there is no immediate replacement of the box objects, but rather the servers' response is awaited. Only after a message with information on the move arrives from the server at either client, the playing field is updated and the opponent is allowed to play.

As the server only implements a really simple move information exchange functionality - there is no game logic present at the server - some of the pure singleplayer functionality is utilized. The `_SetMove(int field, int player)` and `_isGameOver()` functions prepared within the already existing WebAssembly module to keep track of the game were used to determine whether it is finished and who won.

IV. Development Process

The very first milestone in our process to develop this application was the choice for our WebVR framework. As described in *II.A* the choice fell unto *A-Frame*. A well documented "Getting Started"-guide and our familiarity with web technology basics made getting our first virtual reality environment onto the screen very comfortable.

We moved on to constructing a basic VR scene that would later serve as the TicTacToe playing field. This involved many hand drawn sketches and discussions on positioning and measurements. At this point there was no functionality implemented and only a scene to look at.

Being able to play this virtual reality version in singleplayer mode was the next target of our process. Here an architecture decision needed to be made: Where and how would we implement the bot algorithm to play against? Since our team was at the time very excited about the cutting edge technology of WebAssembly (see *II.B*), we at this point needed to leave the web development environment and go into a conservative IDE for developing C++ applications. Here we built a console singleplayer version of TicTacToe. Using some developer forums and looking into our old mathematics notes helped us develop the MinMax algorithm and an improved version of it. These two versions would later serve as different difficulty levels.

The following step of connecting our dummy virtual reality scene with the functions written in C++ and contained within the WebAssembly module was highly rewarding. We finally got to play the game in a VR environment. What followed until our intermediate project presentation was a lot of polishing and graphical improvements to have a nice demo to show to our fellow students.

This point in the process marked the completion of a tremendous milestone. To reach the next and final milestone - the multiplayer functionality - we would experience several setbacks and would go down the wrong path a few times until we would finally be on the road to success.

Our first attempts to connect two clients with each other envisioned as little use of additional frameworks as possible. Therefore we first tried to work with asynchronous requests (XMLHttpRequests) and server-push technologies, but ultimately found that they would not provide the stability and scalability required for our solution. Beyond that our first conservative attempts at having two clients play against each other yielded the design decision to use a database on the server side. Additionally this part of the architecture would later prove very valuable, since it opened the possibility of discussions about not only connecting clients but collecting precious metrics to be used to improve the entire user experience at later points in time.

At the very end of that erratic process was the choice to utilize the M.E.A.N. stack and all its perks, from the comfortable package management of node.js and therefore the easily implemented use of WebSockets to the NoSQL database of mongoDB. Once made, the continuing development process proved comfortable and rewarding.

In order to get the final application out to as many testers as possible we started hosting it on AWS, where it still runs continuously and stable today for anyone to enjoy: tinyurl.com/TogetherTacToe.

V. Outlook

We - as a team of developers - are very satisfied with the opportunities this project provided to us. From server to client technologies, from the very basics of original website development to cutting edge web capabilities, we got to experiment with a broad realm of technologies. Looking at the frameworks and technologies we were able to harmonize in our goal to create a fun and social little game, we are very proud of **TogetherTacToe**.

What we are even more excited about is the possibilities this opens up for us in future projects. The project shows that there

can be efficient and small-footprint communication between a virtual reality web application and high-level functions within a WebAssembly module originally written in C++. It showed us that with modern frameworks the journey from inception to running product of a server architecture is shorter then ever before.

We can very well imagine this little solution being extended to be something much more complex then TicTacToe, e.g. virtual reality chess. Beyond that, we can also already see how the game could be not only VR but AR, i.e. augmented reality. Our chosen framework makes this a comfortably attainable goal. And we have also already considered is how machine learning could be used in a more complex game to more personalized and smarter respond with countermoves to a human player.

The possibilities in our modern realm of technologies seem endless and we are excited to continue building software in it.

REFERENCES

[1] A-FRAME. A-FRAME - Introduction. aframe.io/docs/0.8.0/introduction. Accessed: 2019-01-17.

[2] Alon Zakai. A WebAssembly Toolchain Story: Box2D code size measurements. kripken.github.io/talks/emwasm.html#/9. Accessed: 2019-01-17.

[3] Alon Zakai @ moz://a HACKS. Why WebAssembly is Faster Than asm.js. hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js. Accessed: 2019-01-17.

[4] Automattic. GitHub - socketio/socket.io: Realtime application framework (Node.JS server). github.com/socketio/socket.io. Accessed: 2019-01-18.

[5] Automattic. Socket.IO. socket.io. Accessed: 2019-01-18.

[6] babylon.js. babylon.js - Documentation Overview. doc.babylonjs.com. Accessed: 2019-01-17.

[7] Bojan Đurđević. Pitch detection comparison between Wasm and JS. s3.amazonaws.com/wasm-vs-js-tuner/index.html. Accessed: 2019-01-17.

[8] Bakwa Dunka, Edim Emmanuel, and Yinka Oyerinde. Simplifying web application development using-mean stack technologies. 04, 01 2018.

[9] Evan Wallace @ Figma. WebAssembly cut Figma's load time by 3x. figma.com/blog/webassembly-cut-figmas-load-time-by-3x. Accessed: 2019-01-17.

[10] Indigo Code. WebAssembly overview - It's cool, it's really fast. youtube.com/watch?v=1J6Z5wBfSnQ. Accessed: 2019-01-17.

[11] Lin Clark @ moz://a HACKS. What makes WebAssembly fast? hacks.mozilla.org/2017/02/what-makes-webassembly-fast. Accessed: 2019-01-17.

[12] Linnovate. MEAN Stack - welcome page. mean.io. Accessed: 2019-01-18.

[13] Mirosław Ciastek @ medium.com. Bring Virtual Reality to Your Browser with React VR. medium.com/swinginc/bring-virtual-reality-to-your-browser-with-reactvr-fd9dc62aa8b1. Accessed: 2019-01-17.

[14] MongoDB. What is MongoDB? mongodb.com/what-is-mongodb. Accessed: 2019-01-18.

[15] Never Stop Building LLC. Tic Tac Toe: Understanding the Minimax Algorithm. neverstopbuilding.com/blog/minimax. Accessed: 2019-02-25.

[16] Node.js Foundation. About Node.js®. nodejs.org/en/about/. Accessed: 2019-01-18.

[17] Andrew John Poulter, Steven J Johnston, and Simon J Cox. Using the mean stack to implement a restful service for an internet of things application. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 280–285. IEEE, 2015.

[18] StrongLoop, IBM, and other expressjs.com contributors. Express.js - 4.x API. expressjs.com/en/4x/api.html. Accessed: 2019-01-18.

[19] StrongLoop, IBM, and other expressjs.com contributors. Express.js - Home. expressjs.com. Accessed: 2019-01-18.

[20] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, Nov 2010.

[21] Tom Lagier @ medium.com. Screamin' Speed with WebAssembly. hackernoon.com/screamin-speed-with-webassembly-b30fac90cd92. Accessed: 2019-01-17.

[22] W3C. WebXR Device API - Editor's Draft, 15 January 2019. immersive-web.github.io/webxr. Accessed: 2019-01-17.

[23] WebAssembly. WebAssembly - Overview. webassembly.org. Accessed: 2019-01-17.

[24] WebAssembly. WebAssembly FAQ: Why create a new standard when there is already asm.js? webassembly.org/docs/faq/#why-create-a-new-standard-when-there-is-already-asmjs. Accessed: 2019-01-17.