Universiteit Leiden
The Netherlands

Opleiding Informatica

Toward a

Massive Multiplayer Framework

Wouter D. Vermeulen

Supervisors:
Dr. Michael S.K. Lew & Dr. Erwin M. Bakker

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

3/6/2018

Abstract

A MASSIVE MULTIPLAYER FRAMEWORK is a framework that allows a large number of players to participate simultaneously over an internet connection. Even though this number is theoretically unlimited, the hardware usually limits the amount of possible connections. Having such a framework embedded in a website comes with another problem: the website was never able to address the graphics card directly. For games to work in the web browser, recent massive multiplayer games use a plugin to work around this. Current web technologies though, allow for a different solution.

In the current state-of-the-art, special multiplayer web servers which were designed from the ground up for the web/gaming context like Node.js are considered to have the best performance as opposed to generic servers such as Apache or Nginx. In this thesis we experimented with WebGL, using it to create a massively multiplayer framework with Node.js and Apache. One of the questions we ask within this work is how far can multiplayer servers be scaled and also what are the strengths and weaknesses of each approach?

Figure 1: The game we developed running on some of our machines

Contents

Chapter 1

Introduction

In this chapter we give an introduction to the problem addressed in this thesis.

1.1 Current situation

In 2009 web-based online gaming applications were predominately utilising Adobe Flash or Java Applets as their core technologies. These games were often casual, two-dimensional games and do not utilise the specialist graphics hardware which has proliferated across modern PC's and Consoles. Multi-user online game play in these titles was often either non-existent or extremely limited. [1] However by 2011, WebGL has entered the stage [2] solving some of the largest problems Flash had by utilising the graphics card directly. [3]

WebGL brings hardware accelerated 3D graphics to the web without plug-ins. WebGL works in any browser that supports Khronos OpenGL (originally the Open Graphics Library) or OpenGL ES specification. The development of WebGL opens up new possibilities for browser based games and the internet as a whole. [4] [5] [6] Over the past 15 years OpenGL has become the standard API for teaching computer graphics courses in computer science and WebGL has become an increasingly attractive API for both educational and professional practice. [7]

For example, Google demonstrates a WebGL HTML5 port of the first-person shooter game, Quake 2. Players could play with each other in real time by using WebSocket. According to Google, the frame-per-seconds of the HTML5 Quake 2 can be up to 60 fps. [8]

This opens up the possibility for a larger scale game. There have already been some ambitious projects like the browser-based MMOG RuneScape [9], which efficiently hosted over 1 million subscribers [10] and used to run in Java but has now made the switch to WebGL. This is still small scale compared to the 12 million subscribers of the MMOG World of Warcraft. [11]

1.2 Problem definition

This thesis is about researching the possibilities of implementing current web multi-client algorithms and technologies such as WebGL in an MMOG environment.

The critical issue with making a Massively Multiplayer Online Game (MMOG) is the scalability. [12] More players take up more resources. Other researchers have already proven that it is possible to implement an MMOG with WebGL. [5] However, our goal will be pushing the scalability to its limits for a fixed computational platform.

1.3 Research Question

The research question this thesis aims to answer is:

"What is the largest number of people that can be supported by a massive multiplayer framework based on current web multi-client algorithms and technologies (e.g. WebGL, Node.js and Apache) for a fixed computational platform?"

In addition, we are also interested in studying and assessing the strengths and weaknesses of Node.js versus Apache for multiplayer frameworks.

1.4 Thesis Overview

This first chapter is an introduction to the problem we aim to solve. Next we will discuss the way we want to solve the problem in Chapter 2. The actual experiment is discussed in Chapter 3, the results are featured in Chapter 4 and discussed in Chapter 5. In Chapter 6 all of this will come together in a conclusion.

This "bachelor thesis" is written by Wouter D. Vermeulen, with LIACS supervisors Dr. Michael S.K. Lew and Dr. Erwin M. Bakker.

Chapter 2

Implementation

In this chapter we will describe the work we have done for the experiment. This project has several parts. The first part is the client side. The client side needs to display the client's character in a virtual world, and its contestants' characters in the same world. The server side is the second part. The server needs to update and store every client's current situation and share this information with every other client.

2.1 The game

The requirements we had for this game were that it:

1. Must have 3D graphics;
2. Must not have a limited number of players;
3. Should have players who are not fully predictable in behaviour;
4. Should be the most basic game possible;
5. Could be fun to play;
6. Won't be resource heavy.

Starting from Requirement 4, we decided upon the 3D multiplayer version of Snake: GLtron. Like Snake the player only has three options at any point in the game. Either the player goes 90 degrees left or right, or does nothing which means the player goes forward. The player leaves behind a trail which functions as a wall. The player must avoid crashing into any wall for as long as possible. Unlike Snake there are other players trying to cut the player off to make him crash into the wall. [13]

There were already a couple of versions available of this game online, which means we did not have to reinvent the wheel. [14] For this project we started with an existing version of GLtron. GLtron is an open-source game

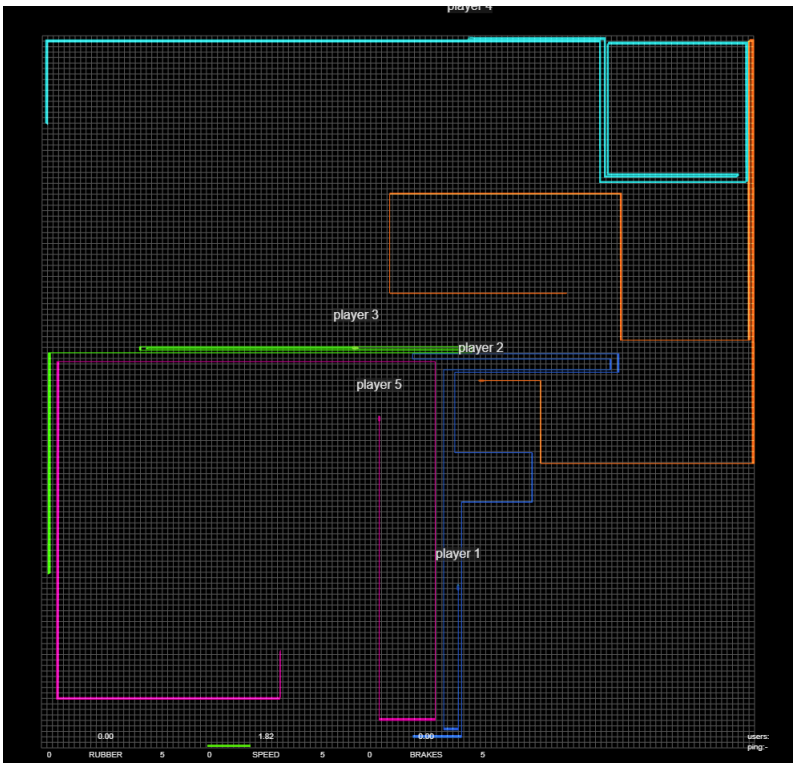

Figure 2.1: GLtron is like a multiplayer game of Snake.

that uses OpenGL graphics and it has been ported to WebGL by Darren Prentice under the MIT License. [15] Next, we stripped the project of everything it does not really need, like a welcome screen or multiple camera positions. We disabled the local AI contestants and enabled AI for the player so that no input was necessary after visiting the appropriate site. As a side effect, this means that the game can be played on mobile devices without a keyboard.

2.2 The client

The first step in the sequence as displayed in Figure 2.2 is the client sending out a request to the Apache server to which the server responds by sending out the HTML and the client side JavaScript. As discussed in Chapter 1, we will implement WebGL.

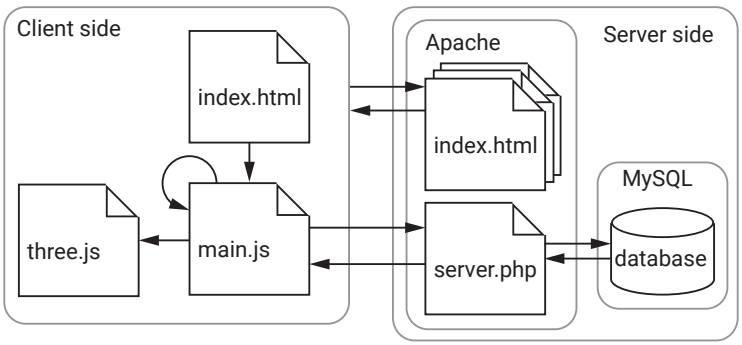

Figure 2.2: A simplified display of the connection between client and server

Programming WebGL directly from JavaScript to create and animate 3D scenes is a very complex and error-prone process. Three.js is a library that makes this a lot easier. [16] So we will be using Three.js revision 71 as an easier to use library for WebGL.

2.3 The server

The server needs to update and store every client's current situation and share this information with every other client in the tiniest block of data possible. [17]

2.3.1 Apache

The first implementation of the server is an Apache httpd server. As Apache is the most popular HTTP server software since 1996 [18], it will serve as our proof of concept with PHP code. We implemented Apache httpd version 2.4.28 in its out-of-the-box form with a PHP7 module. We did not use PHP-FPM in its configuration.

2.3.2 Node.js

Node.js supposedly performs better in web applications [19] than Apache. We will be trying out how this holds up. We have rewritten the PHP code to JavaScript to work with Node.js version 9.5.0, however the entire architecture is as similar as possible to the PHP code, with the exception of one thing: generating output is the first step for the JavaScript, while it was the last step for the PHP script.

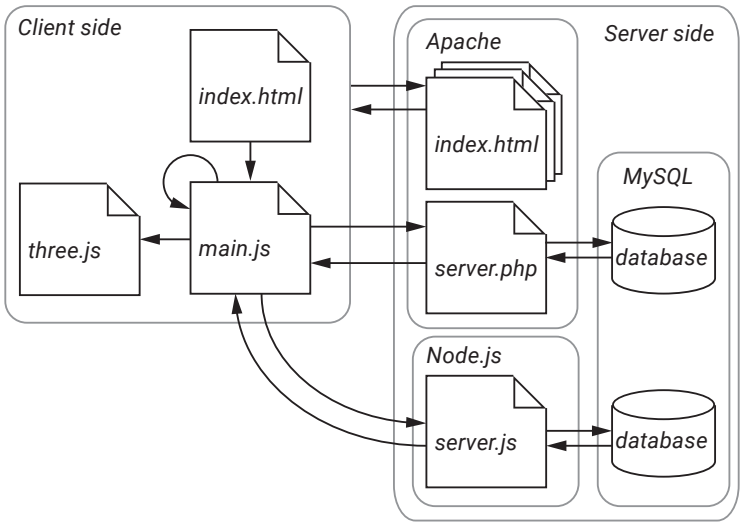

Figure 2.3: Node.js included in the architecture

To switch between servers the client would simply use a boolean to know which server to use. The architecture as described in Figure 2.2 changes with the inclusion of this server to the architecture described in Figure 2.3. We did not use Node.js to serve pages of the game, because we are not trying to measure the loading of the page itself and serving static pages is not one of Node.js' strengths. [20]

We used the 'http' module for the server with the 'mysql' module for the database and the 'querystring' module for parsing the POST requests. Unlike its PHP counterpart, Node.js runs the MySQL commands asynchronously so we had to take that into account when writing for the database. [21] We could not use Socket.IO to handle the connections because we wanted the same code of the client to be able to connect to both servers, even though it is has proven to work well for MMOG's. [8] [5] [22] PHP does support WebSockets [23], but Apache does not.

2.3.3 MySQL

The majority of the server sided scripts consists of communication with the MySQL database, a popular relational database. [24] The database stores as much information as possible. It contains the tables 'players' and 'walls' for the game, as well as 'lag' for monitoring purposes. The command most executed is a command with which the player updates its information and expects information about its world in return. This command creates 4 SQL commands (2 update commands, 1 insert command and 1 select command) which are sent to to the MySQL server.

We are running MySQL version 14.14.

2.4 Combining the two

WebGL uses a main loop to refresh the screen at most 60 times per second and performs calculations between each of these frame refreshes. A second loop handles the communication with the server. In the beginning a request is sent to the server asking for configuration files and to reserve a character and location for the player. The server sends this to the client in XML format. This, in turn, initiates the rest of the loop. The client sends an update to the server in a POST request, the Apache server updates the database and sends an update to the client in XML format. The Node.js server does the same but sends the update to the client first and then updates the database. The client updates its variables and the loop continues. The time between sending two updates is defined as the lag of the player. Every time a status update is requested, the lag of the requester is stored. If there is no response from the server after five seconds, the client assumes its message is lost. This would break the loop, so we use the main loop and send another update in an attempt to get the loop going again. This would mean that during the first screen refresh after 5 seconds of lag, the recorded lag would reset, never surpassing the 5 second mark for more than one frame. The server assumes the connection is lost after ten seconds of inactivity and marks the player as dead, meaning its character will explode in the in-game worlds of all contestants. The client-sided JavaScript file calculates where each player is based on their speed, position and direction since the last received packet from the server.

It may seem more logical to define lag as the time between receiving two requests instead of the time between sending two requests, as this more closely records the experiences of the player. However, we wanted to align all the datapoints in one dataframe which is sent to the server. If we send the lag the player has experienced

when it received the previous dataframe the newly recorded lag is still incomplete (halfway in between receiving two dataframes) and the previous recorded lag has already aged.

There are plenty of optimizations we can make to both the server and the client side to make the server load lighter and make the connection use less bandwidth. [25] [26] [27] We didn't do this however, because we want to compare the performance of the servers in a worst-case scenario. Optimising the performance of this game would only make it more difficult to achieve this worst case scenario.

2.5 Issues

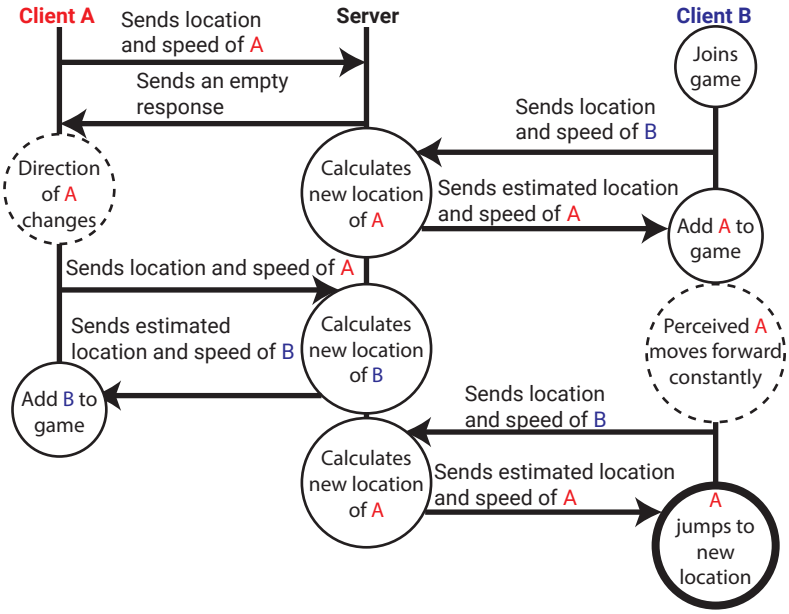

Figure 2.4: An example of communications with the server, showing how we get positional inaccuracy

One problem with this method was that one device may update faster than the other. If device A sends and receives two updates, while device B has not sent an update in between, device A will update its version of device B back to its original position. So, the server must also calculate where each player is based on its time since the last update, so it can generate some new information for device A. The server may sometimes guess wrong which forces a retraction as illustrated in Figure 2.4, but assuming that a player does not change its direction or speed is a safer bet than assuming the player has not moved. [28] Also, the game would draw the clients contestants in motion for 60 frames per second, while server updates would come at most 20 times per second. The time in between is filled in by the client. We ended up using a technique called dead reckoning with time compensation. Dead reckoning as such, could be described as a technique that will decide where an object should be positioned "now" based on where it has been positioned "before" and which velocity it had. [29] More concrete in our game, after a player would upload its location, the server stores the time. All requests for that players location would get an estimate based on the time.

Figure 2.5 shows how this can go wrong. In this example the player has turned and has not updated the server yet. This means that the perceived path and the actual path diverge. The server may even keep sending

estimates to clients based on old information. Eventually the server will receive the update and correct each client with the new direction. For this correction a positional inaccuracy is calculated, which is equal to $distance(x) + distance(y)$.

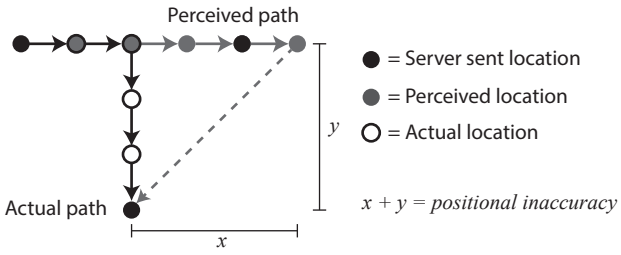

Figure 2.5: Faulty estimations may lead to inaccuracies.

Another problem we encountered was that Three.js appeared to have a data leak. After running an instance of the game for some time (about 20 minutes) WebGL would sometimes crash. Although this leaves behind a white screen for the player, the JavaScript behind the game is still sending, receiving and updating information. To prevent this from happening we altered the re-spawn code to renew the entire window, thereby getting rid of all reserved variables and stored data. This leads to a lost connection with the server for about three seconds in which nothing is recorded by the server.

2.6 Other versions

After we made the 'fun' version of the game, we also made an alternative version of the game, which we made as bare-boned as possible. We started with a clone of the Tron game and servers. The differences are: the characters don't leave any walls; the characters can not die and the characters can stand still. We have programmed the AI to have the choice to go left, go right or stand still. The odds that the AI chooses to stand still is a parameter we can change during the experiment. The fact that players can not die means that contrary to the Tron game, players will constantly need to be connected to the server. The player can die in the worlds of its contestants when the server has not received updates for 10 seconds. We also included the calculations of positional inaccuracy in this version of the game. The reason we did not include these calculations in the Tron version is only meant as a proof of concept and because the speeds in the Tron version vary significantly (depending on how many turns the character made), this would distort the data.

In order to fully overload the server, we also made fake clients without any graphics. By just opening these clients in a couple of tabs we can multiply our active connections without being as resource-heavy on the client. The client only displays the current lag, and location statistics, but still sends and receives the same amount of data. It does nothing with the data it receives about other players. These players are visualised in the game for the other players as players that are standing still. All of this enables us to use multiple tabs and therefore connections at the same time on a single machine.

Chapter 3

Experiments

In this chapter we discuss the experiments conducted in order to get to an answer to the research question once the implementation discussed in Chapter 2 was sufficiently finished.

3.1 The set-up

We used at most 44 desktop Windows PC's with 4 GB of memory and Core i5 650 processors clocked at 3.33GHz and Core i3 processors clocked at 2.93 GHz. The browsers used were Chrome on 36 PC's and Internet Explorer on 8 PC's.

For the server, we used a dedicated server with an Intel Core i5-6500 CPU clocked at 3.20GHz and 15GiB of memory. The Ethernet connection has a data rate of 1 Gbps. The server is running Linux 4.4.0. We will compare two different implementations of the server-side in order to compare the two later on.

We did not change the PC's or the server throughout these experiments.

The size of the server is the biggest difference compared to large scale industry games like World of Warcraft which servers have over 75,000 CPU cores. [30]

3.2 Proof of concept

In order to prove that the game we developed works as a game as well as being capable of handling a large number of players, we first started to experiment on the Apache server with the full game. We had the server set up to accept an unlimited number of connections, although the play world was limited but large enough for around 400 players.

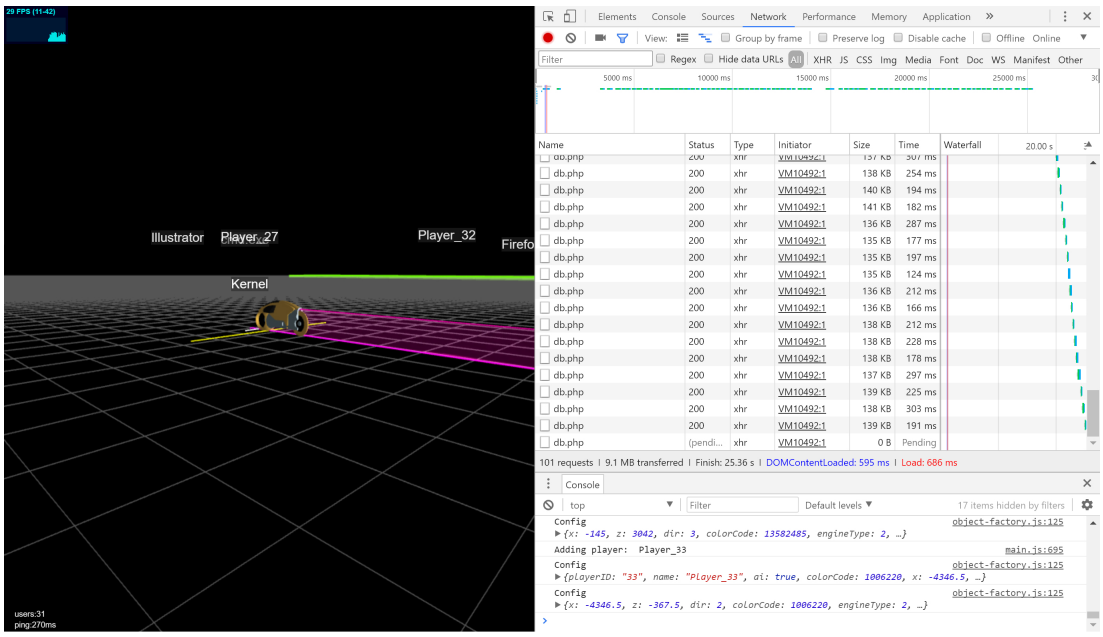

Figure 3.1: Over the span of 25,36 seconds a client would collect 9,1 MB of data from the server, spread out over 101 requests.

We expected we could run multiple instances of the same game on a single machine. However, in practice we noticed that around 45% were inactive at any given time, which appeared to be caused by limited memory on the client's PC's.

Eventually though, we had a pretty constant number of 30 active players in the game world at one time, by only running one instance of the game per machine. The reason we could not get this number constant lies in the nature of the game. Whenever a character died, it would disconnect from the server. That is why we made another version of the game.

3.3 Controlled comparison

While the first experiment proves that an MMOG is possible with the setup and server as we made it, there were some things that disrupted the test results. That is why we made a second version of the game without the walls, where the characters can not die, and the characters can stand still. We will be focusing on this version of the game for the remainder of this chapter.

For the purpose of this experiment we varied between the number of players and the percentage of players that is moving to compare the average lag and the error rate of the Apache server and the Node.js server. We focused on 5, 10, 15 and 20 players and 50%, 25% and 5% of the players moving. With every variation we waited for at least 5 minutes to make sure that we had gotten a reading of the lag, positional inaccuracy and other statistics without too much noise.

It should be noted that it took multiple attempts with Node.js to get the server to maintain 20 players for 5 minutes.

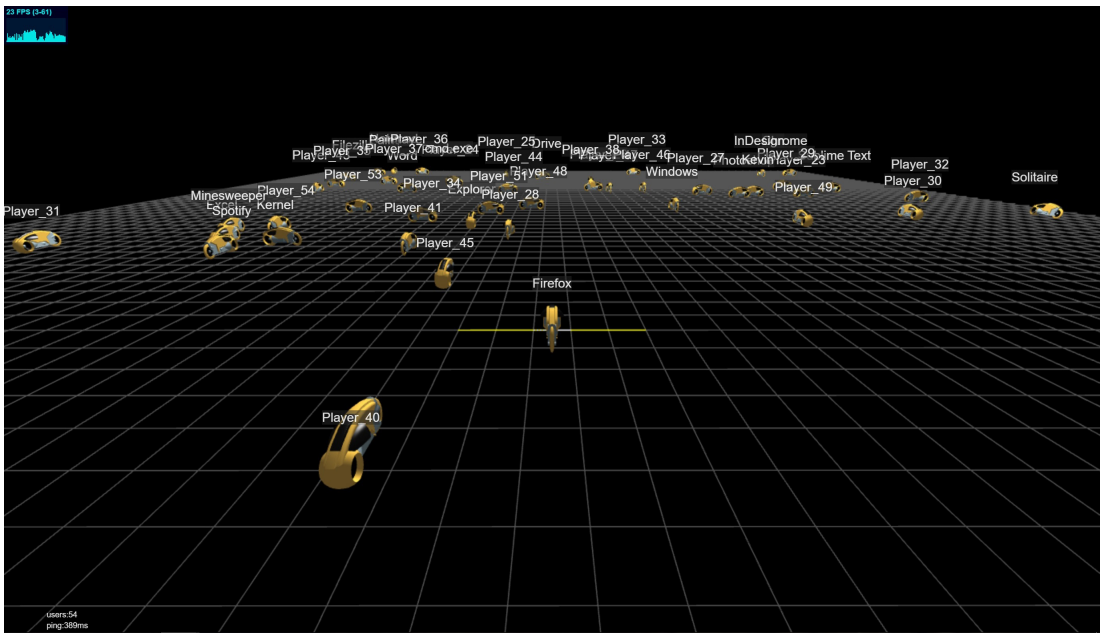

Figure 3.2: In this version, players do not leave walls and have the ability of standing still.

3.4 Stressing the server

In the next experiment we wanted to get a maximum number of connections. While maintaining the same server, we used the fake clients without any 3D graphics. That way we can maximise the number of connections per machine. In our setup, all the regular clients with graphics are constantly moving and all the fake clients are frozen, only changing the direction they are facing.

3.4.1 Node.js

We started this experiment with the Node.js server. After we connected 29 PC's to the game with regular clients, we added the fake clients to spam the servers. In total there was a maximum of 38 clients in the game at one time when we noticed the game would not load any new clients.

The lag of the regular clients went constantly above 5000 ms, which means that the client is already assuming the connection is lost when it finally receives its signal. The result was an unplayable experience. Players stopped receiving any visible updates and new clients could not join the game.

3.4.2 Apache

We connected all the clients with the Apache server.

After the 29 PC's were connected with their games, we started adding the fake clients. After 5 minutes there were a total of 69 connected clients. By this time, we noticed that while the game was still perfectly playable,

we were not able to add any new players. A new client simply could not make a call to the server to receive its location and player data.

After analysis we found out that the game would not allow new players to connect because of a bug in the creation of new characters past the range of characters already in the database. After fixing this bug, we tried again.

This time we managed to add 121 fake clients. After we started adding more fake clients, the regular clients started to crash. In 5 minutes we had a total of 156 fake clients and 0 regular clients.

As a final attempt, we altered the server so that it would not send the fake characters to the regular clients, hoping that the clients would not crash that way. We managed to get a maximum of 192 fake clients and 10 regular clients in the game at one time. This was not a stable number and we could only hold onto this number for two minutes before clients started dropping out. The regular clients showed that they did still receive updates from the server but began showing large bugs. The number of active players in any one of the games varied significantly, meaning players would pop up and disappear. Half of the regular clients crashed in the 5 minutes after the previously mentioned maximum, but this could be due to the fact that the machines upon which the regular clients were running also hosted 5 to 10 fake clients.

Chapter 4

Results

In this chapter we provide the results of all three experiments first mentioned in Chapter 3.

4.1 The proof of concept

The first part of this experiment, we had an average lag of 265,76 ms with 30 characters in the game.

For the second part, where all machines only hosted one instance of the game, the average lag was 267,47 ms with 32 characters in the game.

4.2 Controlled comparison

In Table 4.1 all of the results from the experiments are presented. Each row corresponds with different parameters with the parameters to the left, the results from Node.js in the centre and the results in the Apache server with the same parameter at the right.

The number of players moving is the number of players moving at any given time. For example, we did not assign 50% of the players the ability to move while the other 50% was just standing still, rather the players all had a 50% chance of choosing to stand still.

Although we had a lot of results we could draw information from, these seemed to relay the experience of the player the best. The lag, or ping, is the time between 2 requests made by the client. Because the client initiates a new request after 5 seconds of inactivity, the maximum lag recorded is 5000 ms. The positional inaccuracy is the sum of every error perceived by a player divided by the number of other players.

For the lag columns, the higher the number the worse the perceived lag was for players, which is undesirable. For the positional inaccuracy, the number corresponds with the average distance the client had to correct every

Table 4.1: The results

		Node.js		Apache	
Number of players	Number of players moving	Lag (ms)	Positional inaccuracy per player	Lag (ms)	Positional inaccuracy per player
5	5%	355,6	0,8911	191,2	0,3443
5	25%	338,4	1,192	190,1	0,4835
5	50%	337,9	2,657	173,9	0,6965
10	5%	689,1	1,205	191,5	0,1086
10	25%	679,4	5,453	191,9	0,5376
10	50%	673,4	9,248	192,0	0,9675
15	5%	1021	2,402	193,0	0,08224
15	25%	1063	10,87	201,9	0,5484
15	50%	1037	20,91	209,2	1,073
20	5%	2100	5,430	200,1	0,08563
20	25%	2198	27,26	202,0	0,5984
20	50%	1630	37,98	205,6	1,089

player in its perception of the world. The higher this number, the more the client had to correct its world, which is undesirable.

4.3 Stressing the server

Table 4.2 lays out the results from our attempts at overloading the server. The top row is the point at which both servers had 29 active players. This was moments before we added the fake players. The second row is the highest recorded number of players for Node.js. The third row is the moment Apache reached a similar moment as Node.js did. Note that the number of regular players is still the same as when it started for Apache, while Node.js has already 9 clients at that time. The fourth row is the highest number of clients Apache recorded, before all of its regular clients crashed. The bottom row is the highest recorded number of players for Apache, but when the fake clients were not visible for the regular clients. This may have influenced the lag and the positional inaccuracy.

Table 4.2: The results of our attempts at overloading the server

		Node.js		Apache	
Number of fake players	Number of regular players	Lag (ms)	Positional inaccuracy per regular player	Lag (ms)	Positional inaccuracy per regular player
0	29	2639	263,1	211,0	57,97
18	20	5127	288,6		
18	29			514,9	19,95
120	25			873,4	65,42
192	10			973,5	31,82

For Node.js, recording its server overloading was particularly difficult, because its positional inaccuracy was fluctuating a lot. The five minutes which averaged out to the second row, we recorded the lag had one minute in which the average positional inaccuracy was 1272 and one minute before that, it was just 64.

Just like in the previous experiment the Node.js server was handling a maximum 889 requests per minute for games with less than 20 players with a lag of 1302 ms. Although this is a lot, that number doubled when we

started adding more players, to an average of 2651 ms with 25 players. Apache had an average lag of 203,6 ms with 25 players.

Chapter 5

Analysis & Discussion

In this chapter we review the results provided in Chapter 4. We will first discuss the overall results, before going more in-depth for each individual experiment.

5.1 Overall

5.1.1 Summary

Overall, we found that Node.js performed weaker than expected. In the controlled comparison between both servers we found that by the time Node.js had sent 1 frame of data, Apache would have sent between 1,780 and 10,88 frames on average, depending on the amount of players.

We see similar performance differences when we were stressing the server. While Apache definitely had some performance issues when going over 200 players, Node.js never got near that number. The lag of Node.js went over 5000 ms with just 38 players in the game.

5.1.2 Expectations

These results contradict our expectations based on the research done on this topic. Node.js' strength is its non-blocking coding structure. Although we did utilise this, it seemed insufficient to outperform Apache, which is best at serving static pages quickly.

Previous researchers found that Node.js worked better with web applications than Apache. A big difference between our implementation and these previous implementations is the use of Socket.IO to handle the connections. This is the most probable cause for the difference between our research and previous work.

As for the difference between Node.js and Apache, while we did not monitor this very closely, we did notice CPU usage is considerably higher for Node.js than it is for Apache.

5.1.3 Future work

We did not measure which settings in Apache or Node.js would best be suited for the purpose of running an MMOG. In stead we opted for the most out-of-the-box approach with comparable settings for both servers, that could power the same MMOG on both servers. Future research could benefit from a comparison study with optimal settings for both Apache and Node.js, and tweaking the MMOG accordingly.

5.2 The proof of concept

In our first experiment, the numbers were actually quite positive. With a difference of just 1,71 ms, the difference in lag of the first part of the experiment and the second part is pretty small. Lag however, was not the reason we had to switch up the set-up. The reason was the high drop-out rate, which we could not accurately measure.

5.3 Controlled comparison

There is a significant difference between Node.js and Apache. On every row we see that Node.js is performing worse than Apache. Overall, we can see that the more players are in the game the higher the lag. This difference is far more significant for Node.js than it is for the Apache server.

Remarkably, the positional inaccuracy grows with the number of players that are moving as well as with the number of players that are in the game. We would expect the positional inaccuracy of 25% of 20 players moving and 50% of 10 players moving to be equal, but these two rows are not similar at all.

If we also take a look at the total number of requests the server handles per minute (which can be calculated by $\frac{NumberOfPlayers}{LagInMinutes}$, or, like we did, just by looking at the number of rows in the database at any given time) we get Table 5.1. This table shows the main difference between Node.js and Apache we experienced. Where Node.js sits comfortably between 868,1 and 880,4 requests per minute for less than 16 players, the number of requests the Apache server is handling is multiplying with the number of players. It seems like Node.js has reached its limits immediately with just 5 players, and its limits are not very high. It even lowers this limit for 20 active players.

Table 5.1: Requests per minute

Number of players	Node.js' requests per minute	Apache's requests per minute
5	877,9	1601
10	880,4	3118
15	868,1	4473
20	676,4	5843

We repeated this part of the experiment to check these numbers and Node.js keeps lowering the number of requests it handles above 19 players. This is important because it means that the more clients are added to the game, the less updates each client receives. The client assumes the connection to be lost after 5 seconds. If we assume this is the threshold, a client needs to connect to the server once every 5 seconds, or 12 times per minute. If the server only has 880 requests it can handle, the theoretical maximum number of players would be $\frac{880}{12} \approx 73$ players. Of course, in practice this number would be much lower, as we have already seen in Table 5.1 for 20 players.

5.4 Stressing the server

The lag on Node.js with 18 fake players is very high. We have already stated repeatedly that the lag could not normally go above 5000 ms, because after 5 seconds the client would start sending a new request because it assumes the previous request would not get a response. It seems that the clients were slowed down so much that even the new response took 127 ms to generate on average.

While performing this experiment, we noticed a big difference between Node.js and Apache. Where the Node.js server overloaded with just 38 players, the Apache server never actually overloaded. The first time we experimented with a large number of players in Apache we encountered a bug in the code, the second time the clients crashed and the third time we pushed the clients as hard as we could, but the server still accepted new connections. These numbers confirm that Apache never experiences as much lag as Node.js does.

Chapter 6

Conclusions

In this paper we have looked at a possible implementation of WebGL: as the graphics processor for a browser-based Massively Multiplayer Online Game. We have proven that it is definitely possible to use WebGL and a simple LAMP stack server (Linux, Apache, MySQL and PHP) to create a game that hosts more then 100 clients. This may not seem like enough to create a game hosting a world for millions of players, but this was achieved on a small server and without optimizing the traffic between server and client, contrary to the professional game developers

We have also looked at the option of implementing Node.js with JavaScript instead of implementing Apache with PHP, but the server would not load more then 38 players into the game, whereas the Apache server hosted a game for more than 200 players at the same time. This makes the Apache server a better choice in the situation we tested.

Bibliography

[1] Abdennour El Rhalibi, Madjid Merabti, Chris Carter, Chris Dennett, Simon Cooper, M. Ariff Sabri, and Paul Fergus. 3d java web-based games development and deployment. In *Multimedia Computing and Systems, 2009. ICMCS'09. International Conference on*, pages 553–559. IEEE, 2009.

[2] John Congote, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar Ruiz. Interactive visualization of volumetric data with webgl in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, pages 137–146. ACM, 2011.

[3] Rama C Hoetzlein. Graphics performance in rich internet applications. *IEEE computer graphics and applications*, 32(5):98–104, 2012.

[4] Sixto Ortiz Jr. Is 3d finally ready for the web? *Computer*, 43(1), 2010.

[5] Bijin Chen and Zhiqi Xu. A framework for browser-based multiplayer online games using webgl and websocket. In *Multimedia Technology (ICMT), 2011 International Conference on*, pages 471–474. IEEE, 2011.

[6] Jens Geelhaar and Gabriel Rausch. 3d web applications in e-commerce-a secondary study on the impact of 3d product presentations created with html5 and webgl. In *Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on*, pages 379–382. IEEE, 2015.

[7] Ed Angel. The case for teaching computer graphics with webgl: A 25-year perspective. *IEEE computer graphics and applications*, 37(2):106–112, 2017.

[8] Makzan. *HTML5 Games Development by Example : Beginner's Guide: Create Six Fun Games Using the Latest HTML5, Canvas, CSS, and JavaScript Techniques.* Community Experience Distilled. Packt Publishing, 2011.

[9] Alex Cocilova. The future of video games is in your browser. *PC World*, 31(12):28, 2013.

[10] Runescape breaks through one million subscriber mark. *PR Newswire*, 3 2007.

[11] Blizzard: World of warcraft exceeds 12 million subscribers. *Wireless News*, 10 2010.

[12] Xinbo Jiang, Farzad Safaei, and Paul Boustead. An approach to achieve scalability through a structured peer-to-peer network for massively multiplayer online role playing games. *Computer Communications*, 30(16):3075–3084, 2007.

[13] Peter Cohen. The game room. *Macworld*, 18(9):49, 2001.

[14] Egor Kuryanovich, Shy Shalom, Russell Goldenberg, Mathias Paumgarten, David Strauß, Seb Lee-Delisle, Gaëtan Renaudeau, Jonas Wagner, Jonathan Bergknoff, Brian Danchilla, et al. Cycleblob: A webgl lightcycle game. In *HTML5 Games Most Wanted*, pages 175–211. Springer, 2012.

[15] Darren Prentice. WebGL fundamentals. `www.github.com/dpren/WebGL-Tron`, July 2015. Accessed: April 26th 2017.

[16] Jos Dirksen. *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt Publishing, 2013.

[17] Tsung Teng Chen. Online games: Research perspective and framework. *Computers in Entertainment (CIE)*, 12(1):3, 2014.

[18] Roy T. Fielding, Richard N. Taylor, Justin R. Erenkrantz, Michael M. Gorlick, Jim Whitehead, Rohit Khare, and Peyman Oreizy. Reflections on the REST architectural style and principled design of the modern web architecture. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, page 5. ACM, 2017.

[19] T.J. Holowaychuk Nathan Rajlich Mike Cantelon, Marc Harter. *Node.js in Action*. Manning Publications Co., 2014.

[20] I.K. Chaniotis, K.I.D. Kyriakou, and N.D. Tselikas. Is node.js a viable option for building modern web applications? a performance evaluation study. *Computing*, 97:1023–1044, 10 2015.

[21] Colin J. Ihrig. *Pro Node.js for Developers*. Apress, 2013.

[22] Isaac Sukin. *Game Development with Three.js*. Packt Publishing Ltd, 2013.

[23] J Calvo, J Gracia, and E Bayo. Robust design to optimize client–server bi-directional communication for structural analysis web applications or services. *Advances in Engineering Software*, 112:136–146, 2017.

[24] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[25] Vlad Nae, Alexandru Iosup, Stefan Podlipnig, Radu Prodan, Dick Epema, and Thomas Fahringer. Efficient management of data center resources for massively multiplayer online games. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 10. IEEE Press, 2008.

[26] Tiago Alves and Jorge Coelho. A framework for massively multiplayer online game content generation. In *Advanced Information Networking and Applications (AINA), 2016 IEEE 30th International Conference on*, pages 834–841. IEEE, 2016.

[27] Bart De Vleeschauwer, Bruno Van Den Bossche, Tom Verdickt, Filip De Turck, Bart Dhoedt, and Piet Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–7. ACM, 2005.

[28] Sladjan Bogojevic and Mohsen Kazemzadeh. The architecture of massive multiplayer online games. Master's thesis, Lund University, 2003.

[29] Xiang-Bin Shi. A DR algorithm based on artificial potential field method. *Multimedia Tools and Applications*, 45(1):247–261, 2009.

[30] James Dean Mathias. *Peer-to-peer simulation of massive virtual environments*. Utah State University, 2012.