Bachelor of Science in Computer Science
June 2018

Testing scalability of cloud gaming for multiplayer game

Dan Printzell

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Dan Printzell
E-mail: dapf15@student.bth.se

University advisor:
PhD, Prashant Goswami
Department of Creative Technologies (DIKR)

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. The rendering of games takes a lot of processing power and requires expensive hardware to be able to perform this task in a real-time with an acceptable frame-rate. Games often also require an anti-cheat system that require extra power to be able to always verify that the game has not been modified. With the help of game streaming these disadvantages could be removed from the clients.
Objectives. The objective of this thesis is to create a game streaming server and client to see if a game streaming server could scale with the amount of cores it has access to.
Methods. The research question will be answered using the implementation methodology, and an experiment will be conducted using that implementation. Two programs are implemented, the server program and the client program.

The servers implement the management of clients, the game logic, the rendering and the compression. Each client can only be connected to one server and the server and its clients live inside of a game instance. Everyone that is connected to one server play on the same instance.

The implementation is implemented in the D programming language, and it uses the ZLib and the SDL2 libraries as the building blocks. With all of these connected an experiment is designed where as many clients as possible connect to the server. With this data a plot is create in the result section.
Results. The output data shows that the implementation scale and a formula was made-up to match the scalability. The formula is $f(x) = 8 + 5x - 0.11x^2$.
Conclusions. The experiment was successful and showed that the game server successfully scaled based on the number of cores that where allocated. It does not scale as good as expected, but it is still an success. The test results are limited as it was only tested on one setup. More research is needed to test it on more hardware and to be able find more optimized implementations.

Keywords: Cloud Gaming, Scalability, Network, Software Rendering

Acknowledgments

I would like to thank Dr. Prashant Goswami for all the help I received during the whole thesis work. I also want to thank Olof Christensson and Adam Georgsson for the all the feedback I received from the opposition.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

An idea that has been popping up in the game industry is to run games on a powerful workstation and then stream the rendered frames to a low powered device. One example of this is Nvidia GeForce NOW™. It is a service where you can rent server time, to stream a high demanding game, over the internet, to their low powered handheld console. Another example is Steam In-Home Streaming. The concept is the same, the game is played on a workstation, but instead of renting server time, you stream the game from a computer in your home to another computer. Both services work the same. The computer that is playing the game, referred later as the server, will run the game logic, render and compress the frames, to send them to the target computer or handheld console, later referred as the client.

There has been research into how different games perform when played with different services. In the paper by Quadrio et al. [10], they researched how Steam In-Home streaming worked on three different games, with different levels of graphical fidelity. They measure both the network traffic and the difference in frames-per-second on the two computers. Sadly, their test case was really limited, they only tried 3D games, all with medium to very high graphical complexity. As 2D games often have simple graphics, the compression code for in-game streaming would operate better and would be able to output a smaller compressed version.

Liang Cheng et al. in the research "Real-time 3d Graphics Streaming Using Mpeg-4" [6] proposed a way of using their algorithm as input into the MPEG-4 compression code to optimize the compression. Like the other, this one also used a 3D rendered environment as the input image. Also like the other this one was targeted towards only one application instance per server.

In the research from Yeng-Ting Lee et al. [13], they talked about that not all games are good candidates to be streamed. With the help of Quality of Experience model they used, they could figure out what games are good candidates. For example, games where low latency is a must are not great candidates. One game that is a good candidate is Blackjack, as it does not require quick actions.

All the previous mentioned research papers limit their research by only talking about streaming a single game per computer. They did not research streaming multiple games per machine. In the field, this is exactly what Nvidia doing in their NVIDIA GRID servers [3]. These servers can stream up to 48 HD-quality games, per server. This shows that with customized hardware and software it is possible to implement a scalable system to do cloud rendering of games.

1.2 Aim and Objectives

The goal of this project is not to create a game but rather to create a scalable cloud streaming system, that in this case implements a multiplayer Blackjack game. The game is just to fill the need for something to be tested against. The server will handle the game logic and the rendering. The client's job is to decode the incoming frames from the server and display them, but also send game interactions to the server. The metrics that will be logged is how many players that are currently connected, how much data that is sent over all the network sockets and how much memory the program is using. The goal is to make sure that this system scales based on the number of CPU core the program is allowed to use.

1.2.1 Objectives

- Setup a frame rendering on the server.
- Setup frame displaying on the client.
- Communication layer between the server and the clients, using non-blocking TCP sockets.
- Implement image compression using an external library.
- Implement the game rules for Blackjack.
- Implement metrics outputs.
- Run the test to gain metrics.

1.3 Research Question

Is it possible to implement a scalable, software rendered, multiplayer Blackjack game, that scales depending on the number of CPU cores allocated?

1.4 Chapter Overview

Chapter 2 presents the information that is needed to understand the rest of the thesis. This chapter also talks about related works.

Chapter 3 talks about the implementation, for example what concepts it uses, what libraries it uses and how they are used. But also about how the experiment will be conducted.

Chapter 4 presents the result that is gained from the experiment.

Chapter 5 analyze the data gained from the result to try and get some more higher levels concepts from it. It is also here where it is proved that the result is not bottlenecked in the wrong places.

Lastly, in Chapter 6 it is concluded an answer to the research question. But also about future work that could be done to gain more knowledge about this field.

Chapter 2

Background and Related Work

2.1 Background

The rendering of games takes a lot of processing power and requires expensive hardware to be able to perform this task in real-time with an acceptable frame-rate. Games often also require an anti-cheat system that require extra power to always verify that the game has not been modified. With the help of game streaming these disadvantages could be removed from the clients.

To be able to understand the game streaming implementation that is explained later in the thesis a few concepts are needed to be explained.

2.1.1 The D Programming Language

The D programming language is a relatively new programming language, when you compare it to languages like C and C++. The language is created by Walter Bright, a compiler expert, and Andrei Alexandrescu, author and speaker in the C++ community. Walter Brights first published work was back in 1989 called *Secrets Of Compiler Optimization* [2]. Andrei Alexandrescu and Herb Sutter have published a book called *C++ coding standards: 101 rules, guidelines, and best practices* [11].

Both the authors have publish works on the D language: Walter Bright have a presentation called *The D programming language* [3]. It is an small introduction to D. It present the feature that D have and explains how C code could look like if it was ported to D. Andrei Alexandrescu have published a book called *The D programming language* [1]. It is a book that teaches you the D programming language from scratch.

D has not been used in many research works. One example of a research work that uses the D language is a research work from N. Chouhan et al. called *A Code Analysis Base Regression Test Selection Technique for D Programming Language* [7]. In this work they explain both in the abstract and in the introduction the appeal of the programming language. The summary of what they said is that D is a language that have the feel of C++ but with more modern language technique added. Some example they mentions is the automatic memory management, in D you don't need to care about memory leak as the memory management system will manage this for you. The second example they mention is the module system, similar to language like Java where you only have source files and no header files, like you have in both C and C++.

The D programming language was chosen as the implementation language as its standard library contains code that will help the implementation to be as concise

and readable as possible. Another reason why D was chosen is because the author have previous knowledge of the language, which really helps the implementation. In the end, the usage of D will not hurt the implementation. It will only help it.

2.1.2 Software rendering

A rendering technique that is not often used in modern games is software rendering. This rendering way does not use a graphics cards, rather it uses the processor to create the final image. The big advantage of software rendering over accelerated rendering, which implies the usage of an accelerator card often a GPU, is that it could be done on any type of processor. Another advantage of software rendering is that processor often have multiple cores, and each core could work on rendering task in parallel.

One implementation, and the one this implemention will use is, Simple Direct-Media Layer, later referred to as SDL. SDL define concepts like pixels, surfaces, filling and blitting. A pixel is a two-dimensional box that has the size of one by one size units that contains a single color. This pixel type could store the color in many memory layouts. Each memory layout has advantages and disadvantage. An example would be RGB888. This means that the color information will be split into three different parts, also referred as channels, The red, green and blue channel. Each channel will have eight bits of information. A surface, in the layman's term an image, is a collection of pixel in a two-dimensional grid. Filling is a rendering technique where you select a region in the surface and fill each of the pixels in that region with specified color. Lastly, blitting is where you copy a region of pixels from one surface to a region on the same or a different surface. This technique can be explained as copy-pasting.

SDL is not the only implementation that was considered. Another implementation of software rendering, that in written D, is CPUblit by László Szerémi [12]. It uses CPU extensions to accelerate the rendering, the same as SDL does in their backend. This implementation was not chosen as it is not as feature complete as SDL, but also as SDL will be used to manage the input and the game window. It is more optimize to use SDL for everything instead of mixing different libraries.

2.2 Related work

There has been research into how different games perform when played with different services. In the paper by Quadrio et al. [10], they researched how Steam In-Home streaming worked on three different games, with different levels of graphical fidelity. They measure both the network traffic and the difference in frames-per-second on the two computers. Sadly, their test case was really limited, they only tried 3D games, all with medium to very high graphical complexity. As 2D games often have simple graphics the compression code in-game streaming would operate better and would be able to output a smaller compressed version.

In the research from Yeng-Ting Lee et al. [13], they talked about that not all games are good candidates to be streamed. With the help of Quality of Experience model they used, they could figure out what games are good candidates. For example,

games where low latency is a must, will not be great candidates. One game that is a good candidate is Blackjack, as it does not require quick actions.

Works that researched into 2D games could not be found. This could be because all the previous mentioned research papers limit their research by only talking about streaming a single game per computer. This could be because 2D game are generally not as rendering expensive, and thus would not make be needed in a home situation.

In the field there is a system that is streaming multiple games per computer. This is what Nvidia is doing in their NVIDIA GRID servers [8]. These servers can stream up to 48 HD-quality games, per server. This shows that with customized hardware and software it is possible to implement a scalable system to do cloud rendering of games.

Chapter 3

Method

The research question will be answered with using the implementation methodology, and an experiment will be conducted using that implementation. Two programs are implemented, the server program and the client program. The servers implement the management of clients, the game logic, rendering and compression. Each client can only be connected to one server and the server and its clients live inside of a game instance, as shown in Figure 3.1. Everyone that is connected to one server play on the same instance.

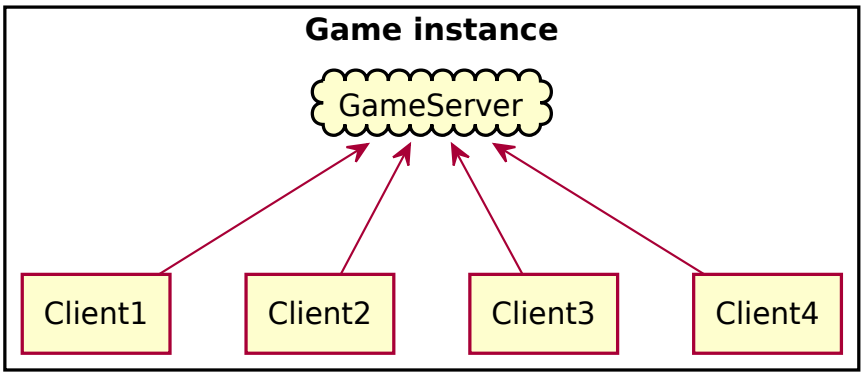

Figure 3.1: The relation between the clients and servers are as the following. Each client can only be connected to one server, and one server can be connected to multiple clients. Every server has their own game instance, thus everyone that is connected to that server will play on that game instance.

3.1 Implementation

The implementation source-code and textures can be found in the GitHub repo at the link in the appendix.

3.1.1 Server Implementation

The server has the biggest role in the whole environment. It is its job to handle the clients, the Blackjack game logic and the rendering, as shown in 3.2. It is required to achieve this task at atleast 30Hz. The implementation uses a few libraries to help minimize the code needed to be implemented and to not reinvent the wheel in places where there is already industry tested code.

The first library that is used is SDL. Its job in the server is to handle the framebuffers and do the blitting of textures onto the final frame. An example of an image that is rendered with the help of SDL can be seen in Figure 3.3. This image uses the `SDL_PIXELFORMAT_RGB332` pixel format, also called 8-bit TrueColor. This format instructs SDL to use only one byte per pixel. This is used as an optimization to decrease the size of the final image data. This frame will be compress using ZLib as describes later in section 3.1.3.

The last step in the client update is sending the client the newly rendered frame over a TCP socket. SDL_net, an extension for SDL, is used for this TCP socket. It is a library that adds a cross-platform API on top of the underlining operating system. This library is used to make sure that the server implementation needs to know as little as possible of any implementations quirks for the sockets, and thus taking time from the server implementation. TCP was chosen over UDP as the socket type, as it would furthermore limit the handling the server needs to do with the sockets. The server just needs to send data over the socket and it knows that the data will get to the client, in the correct order.

The scheduling of the client updates are schedules with the help of `TaskPool`, a parallelization helper class that is built into the D standard library [4]. Its job is to help schedule the update onto all the allocated CPU cores. Under the hood it works by launching one thread per each core allocated. Those threads will use mutexes to lock and check a global work list for work. The number of allocated CPU cores are specified when the server program is launched as a terminal parameter.

3.1.2 Client Implementation

The client has few but important tasks it need to do. Like the server implementation it also uses SDL, SDL_net and ZLib for its implementation. The clients job it to send keyboard presses to the server, and retrieve frames and display frame it gets from the server, as shown in Figure 3.4. Furthermore the client will display the state the game is in at the top of the screen, and at the bottom it shows if the user should input any action. In the game that is shown in Figure 3.3, it is shown at the top of the screen that is it the users turn to play the game, and at the bottom it shows the actions that can be done and what key to press to do the specified action.

Like to the server, the client also requires one command-line parameter. In this case it is the address to the server. Without this parameter it cannot connect to the server. The client also has an additional command-line parameter, called `nogui`, which you can specify if it should ignore the displaying of frames it receives from the server.

3.1.3 Compression and Decompression

The compression and decompression code that is used comes from the ZLib library. ZLib is an industry tested, open source, compression library that implemented the lossless DEFLATE algorithm. The DEFLATE algorithm is explained in the paper *A Fast Implementation of Deflate* by Harnik et al. [9]. They explain that the algorithm is a collection of other compression algorithms, specifically the LZ77 encoding and Huffman tree encoding.

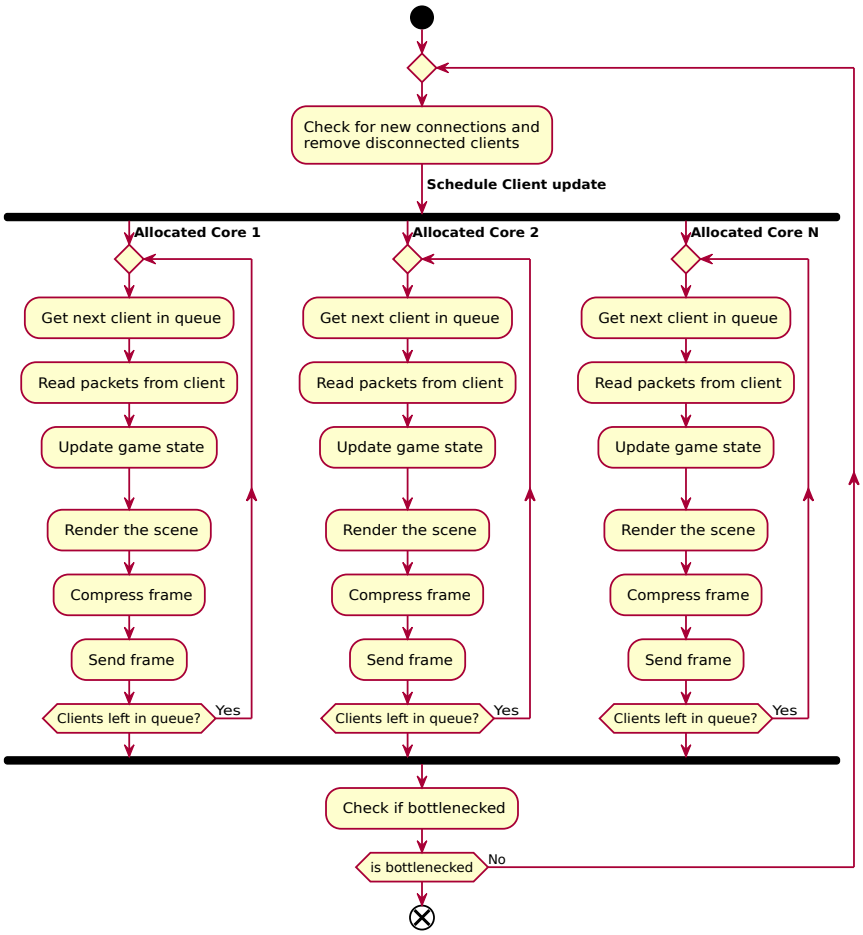

Figure 3.2: This represent the task-flow of the server application. Between the two big black horizontal lines are the core that will run on the allocated threads. The three paths are all the same and shows what each thread will do during the execution.

ZLib is used instead of a custom implemented algorithm because compression algorithms are a huge research field and this thesis is not focused on compression algorithms. Another reason why ZLib was used is because it is built into the D programming language standard library [5] and thus made it easier to use. These reasons are also the explanation to why no other compression algorithm was tested.

3.1.4 Synchronization

There is no synchronization between client to make sure that everyone sees the new frame at the same time, as shown in the Figure 3.2. The only synchronization that exist, is to make sure that every client will have the new frame before the main loops back to the beginning. As only one client at a time can do actions in the game this seems like a good way of implementing synchronization.

Figure 3.3: A frame of the game rendered by the server for the client that is currently playing. At the top it shows the current state the game is in and if an action is required. At the bottom it shows the potentially actions the user can do, and what button to press to do that action.

3.2 Limitations

3.2.1 Game

The content that will be streamed for the test, in this case a Blackjack game, is not a focus in this thesis. Its job is rather just to fill the purpose of something that is needed to be streamed.

3.2.2 Task scheduling

There are copious amounts of ways a program can implement a task scheduling system. Like the compression code, these algorithms have their own unique requirements to make sure that they are as optimal as possible. This is also a huge research field, and as a scheduling algorithm focus isn't required for this thesis it was chosen to use the preexisting scheduling system that is built into the D standard library [4].

3.2.3 Overall implementation

Due to all the different ways you can implement each part, where you can apply customized optimization, Et cetera. It was chosen that the implementation will reuse previously written code, that is, use libraries instead of writing all the code ourselves. This means that if this implementation success, all improved to the code should only improve the end result.

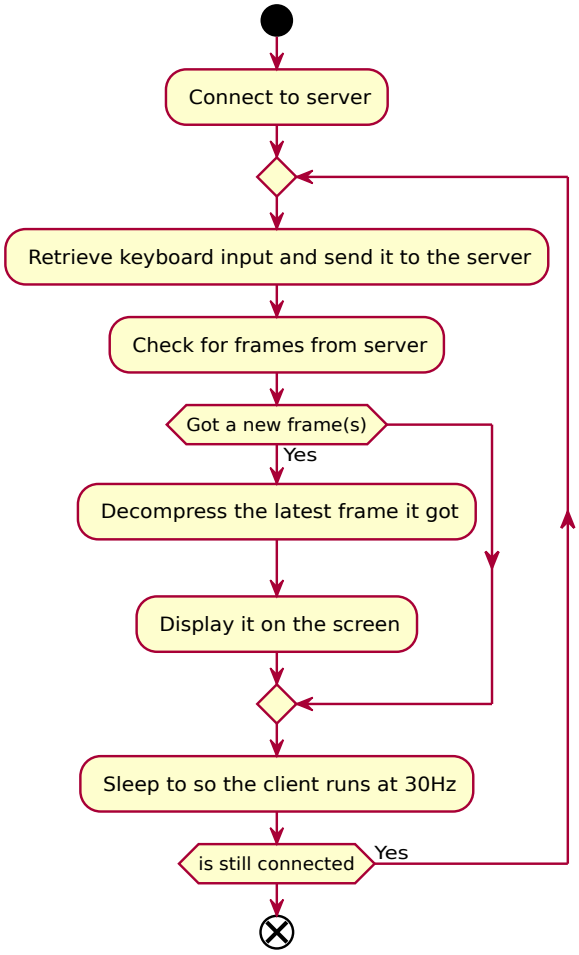

Figure 3.4: The task flow of the client application is not as big as the one for the server, as all the client need to do is receive frames from the server and send its keyboard input to the server.

3.3 Experiment

The tests will be executed as the following:

1. Set X to 0
2. The server is started with X number of cores allocated.
3. Launch one client that connects to the server.
4. Wait around 2 second for the server to set up a game with the newly connected client.
5. Launch another client.
6. Did the server exit due to having a bottleneck? If no, go back to step 4.
7. Note down the number of clients that where connected and then number of cores allocated.

8. Restart on step 2, where X is equal to X + 1, but only if X is not bigger than the number of cores the server has.

These test steps are ran manually and they will be run five times to be able to verify that the data collected is correct.

During the experiment the clients are started with the `nogui` option. As mentioned in Section 3.1.2, this disables the input and the displaying of the received frames. This is used to make sure that the computer that is running all the clients will not lag from all clients running.

The number of client metric is collected from the output of the program. Right before it pausing for being overloaded it prints out the number of clients it could handle. The second metric that is collected is the number of bytes one update on the server sent out over all the sockets combined. This number is also printed out at the same time as the number of clients the server could handle. Lastly, the last metric that is collected is the amount of memory the program required. This was collected at the same time when the server gets bottlenecked and it gets collected with the help of the `ps` commandline program, on the server.

It is these metrics that will be used to answer the research question. The first metric is used to make plots to prove that the server is scaling. The second and last metrics are used to prove that it is the CPU that is bottlenecking, not any other component. It is important that it is the CPU is bottlenecking as this is the component that is used in the research question.

3.3.1 Test Setup

The test setup will only include two computers, where the first one is a server computer that is in data center with a 100 megabits per second up and 100 megabits per second down connection. Its job is only to run the server application and collect data. The second computer is a regular workstation computer with a 250 megabits per second up and 250 megabit per second down connection. Its job is to run as many clients as possible until the server is bottle necked. To make sure that the workstation computer will not be bottleneck during the test, the command-line parameter that disables the displaying of the rendered frame will be turned on.

CPU	2x Intel(R) Xeon(R) E5530 @ 2.40GHz
RAM	24GB
Disk	250GB SSD
Internet	100Mbps up, 100Mbps down

Table 3.1: The setup for the computer that will run the server application.

CPU	AMD Ryzen 7 1800X @ 3.5GHz
RAM	16GB
Disk	250GB SSD
Internet	250Mbps up, 250Mbps down

Table 3.2: The setup for the computer that will run the client application.

Chapter 4

Results

All the follow input is received from the test according to the experiment section. The outputs from the tests are shown in the Figure 4.1. In the same figure the formula $f(x) = 8 + 5x - 0.11x^2$ is made-up to be able to calculate an estimate of the number of clients it will be able to handle. It is only valid when $1 \leq x \leq$ max number of allocated cores. Figure 4.2 it another way of plotting Figure 4.1's data. In this case it shows the gain you would get by adding an extra core. For example, going from two to three cores you gain the performance to handle about five extra clients.

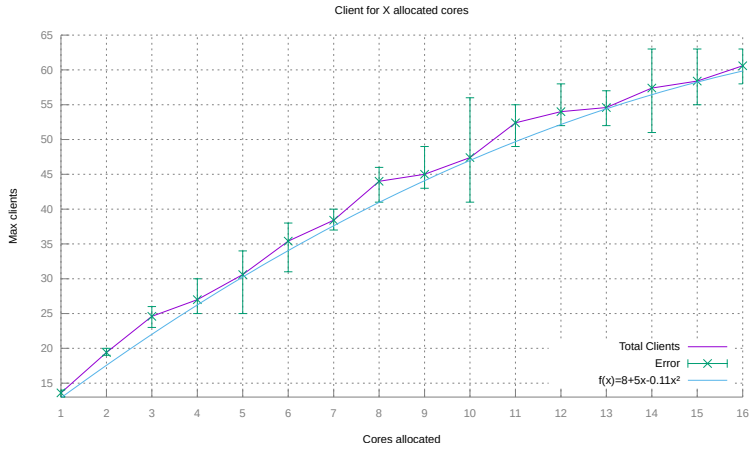

Figure 4.1: The X-axis shows the number of cores allocated. The Y-axis shows the number of clients it could handle. The error margin shows the upper and lower bounds of the result at that allocation. The formula f(x) is a made-up formula that is an approximation on the number of clients the server could handle with X number of cores allocated. f(x) is only valid where $1 \leq x \leq$ max number of allocated cores.

The Figure 4.3 shows an example of how it looks when three clients, the windows with the cards, are playing again one remote server, the black window with green text.

An important metrics that was collected during the final test was the amount of data sent per updated. For 16 cores with 63 clients connected, the server programs sent a total of 1933.90Kbits per update. With the knowledge that the server runs at 30Hz and that 1Mbps is 1000Kbps, we can calculate the total megabits per second sent for the server, $1933.9 * \frac{30}{1000} = 56.017$Mbps. The last additional metric that was collected at the same time as the transfer rate is the total ram usage, and it was 1461MB.

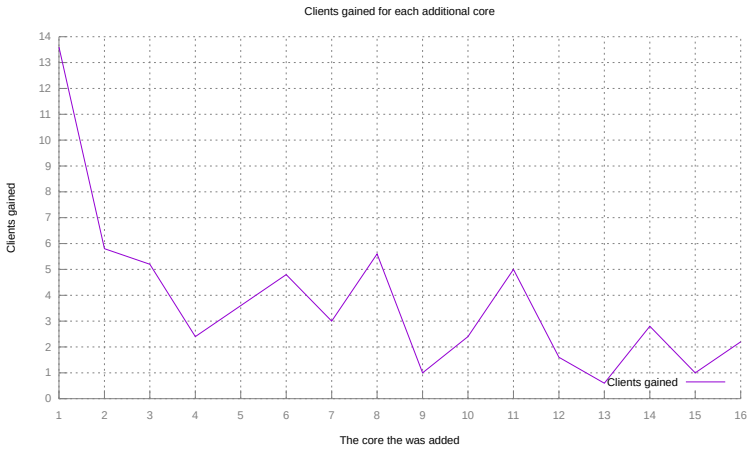

Figure 4.2: The X-value is the core that was added, and Y-value is the number of clients gained from that core. The first core can manage 14 clients. If you add another core you will gain a additional of 6 clients.

Figure 4.3: The black window with screen text is a remote connection to the server running the server application. The bottom-left client has finished its move, top-right is currently playing, and bottom-right waits for its turns.

Chapter 5

Analysis and Discussion

From the Figure 4.1 we can see that it is always going upwards. Which means that you will always be able to run more clients on the server by allocating more cores to the server program. But when you look at Figure 4.2 you can see that you won't gain a constant amount of clients for each extra core. This to due to overhead in the task scheduling. As all the clients are in a single list and the allocated cores want to grab clients from that list. They will have to wait until they can gain exclusive access over that list. This is because two threads cannot modify the same list at the same time; This would corrupt the list. This overhead could be removed by implementing a custom task scheduler. But in this work it was chosen to use the function you can get for free from the standard library, as this improves the readability and speed of development. Furthermore if a none-optimized implementation can show that it can scale that means that a optimized implementation also would do that. Thus it would only be worth write custom code to optimize the implementation, if the standard library code failed to prove scalability.

5.1 Scalability

To be able to answer the research questions, *Is it possible to implement a scalable, software rendered, multiplayer Blackjack game, that scales depending on the number of CPU cores allocated?*, the function $f(x)$ is needed from the Figure 4.1. This function is the closest representation of the scalability of the program. Connecting this function with the research question yields the result that, yes, it is possible to create a scalable Blackjack game as you would still get a growth of clients the server will support by sending setting x to a value that is bigger than 16.

5.2 Bottlenecks

To be able to prove that the limiting factor in the scalability is the CPU and not anything else. It is necessary to prove that the memory and the network, the only other possible bottlenecks, are not bottlenecking.

If you take the total ram usage collected in result and compare it with the server ram about in Section 3.3.1. It can be concluded that memory is not a bottleneck as $1.461\text{GB} < 24\text{GB}$. We now do the same thing but for the internet connection. We can see that the server peaks at 56.017Mbps and in the Test Setup section we can

see that the server have a total upload speed of 100Mbps, thus the network is not bottlenecking.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The conclusion of this thesis is yes, it is possible to create a scalable, software rendered, multiplayer Blackjack game, that scales depending on the number of CPU cores allocated. The results show that even if the implementation is not as optimize as it could be, the experiment is a success. This could be verified by checking if $f(x+1) > f(x)$, or by looking at Figure 4.2.

It was expected that server would scale better than what it did in Figure 4.2. But this is probably because of the missing optimization that could have been done to the compression code and the scheduling code. This scaling factor could probably also do better on more modern hardware.

6.2 Future Work

There is more research to be done within this subject. Firstly, it would be interesting to see how this would work on more modern hardware. Secondly, it would be interesting to see how this would work for more demanding game where a GPU would be needed. This would add one more bottleneck, the communication between the CPU and the GPU, and a challenge to combat this bottleneck. Another interesting topic would be to try to optimize the network communication to not send all the data at once. This could possible remove some bottleneck in the CPU as currently the game will send data on multiple socket each from a different thread, on one network interface. Adding one extra network interface to the current setup could change results.

Another topic that could be future researched would be the compression. The current compression algorithm is not optimized for images, it is made to compress generic data. A compression algorithm that only sent the changed data could be a high gain. Another place where you could hook up the algorithm is before the rendering is executed. The compression code could see that nothing will be changed in the scene and thus could skip the entire rendering code.

There are also a few smaller topics that could be researched for example, how different would the result be if server render at a different resolution and with another pixel format, but also could throughput be better with a different client update scheduler.

References

[1] Andrei Alexandrescu. *The D programming language*. Addison-Wesley Professional, 2010.

[2] Walter Bright. The secrets of compiler optimization. *Micro Cornucopia*, 45:26–33, 1989.

[3] Walter Bright. The d programming language. *DOCTOR DOBBS JOURNAL*, 27(2):36–41, 2002.

[4] Walter Bright. std.parallelism - d programming language, May 2018. [online] Available at: `https://dlang.org/phobos/std_parallelism.html` [Accessed 2018-05-13].

[5] Walter Bright. std.zlib - d programming language, May 2018. [online] Available at: `https://dlang.org/phobos/std_zlib.html` [Accessed 2018-05-13].

[6] Liang Cheng, Anusheel Bhushan, Renato Pajarola, and Magda El Zarki. Real-time 3d graphics streaming using mpeg-4. 2004.

[7] N. Chouhan, M. Dutta, and M. Singh. A code analysis base regression test selection technique for d programming language. In *2014 International Conference on Computational Intelligence and Communication Networks*, pages 1106–1112, Nov 2014.

[8] NVIDIA Corporation. Grid benefits for cloud gaming, Mar 2014. [online] Available at: `https://www.nvidia.com/object/cloud-gaming-benefits.html` [Accessed 2018-05-13].

[9] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev. A fast implementation of deflate. In *2014 Data Compression Conference*, pages 223–232, March 2014.

[10] G. Quadrio, A. Bujari, C. E. Palazzi, D. Ronzani, D. Maggiorini, and L. A. Ripamonti. Network analysis of the steam in-home streaming game system: Poster. pages 475–476. ACM Press, 2016.

[11] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices*. Pearson Education, 2004.

[12] László Szerémi. Image composing and drawing algorithms., Jun 2018. [online] Available at: `https://github.com/ZILtoid1991/cpublit` [Accessed 2018-06-10].

[13] Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. Are all games equally cloud-gaming-friendly? An electromyographic approach. pages 1–6. IEEE, November 2012.

Appendix A

Supplemental Information

The source code for both the Server and the Client:

`https://github.com/Vild/CloudBlackJack`

Appendix B

Plot data

# of Core	Run 1	Run 2	Run 3	Run 4	Run 5
1	14	13	13	14	14
2	19	19	19	20	20
3	25	23	23	26	26
4	30	25	25	27	28
5	25	34	31	33	30
6	37	34	31	38	37
7	40	39	38	37	38
8	46	44	43	41	46
9	49	43	43	46	44
10	44	41	50	56	46
11	50	49	53	55	55
12	54	52	54	58	52
13	57	54	57	52	53
14	61	51	57	63	55
15	62	55	55	63	57
16	58	60	63	60	62

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden