UiO : Department of Informatics
University of Oslo

Real-Time Interactive Cloud Applications

Martin Alexander Wilhelmsen

Master's Thesis Autumn 2014

Abstract

In a time when people are moving away from desktop computers to mobile devices, technology places higher constraints on the computational power and quality of graphics. To solve this, we have designed and implemented a streaming engine which can run a graphical heavy OpenGL application on a server and stream only the output to a client web browser. The streaming engine performs so efficiently that the time it takes for the user to interact with the application until the user sees the effect is fewer than 90 milliseconds. We manage this by utilizing NVENC, Nvidia's hardware H.264 encoder, which is located on the GPU. By running all the expensive tasks on the GPU, we are able to keep the latency low, while saving CPU resources and power. In the future, we see the possibility of streaming rich operating systems, while the client is just a simple video decoder and monitor connected to the Internet.

Contents

List of Figures

List of Tables

Acknowledgements

Jeg vil takke mine fantastiske veiledere, spesielt Håkon Kvale Stensland og Pål Halvorsen, som hjalp meg så mye med å fullføre denne oppgaven samt gjorde det mulig for meg å oppleve GTC 2014. I tillegg vil jeg takke gutta på labben for lange hypotetiske diskusjoner og gode vitser. Min takk går også til Eirik H. Blix og Lasse Karstensen som gjennom sitt vennskap, rike innspill og single malt whiskey har vært uvurderlig for meg. Brynjar Thoresen og Stephen Fenne skal også ha takk for sitt gode selskap. Til sist vil jeg takke mamma, pappa, Ida, Bella og Fredrik for at de har vært der for meg gjennom studietiden og hele livet ellers.

Chapter 1

Introduction

1.1 Background

In recent years, desktop computer sales have plummeted, while sales of smaller electronic devices such as phones, tables and laptops have risen [1–3]. While today's hand-held devices are powerful compared to yesterday's computers, they are still far slower than high-end servers powered by Intel CPUs and Nvidia GPUs.

The difference in computational power between hand-held devices and high-end computers create a gap between what is possible to compute and what the user is able to experience, given a real time requirement. The gap is especially noticeable when looking at games, the big games available on the Apple App Store for iPhone today are often games which were released for PC over a decade ago [4, 5].

Another segment in which the smaller devices are lagging behind is in processing of larger data sets. Our research group has been working on a project called Bagadus where we track soccer player's position and create a panorama video of the soccer field while they are playing [6]. The entire Bagadus pipeline has been optimized to run in real-time, including grabbing frames from 5 Full HD cameras, storing player positional information, and stitching together a panorama image of the entire soccer field [7].

Using the data extracted by the Bagadus pipeline, we have been able to create a virtual camera where the user, in real-time, can control the camera heading and even automatically follow their favorite player [8]. However, running the program requires a lot of resources from the client as it has to decode the high resolution panorama video and make a cylindrical projection of it. With the shift from powerful desktop computers to low powered hand-held devices and laptops, this presents a challenge in availability.

The accessibility of a program is often limited by computational power requirements. A modern laptop or tablet is not able to run the latest games and heavy duty programs which an Intel powered server with a desktop level GPU could run.

A program, like the Bagadus virtual camera, requires about six gigabyte of video in addition to player tracking data. It also requires a powerful GPU to process a panorama image when creating the virtual camera image. As the Bagadus system was designed for doing soccer analysis, it is conceivable that the user wants to keep all the games her team has played in one season. With 16 teams in the Norwegian *Tippeligaen*, the combined data set would then be in the range of 96 gigabytes.

Storing 96 gigabytes on a tablet is possible, e.g., Apple's iPad can be bought with a 128 gigabyte solid state disk (SSD). However, the data set would then occupy almost the entire disk and downloading it would require a lot of waiting for the user. Another key point to this data set is that it may be company property which requires theft protection, such as encryption. Still, the space requirement for the Bagadus virtual camera application is more an annoyance than a challenge.

The main challenge with running a system like the Bagadus virtual camera is the requirements it imposes on the system computation power. Most modern laptops and tablets have the available hardware to decode a Full HD 1920×1080 stream, however decoding the full panorama video at 6700×960 is something very different. In addition to decoding the video, the analysis system performs a lot of transformations on the decoded video to achieve the illusion of the user being in control of the camera.

In this thesis we have designed, implemented and tested a solution where programs, like the Bagadus virtual camera, can run entirely on a server and the final framebuffer is streamed to the client as a H.264 encoded video stream. We have also looked into the responsiveness of the system by running a video game through it.

Technologies enabling running applications remotely have been available for decades [9]. The main problem with streaming framebuffers used to be encoding them. Specialized algorithms were designed to encode changes in typical desktop environment images [10], but they are struggling with rich content such as video playback and computer generated graphics.

With the introduction of high performing H.264 video encoders on consumer GPUs [11], enabling applications to run in a cloud environment should now be possible without much added cost. In addition, almost all modern mobile devices have specialized hardware for decoding video. This combination should make it possible to create low cost, high performing cloud application streaming solutions.

1.2 Problem Statement

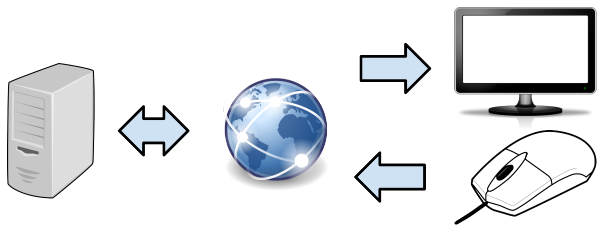


Figure 1.1: Basic overview of application streaming

As an alternative to lowering the power requirements and creating a lower quality service for running on modern mobile devices, we suggest moving entire programs to a GPU enabled server which can stream the framebuffer to a client, i.e., a 3D cloud application (figure 1.1). This will enable fast access to all the data at a lower risk of data theft and with very low system requirements for the client.

The problem we are trying to solve in this thesis is designing and implementing a high performance, remote computing service for running graphical applications, where all the client needs to access it is a common web browser with no extensions or browser plugins installed. This service should be possible to implement using only readily available consumer hardware.

Instead of using a custom algorithm for compressing the images sent from the server to the client, we will try to use H.264, a general video encoding format which several web browsers support playback of. Modern GPUs now include a hardware video encoder supporting the H.264 video format. We will test one such encoder, the Nvidia NVENC, and compare it to the popular software-based H.264 encoder x264. When comparing the two, our main focus will be on factors important for low latency streaming, such as encoding latency, visual image quality, and resource usage in terms of CPU utilization and energy usage.

As client for our streaming service, we will look into using a common web browser. This includes sending user mouse and keyboard input back to the server and displaying the video generated on the server. The HTML 5 standard does contain several technologies which can facilitate our use case including the HTML 5 video element and WebSockets. We will try to use the video element for displaying the H.264 stream generated on the server with very low latency, i.e., minimal buffering, while streaming the user input through a WebSocket. This will enable users fast access to the streamed application without installing any extensions in the browser or a stand-alone client application.

To verify the streaming service, we will try to enable the Bagadus virtual camera application to run inside the streaming service we have designed. Once there, we will do some additional comparison of using NVENC and x264 as video encoder and look into how well the application scales if we were to run several instances on a single server.

The last case study we will perform will be to make Quake III Arena [12] run inside the streaming service. Quake 3 is a high paced first person shooter game which requires low interaction latency and high frame rate to be playable. We will then be able to conclude whether our streaming engine is able to run an actual video game, given the latency and frame rate requirements.

1.3 Limitations

To keep the complexity of the system as low as possible while still being usable, we will design it with a single user in mind. Preliminary tests shows that enabling multiple users in one single instance adds a lot of new problems to be solved, e.g., OpenGL call scheduling, sharing of shaders and other resources, and interaction input routing. Instead, we use the facilities provided by the operating system and device drivers, which may not be as efficient as a custom solution.

As HTML5 video elements are not yet as portable as we would like [13], we will focus on Google Chrome which at the moment is the browser with the highest market share [14]. This does not mean that other browsers will be excluded, however the application will be optimized for Google Chrome. Google has announced that it in the future will remove support for H.264 and focus more on VP8/VP9, but this has not happened yet and might never happen [15].

To be able to deliver this thesis in a timely fashion, we will focus on NVIDIA NVENC as the hardware H.264 encoder. We know that AMD also has an H.264 encoder available in their hardware, but support on Linux was not released until February 2014, and only for their newer GPUs.

1.4 Research Method

The research presented in this thesis is done in accordance to the *Design paradigm* as described by the ACM Task Force in *Computing as a discipline* [16]. We have stated requirments together with specifications, and designed and implemented a functional prototype system on which we have performed a series of tests to very that it meets the requirments.

1.5 Main Contributions

This thesis is a study of using web browsers as thin clients connecting to a remote server where an OpenGL application is running. As such, we contribute several details of this process, including the design and implementation of a working prototype.

Other contributions include:

- Analysis of the NVENC hardware encoder in the context of low latency, live streaming.
- Fast downloading of OpenGL buffers from GPU memory to system memory.
- Techniques for lowering latency when streaming to Google Chrome.
- A way to measure latency when streaming to a web browser.
- Analysis of latency when streaming to a web browser.
- Proving it possible to run fast paced games through the streaming engine.

In March of 2014, we attended the Nvidia GPU Technology Conference (GTC) where we presented a poster showing the design of our pipeline and some preliminary results from running it with the Bagadus virtual camera [17].

We have also a paper describing the latency and scalability of our streaming engine in proceedings for the ICM conference of 2014 [18].

1.6 Outline

Chapter 2 — Remote Computing.
Chapter 2 will introduce the reader to what remote computing is and how it has evolved through time.

Chapter 3 — Commodity Hardware Encoders.
Chapter 3 will introduce basic H.264 and common encoders. Main focus will be on the Nvidia NVENC H.264 hardware encoder.

Chapter 4 — Design and Implementation.
Chapter 4 will propose a design for a graphics streaming engine. The input for this engine is an OpenGL program rendering to an offline RGB framebuffer which is then converted to NV12 before being encoded as H.264 and streamed to a client web browser. Lessons learned in Chapter 3 are used to select the best encoder and encoder settings for the application.

Chapter 5 — Case Study: Bagadus.
Chapter 4 is a case study of the Bagadus Virtual Viewer running inside the streaming engine with emphasis on resource usage.

Chapter 6 — Case Study: Quake III Arena.
In Chapter 6 we put an implementation of the streaming engine proposed in Chapter 4 to the test by running a modified version of Quake III Arena through it. We look into how well the engine performs in terms of latency and how it affects the experience for the user.

Chapter 7 — Summary and Conclusion.
In Chapter 7 we draw some final conclusions from the stream engine and suggest ways of improving it further.

Chapter 2

Remote Computing

2.1 Background

The idea of streaming applications, or rather the output and input of applications, over a network is not new. It dates back to the early time sharing computers of the 1960s when the user would connect and share resources on a mainframe computer through a teletypewriter terminal and later a video display[19]. Today, most people have their own personal computer (PC) in form of a desktop computer, laptop, tablet, and/or smart phone, but the notion of controlling a remote computer is still prevalent. Servers are often controlled remotely by running a Secure Shell (SSH) daemon where a user can connect and get a virtual teletypewriter console. Through this, an administrator can service and configure almost all of the modern servers in a concise fashion.

On a Windows server, an administrator can log on using the Remote Desktop Protocol (RDP)[20]. The protocol was designed for presenting the user with a virtual desktop of the computer the user has connected to. The user can then point and click using a mouse, just as if he was sitting right next to the server with his human input devices (HID) and monitor connected right into the server. Usually, when using RDP the client is presented with a new application window frame in which all the externally running applications reside, but there is nothing actually preventing a program to be displayed on the client just as if it was actually running there, just as its Unix brother the X Window System (X11).

On Unix servers running the X Window System (X11)[21], it is also possible to forward windows running on an external server to a local running instance of the X11 server. This is a side effect of X11 actually being designed with networking in mind. Such a forwarding connection is often initiated using X11 forwarding when connecting to a remote SSH server. For the user, there is no visual difference when interacting with a remotely running application and a local one, other than that there may be added latency.

A popular fourth and portable option is to use a Virtual Network Computing (VNC) service, which functionally does the same as the RDP, where programs run inside a window frame, only it is also available for Unix systems running X11.

The main problem with RDP, X11 forwarding, and VNC is that the latency may is high, especially for applications which include a lot of graphics. While RDP and X11 forwarding is optimized for drawing “normal” window applications, VNC only transfers changes in the remote framebuffer. Users who have tried this will often experience a lot of latency when trying for instance to move windows or run their favorite game. The root cause of this latency is that the images must be encoded. At least two of the most popular VNC implementations use a modified Tight Encoder[10] algorithm for the frame buffers, which uses a combination of zlib and a set of image filters. While Tight Encoder uses a mixture of image filters and zlib, the modified version extends the preprocessing of the framebuffer to include search for high entropy areas which are encoded using JPEG. As this algorithm is without motion estimation, it performs sub optimal on moving elements, such as scrolling a web page.[22].

Instead of using any of the previously mentioned technologies, we use a coding format designed for video, H.264/MPEG-4 Part 10 (H.264). This format has since been succeeded by HEVC, but as of now the video encoders embedded in consumer hardware does not support this new standard. In a discussion on TigerVNC’s bug tracker[22], the maintainer of TurboVNC presents a few valid points on why VNC would not benefit from using H.264 instead of tight encoding, most of which are directed at streaming desktop applications. He admits that for scenarios such as games, H.264 may be better but believes that the performance of the encoder could be a problem.

2.2 Related Work

2.2.1 Remote data visualization through WebSockets

Wessels et. al.[23] present a way of streaming remotely rendered visualization of large data sets. A server uses a GPU to create a visualization of the data set. The visualization is then encoded as JPEG and sent to the user through a HTML WebSocket where the image is presented to the user on a web page in real-time. The paper suggests Base64 encoding of the image before transferring it to the client because binary data may interfere with the WebSocket protocol and JavaScript running in the web browser only supports handling Base64 encoded images.

This paper was released in 2011, but we see no reason why binary data would ever interfere with the WebSocket protocol for two reasons. A WebSocket frame includes the length of the frame, and a WebSocket frame may include a mask which basically renders the transferred data “binary” on the wire, even if the message is encoded as 7-bit ASCII. We do not know whether working with binary arrays was possible back then, e.g., unsigned 8-bit arrays, but there are specifications suggesting they were available already in 2010[24].

The main flaw with the project is that they are using JPEG for encoding the images. Streaming JPEG is basically the same as streaming H.264 with intra encoding only. As images in such a stream usually are very similar, it would be preferable to use some form of inter encoding.

2.2.2 A hybrid thin-client protocol for multimedia streaming and interactive gaming applications[25]

D. De Winter et. al.[25] propose a complex hybrid thin-client protocol where they combine a normal thin-client protocol with a video streaming advanced graphics. They rely on having an abstract 3D driver which can inspect the rendering commands issued and decide whether they should be forwarded to the client’s GPU or be rendered on the server and streamed to the client as an H.264 video, depending on the amount of motion in the framebuffer. There is no need to modify applications to work in the their thin-client architecture.

Because of hardware limitations in 2006, to test their service they used a setup with an additional frame grabbing computer reading the VGA signal from the computer running the application. The frame grabber uses x264 to encode the stream to H.264 at 25 frames per second which is then sent to the thin client through a local network.

They found that they were able to display a frame within 100 ms of reading it on the server, an impressive result for the hardware available to them. They also mention the fact that using B frames when encoding would add extra latency as they are bidirectionally predicted. However, there is no mention of how user input latency is affected.

2.2.3 Remote rendering of industrial HMI applications

Perez et. al.[26] presents a solution very similar to the one we are proposing in this thesis. Their domain is in HMI (Human Machine Interfaces) for industrial machine control systems, but the solution is general.

To grab frames they use the Windows GDI API, which allows an application to read the framebuffer of a computer running Windows, and tests both using Nvidia NVENC, Nvidia's dedicated hardware H.264 encoder, and x264, an open source software H.264 encoder. Frames grabbed by GDI are in the RGB format and have to be converted to YUV (YCbCr). For this they use OpenCV [27], an open source software library for working with images.

They were able to run up to 10 applications at what they call a smooth 25 frames per second, and found that "x264 based encoding is more robust in terms of keeping up the frame rate", while they got a latency of 500ms, due to buffering in the client. Their results are in stark contrast to the results we were able to obtain with similar hardware.

2.2.4 A Networked Service for the Remote Execution of Interactive Multimedia Applications

In this master thesis from July 2008, Mogstad [28] attempts to develop a system that allows end users to play computer games remotely. For some reason, probably due to hardware, he is testing the system with games at a very low frame rate (5-20 fps, page 88). He concludes that the delay (i.e., latency) was the primary factor for users not using his system. Video quality was this second highest reason for not using it, but he argues that higher bandwidth may solve the problem. The primary cause for high latency was because of network and processing.

When looking into types of games playable using the service, he found that arcade racing games and real time strategy games were much more enjoyable than first person shooters. He concludes that first person shooters are the ultimate stress test for thin-client services.

2.2.5 Cloud Gaming

A technology very similar to the one we are presenting in this thesis is the one used by cloud gaming companies. Cloud gaming companies, such as OnLive [29] and Gaikai [30], run games on a remote server and stream only the input and output of the game from and to the client. As such, it does exactly what we are trying to achieve with this thesis.

OnLive is a company which rents out games which run on their servers. Users can select from a large library of games which will be streamed directly to their mobile device, laptop or desktop computer. The only requirements they put on the client is that it has at least 2 Mbps internet connection and the facility to decode a H.264 stream.

Exactly how OnLive's architecture works is not fully confirmed, but they seem to have a separate piece of hardware which grabs the output from the GPU and encodes it [31]. Whether this is true today, after Nvidia released it's NVENC encoder is not known.

The latency from gaming on OnLive's servers was measured by a independent source which found it to be in the range of 135 to 240 ms [32].

Chapter 3

Commodity Hardware Encoders

3.1 Introduction

The last few generations of hardware from the big consumer hardware vendors have included dedicated hardware for encoding H.264 video. This includes smart phones, CPUs and GPUs. The main advantages of using a dedicated encoder is that it may be faster than a pure software implementation, it may use less power, and it frees CPU cycles which can be used for other computations.

Hardware video encoders have been around for a long time, but have mainly been used in commercial TV production. The form of this kind of hardware is often external devices which takes AUX signals as input and produce a digital encoded video over Ethernet as output. There are also dedicated hardware encoders which can be installed into a computer using the PCIe bus. We have tested no such hardware and will not be able to reason about their video quality and performance.

The most popular software encoder seems to be x264, which is available as a package in all the major Linux distributions, and there are pre-built binaries for both Windows and OSX available. x264 is an open source project that is highly optimized to utilize modern CPU SIMD (Single instruction, multiple data) instruction sets, and is known for producing very high quality. However, as a software encoder, it will consume a lot of CPU resources. Because our goal is to create an engine for streaming applications, a key requirement is that the encoder does not inhibit resources needed for running the source application as this would constrict scaling. This makes x264 less attractive for our use case.

Before the advent of dedicated H.264 encoders in Nvidia and AMD's hardware, there were several projects to make use of the GPU cores for encoding [33]. Generally, they have all been recognized as bad in that they do not produce adequate visual quality and compression. The lead maintainer of x264, Jason Garrett-Glaser (aka Dark Shikari), explained in great detail why running the encoding on the GPU cores fail [34]. In short, his explanation is that motion estimation can not be done good when each macro block is computed in isolation as they need some knowledge about their neighbor blocks for the entropy encoder to be efficient. For x264, he has implemented OpenCL look-ahead, i.e., pre encoding, with the following commit comment in the project repository:

> Because of data dependencies, the GPU must use an iterative motion search which performs more total work than the CPU would do, so this is not work efficient or power efficient. But if there are spare GPU cycles to spare, it can often speed up the encode. Output quality when OpenCL lookahead is enabled is often very slightly worse in quality than the CPU quality (because of the same data dependencies) [35].

3.2 H.264 video compression format

H.264 (H.264/MPEG-4 Part 10) is a standard developed by ITU-T Video Coding Experts Group [36]. The details of H.264 are too convoluted to be discussed all in detail here, but some are important for understanding the basics of how video encoding works and how video can be streamed over a network.

One important attribute is the Quantization Parameter (**QP**). Quantization is an algorithm for lossy compression where the Discrete Cosine Transform (**DCT**), the real values of a Discrete Fourier transformation, of a block, e.g., 8x8, of pixels are computed. This produces another block which the QP is applied to set the accuracy of this calculation. A QP value of zero is a lossless transformation, and as the QP increases, the accuracy decreases. Because the quantized blocks are passed through an entropy coder, only the deltas of the block values in a zig-zag pattern is coded. Smaller deltas means lower entropy, which in turn means better compression.

H.264 has two entropy coding algorithms: Context-Adaptive Binary Arithmetic Coding (**CABAC**)and Context-Adaptive Variable-Length Coding (**CAVLC**). The main difference between the two is that CABAC has a somewhat better compression rate than CAVLC. While CAVLC was introduced early in H.264's lifetime, CABAC was introduced more recently with the H.264 High Profile.

Another important thing to know about H.264 is slice prediction. The most important slice types are **I frames**, **P frames**, and **B frames**. I frames are *inter predicted* meaning that they are predicted only using information available in the current frame. They are therefore free standin, meaning that they can be decoded without any of the preceding of following frames. Both P and B frames are what is called *intra predicted*, e.g., temporal predictions. While a P frame is encoded using only preceding frames, a B frame is bi-directional and is encoded using both preceding and following frames.

In the case of live streaming, we do not want to use B frames as using them would add several frames of additional latency. Therefore, we are limited to only using I and P frames.

3.3 Selecting an encoder

There are several possibilities when selecting a hardware H.264 encoder. A quick search on ebay.com shows that the low-end encoders typically starts at \$150[1] while the high-end Cisco (previously Tandberg) equipment can set you back \$6500[2]. Most of this hardware are external devices which you connect to using a vendor specific port and have variable configurability. Instead of focusing on these, we will be looking into using cheap commodity and common available hardware shipped with modern GPUs and CPUs. All the major vendors now incorporate some kind of H.264 encoder hardware: AMD VCE, Intel QuickSync, Nvidia NVENC.

All the applications we have in our current setup are fine tuned for running in a Linux environment. As we already have a Linux environment; to not add a lot of complexity by having some services running on Windows, we make Linux support a requirement for the encoder.

[1]Samsung SPE-100N 1CH H.264 VIDEO ENCODER
[2]Tandberg Ericsson EN8090 HD SDI MPEG4 H.264 AVC

3.3.1 AMD VCE

AMD VCE was available starting with the Radeon HD 7000-series. VCE version 1 is an extension to OpenCL which enables the GPU to generate an H.264 stream from input NV12 formatted frames. The NV12 format will be discussed in section 4.2. VCE version 1 does not support the more advanced features of H.264, such as B frames, but includes all parts we see as vital for live streaming, e.g., I/P frames and some rate controlling mechanism. Sadly, VCE1 is currently only available on Windows.

VCE2 was released with the Radeon Rx200-series in late 2013. As of 4th of February 2014, Linux support was published. However, the release date was too late for us to include it in our streaming engine.

3.3.2 Intel QuickSync

QuickSync was first available with the Sandy Bridge CPU architecture. The hardware has since been developed further and the latest version, available on Intel's Haswell architecture has displayed some promising results [37]. On Linux, QuickSync can be accessed through the libva library. We performed some tests with it on an Ivy Bridge CPU and we were pleased with the results. We did not, however, integrate it in our streaming engine because of lack of time, and the fact that images have to be rendered on the GPU and transferred to system memory to be encoded (see section4.3.4).

3.3.3 NVENC

Starting with the Kepler architecture which was released in 2012, Nvidia included the NVENC encoder. It supports resolutions up to 4096x4096 pixels and advanced parts of H.264 such as B frames and CABAC entropy coding. Linux support includes all the features except asynchronous encoding mode. We were able to run our experiments on two generations of Nvidia hardware, the Nvidia Quadro K2000 which is of the older Kepler-generation, and the Nvidia GTX 750 Ti which is a Maxwell generation chip.

Quadro K2000 is a card in Nvidia's professional line of video cards, i.e., specialized for Computer-Aided Design (CAD) etc., and is built using the Kepler architecture. We will only use this card for testing the performance of NVENC on Kepler and compare it to the newer GTX 750 Ti graphics card.

GTX 750 Ti is one of the first GPUs released by Nvidia on the new Maxwell architecture [38, 39]. It includes a version of NVENC which Nvidia claims is two times faster than the NVENC found on Kepler (6-8x real-time vs. 4x on Kepler).

3.3.4 Encoder Matrix

Encoder	Max Resolution	B frames	Windows	Linux
Intel QuickSync	4096x4096	✓	✓	✓
AMD VCE	1920x1088		✓	✓[3]
Nvidia NVENC	4096x4096	✓	✓	✓

Table 3.1: Device support matrix

[3]VCE2 only as of February 4th, 2014

Based on these findings we chose to use NVENC, but the design of the streaming back-end will be done in a way that does not couple it too closely to NVENC. In addition, the streaming engine will have an x264 back-end enabling us to compare NVENC to a software encoder.

3.4 NVENC

3.4.1 Method

The Nvidia NVENC API provides a lot of configuration options, but due to time constraints we were only able to explore some of them. The ones we picked were the ones we saw as the most important: the ones controlling visual quality, performance, and rate control. In addition, we tested how the encoder interacted with the rest of the system in terms of CPU processing power and electrical power usage.

Our use-case requires an encoder which can produce visually pleasing images while not degrading the user experience. For measuring image quality objectively, we used the Structural SIMilarity index algorithm (SSIM [40]) which generates a value between -1.00 and 1.00, where 1 is only obtainable when comparing two identical images. Good SSIM results, by our experience, have values 0.96 to 1.00. We are aware of the weaknesses of using both SSIM and the older PSNR (Peak signal-to-noise ratio), but the lack of better simple well-known algorithms for determining visual quality have forced us to use them [41, 42].

Quality is of course tightly connected with the stream's bit rate, which create a trade-off between bandwidth and quality. A site called nettfart.no [43] operated by the Norwegian Post and Telecommunications Authority claims that the average bandwidth in Norway is 6-8 Mbit/s while Akamai claims it to be 8.7 Mbit/s ([44] pg. 29). Streaming 1080p video should be possible at that speed, but the quality may suffer, especially on high frame rates.

The main reasons for using a hardware encoder is speed, lower CPU usage and lower power usage. When streaming an application we do not want our streaming engine to degrade the available resources for applications thus reducing overall user experience. To test this we compared NVENC to the resources needed by x264 when encoding at similar quality and bit rate.

We also compared both power usage and CPU usage between the hardware and the software encoder. When measuring power usage of the full system we used an APC Rack PDU 2G specified to have an accuracy of $\pm$ 3% of the reading. The PDU was not designed for measuring the power usage of just one single computer but rather for an entire rack filled with servers and therefore only yielded power usage in kW with two decimals. A typical server will usually use about 170 W on average, which means that the equipment yielded results in a reasonable range for what we are measuring.

3.4.2 Configurability

NVENC is bundled with a set of default configurations for various scenarios, known as presets, which the user can use as basis for further tuning according to the wanted use-case. We found that some of the presets yielded exactly the same configuration and may be placeholders for more specialized configurations in later versions. Table 3.2 shows the different presets, a short description of them, and which are equal. We found their equality by both comparing the configurations yielded by them and running through 690 frames of tractor.yuv (a common 1080p test sequence) and checked for binary differences. The ones we claim to be similar are equal in both tests and there are none which were different in one of the tests and different in the other. Throughout this thesis we will just refer to three presets: Low Latency (LL), High Performance (HP), and High Quality (HQ).

Id	Name	Description	Comment
0	DEFAULT	Low latency	Equal to Id 4, 5, and 6
1	HP	High performance	Unique
2	HQ	High quality	Equal to Id 3
3	BD	Blue ray	Equal to Id 2
4	LL DEFAULT	Low latency	Equal to Id 0, 5, and 6
5	LL HQ	Low latency	Equal to Id 0, 4, and 6
6	LL HP	Low latency	Equal to Id 0, 4, and 5

Table 3.2: Available presets

Because NVENC supports H.264 High Profile, the user has a lot of options to configure. They are divided into two sections, the general encoder settings and the H.264 specific settings. In our streaming engine it is important for us that we can disable automatic Picture Type Decisions (PTD). This means that we can decide how frames should be encoded, e.g., intra (I frame) or P/B inter prediction (P frame / B frame). With PTD enabled the encoder will produce I frames on a given interval or on request, while when disabled the user can choose to only have an intra predicted frame at the beginning of the stream and use inter predicted frames for the rest of the stream. Intra frames are encoded using no temporal predictions and are usually much larger than P inter predicted frames, which only look backwards, and B inter predicted frames, which look in both temporal directions. There are clearly situations where I frames are preferred, i.e., as the first frame or as the first frame after a scene change. It is possible to detect larger changes in the image like that, but is not something we will focus on in this thesis. The effect of not sending I frames on scene changes is that the first few frames will be visually blurry and gradually get clearer after a short amount of time.

Another general encoder setting is the rate control mode. NVENC supports 7 different rate control modes, presented in table 3.3. As we are looking for the lowest latency possible, the two pass settings are not suitable for our streaming engine. The most interesting modes are constant bit rate (CBR) and variable bit rate (VBR). In the next chapter, we will discuss the importance of high bit rates to fill browser buffers faster. This was discovered much later, after we had run the tests presented in this chapter. Still, we have chosen to include the compression available in NVENC to be compared with x264 in this chapter for completeness of the encoder evaluation.

Handle	Description
CONSTQP	Constant QP mode
VBR	Variable Bit rate
CBR	Constant Bit rate
VBR_MINQP	Variable with a minimum QP.
2_PASS_QUALITY	Two pass encoding optimizing for quality.
2_PASS_FRAMESIZE_CAP	Two pass encoding optimizing for bit. rate.
CBR2	Two pass encoding optimizing for bit rate on I frames.

Table 3.3: Rate control modes available in NVENC

3.4.3 Using NVENC

Nvidia provides an API for interfacing with their encoder. To set up the encoder, the user has to provide an encoder configuration, and allocate input and output buffers.

To encode a frame, an input buffer must first be locked down[4] before a frame in the NV12[5] format can be copied to it. When the frame has been copied, the input buffer must be unlocked. Conceptually, when an input buffer has been unlocked the data it stores has been transferred to the device where it can be used. In practice this spawns a direct memory access (DMA) transfer of the data from main memory to device memory.

When a complete input buffer has been created, the user can run the encode function where she provides a pointer to the input buffer and an output buffer. The output buffer is where the resulting H.264 stream for the frame in the input buffer will be stored.

The encode function can return one out of two success flags: *success* or *need more data*. If it returns success, the user can lock the output buffer and copy the fully encoded frame from it. If the encode function returned *need more data*, the encoder was trying to create a B frame and needed more frames to complete it. The user will then have to upload more frames and run encode on them until the encode function returns success. Then the user can download all the frames enqueued for encoding up to that point. This process is displayed in figure 3.1.

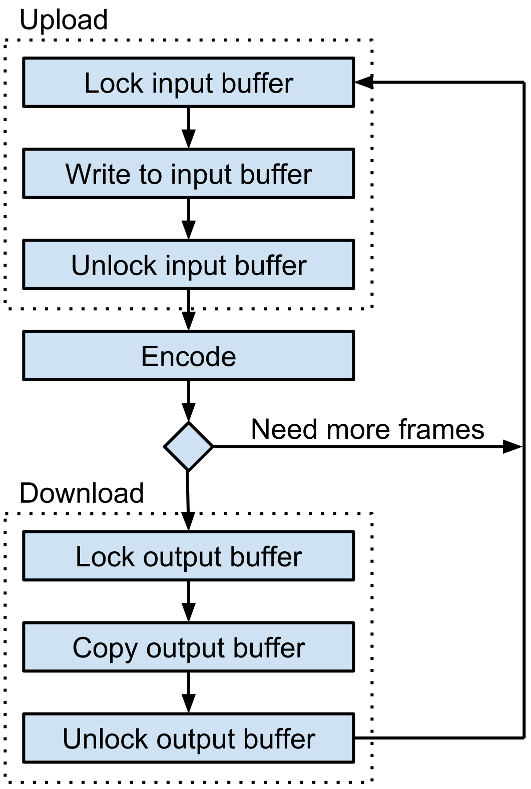


Figure 3.1: Sequential NVENC encoding flow chart

We quickly found that this way of using NVENC is not the best way when aiming for high throughput. Since the encoder is located on a discrete GPU, when uploading a frame the data has to travel across the PCIe bus to the memory located on the video card. This process takes time which can be hidden by overlapping computations.

[4]Locking or mapping a buffer is common terminology across several APIs concerning transferring data to buffers non-local to the CPU, including OpenCL, CUDA, and Microsoft GDI.

[5]Information about NV12 is available in section 4.2. In short, it is a derivative of YUV420, where the chroma channels are stored interleaved.

When profiling the application space code for encoding, we found that uploading and running the encode function takes a very short time while locking the output buffer takes very long. This is counter intuitive because transferring the data as H.264 should be very fast as it is highly compressed.

We believe this is because uploading and encoding will just add commands to a queue. When an output buffer is being locked down, it will have to wait for all the previously issued commands to complete and the buffer has been downloaded to the host memory.

For higher throughput we found that if we first upload one frame before entering the encode loop the throughput increased from 152 fps to 210 fps on Maxwell. This approach did of course add one frame of extra latency, or $1 \times \frac{1s}{152} \approx 6.58ms$ versus $2 \times \frac{1s}{210} \approx 9.52ms$. As most streaming applications will be running at 60 fps, using this approach will not be necessary to maintain the frame rate and the added latency of about $2.93ms$ can be avoided.

3.4.4 Test sequences

To test performance and visual quality of the NVENC H.264 encoder, we selected a few well known test sequences [45] for the resolutions we were interested in. We limited the resolutions tested to the High Definition ("HD") resolutions: 720p ("HD-ready"), 1080p ("Full-HD"), and 2160p ("Ultra-HD"). Encoding a Standard Definition ("SD") stream is very fast using software encoders on modern CPUs, which is why we did not select them for testing.

The test sequence for 720p (figure 3.2a) is a camera filming the city of Stockholm. For 1080p (figure 3.2b) there is a tractor driving through a field. In the 2160p (figure 3.2c) there is a cobra sitting in a tree. What all the test sequences have in common is that they include zoom and camera panning which are features very common in films and is exactly what motion estimation was designed to handle. They are all 30Hz clips stored as raw YUV 4:2:0, an image format explained in section 4.2.2.

Two of the clips, Tractor and Stockholm, were found on a website run by xiph which is the organization behind several free and open video and audio compression formats such as Ogg Vorbis, FLAC, Theora etc. [46]. The cobra clip was found on a website run by Harmonic which has released it under the Creative Commons license.

(a) Stockholm, 1280x720 YUV 4:2:0

(b) Tractor, 1080p, 1920x1080 YUV 4:2:0

(c) Cobra, 2160p, 3840x2160 YUV 4:2:0

Figure 3.2: Sample frames from the test sequences.

3.4.5 Performance

Encoder performance is important in a live streaming system which enables real-time interaction. The longer it takes from a frame is fully rendered until it is displayed to the user the more the user will notice the latency when interacting. For applications requiring accurate input using a mouse, high latency can make the user experience unpleasant or even render the application unusable.

In this section, we will test the performance of NVENC on both the Kepler based Quadro K2000 and the Maxwell based GTX 750 Ti. We will also compare the numbers with x264 on a comparable visual quality. The test will be performed on an Intel i7-3930K running at 3.20GHz with 32 gigabytes of memory. The i7-3930K is a 6 core CPU with hyperthreading enabled making a total of 12 threads that support AVX1 vector instructions.

As part of the NVENC SDK there is a sample encoder which uses NVENC for encoding raw input YUV 420 and output H.264. We had several problems with this implementation and it did not actually demonstrate the performance of NVENC as it was poorly implemented. Fixing bugs and improving performance proved difficult because it was built as a framework using a lot of custom Nvidia libraries, such as thread handling libraries and containers supporting concurrent access. Our solution was to scrap the encoder, and build a new shell for NVENC where we would have better control over the environment and provide easier methods for profiling.

For x264, we tested the ultrafast, superfast, and medium presets. Our experience was that the ultrafast preset produced a visual quality too low to be compared with NVENC, while superfast and medium was somewhat closer to the NVENC presets.

The performance comparison of x264 and NVENC is not of much value other than it can give a view of how the two perform on the hardware given. Especially for x264, the performance is very dependent on the processor running it.

On NVENC, we found that there is not that much difference in performance between the presets LL and HP. Using a pipeline approach adds almost 50% performance on both presets. If we look more closely at the numbers we find that there is a connection between the frame rate and the number of pixels in the image. From this, we looked into creating a formula to approximate the expected performance in pixels per second: $\text{pps} = \text{res} \times \text{fps}$. Table 3.4 shows the numbers for the different resolutions using the LL preset, both sequential and pipelined encoding. The numbers we got are not very close but in the same ball park.

A 1440p frame is about 3.7 mega pixels. Just by looking at the plot in figure 3.3, NVENC should be able to encode it at about 470 MPPS, or 127 fps. We tested this with 510 frames from a 1440p test clip and got 137 fps, which we consider close, e.g., only 7.87% more than expected.

To verify that the numbers we got were not dependent on the video test sequences we used, we tried with a second set of test sequences and got very similar numbers, i.e., a common test clip “Susie logoed @ 2160p” ran at 66.54 fps versus Cobra at 61.60 fps.

No tests were done on resolutions lower than 720p. The reason for this is that we believe that people today expect at least Full HD/HD-ready video while Standard Definition (SD) is a thing of the past. There are still some applications for SD video, such as mobile streaming, but by judging the results we have got, we can tell that encoder performance will not be a problem in those cases.

Resolution	Approach	Megapixels per second
720p	Sequential	271.10
1080p	Sequential	302.97
2160p	Sequential	354.50
720p	Pipelined	399.61
1080p	Pipelined	452.05
2160p	Pipelined	510.88

Table 3.4: LL megapixels per second

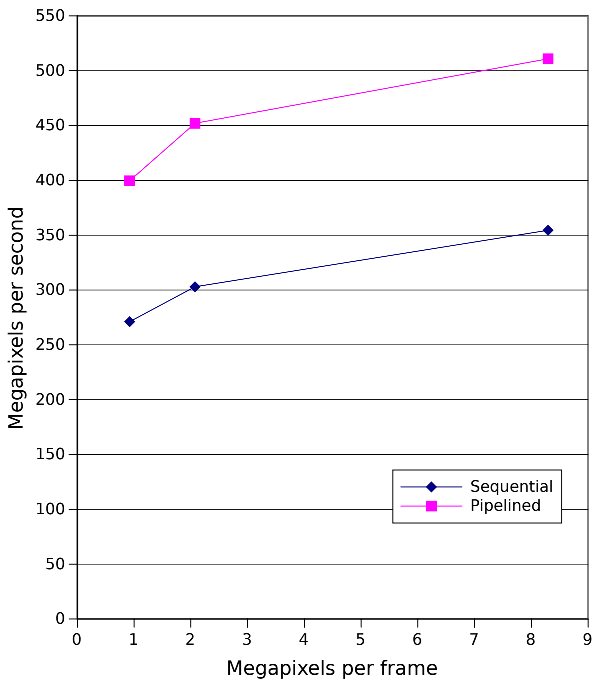
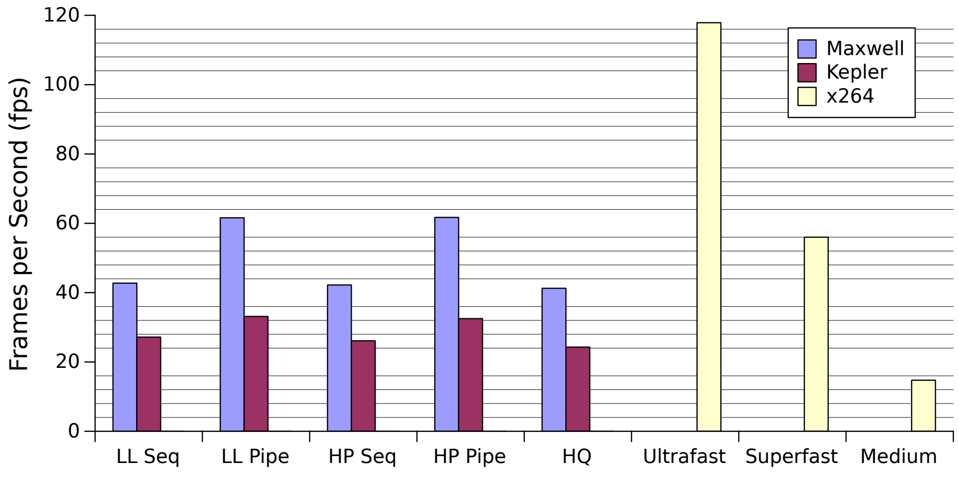


Figure 3.3: Performance in megapixels

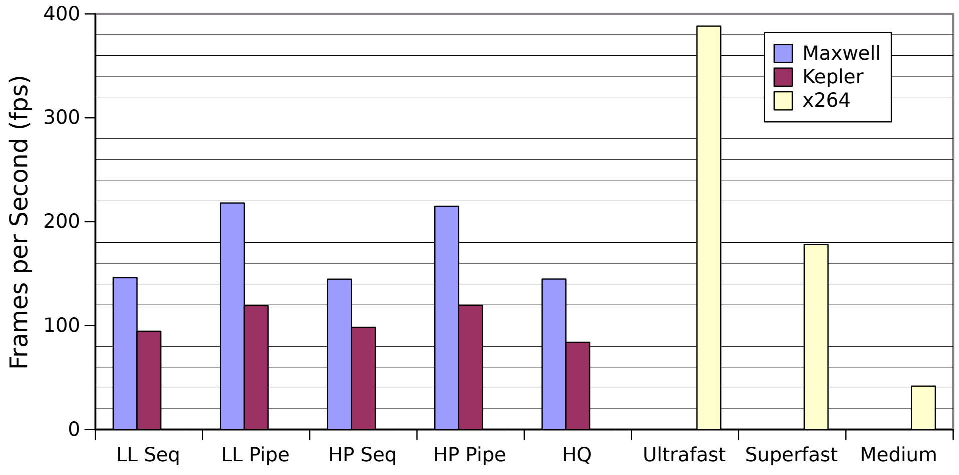
In figure 3.5 we show a detailed profile of where the time is spent while encoding in sequential (3.5a) and pipelined mode (3.5b). We see that the time spent uploading is constant at about 1500 microseconds. A pipelined approach seems not to add time in any of the stages, it only cut the time spent locking the output buffer in half. In a real time streaming scenario, there would be no need to upload a frame to the GPU when using NVENC as the frame is produced by the GPU and reside only in GPU memory.

For the profile to better reflect the true performance of NVENC we created a benchmark where we preconvert our YUV 420 video to NV12 and only use a small number of frames so that we can be sure that they are fully loaded in CPU memory before testing.

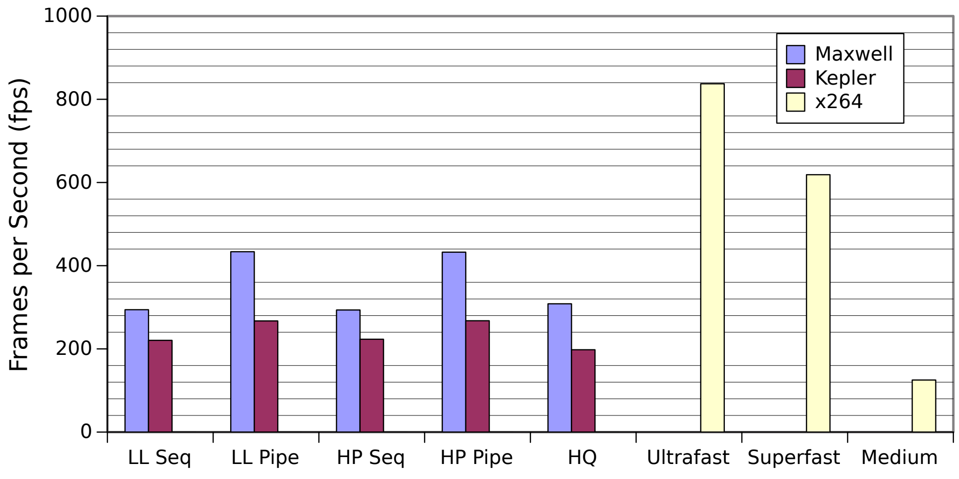
Our conclusion on performance is that for real time scenarios, NVENC's performance is well suited. Smooth running 3D applications often run at 60Hz, which both NVENC and x264 can do, given the hardware we are using. We see that at higher resolutions, such as 2160p (4K), a pipelined encoding approach is required to get reach 60 fps and that x264 running at medium is not an option for real time applications at all on resolutions higher than 720p.



(a) Encoding performance for 2160p

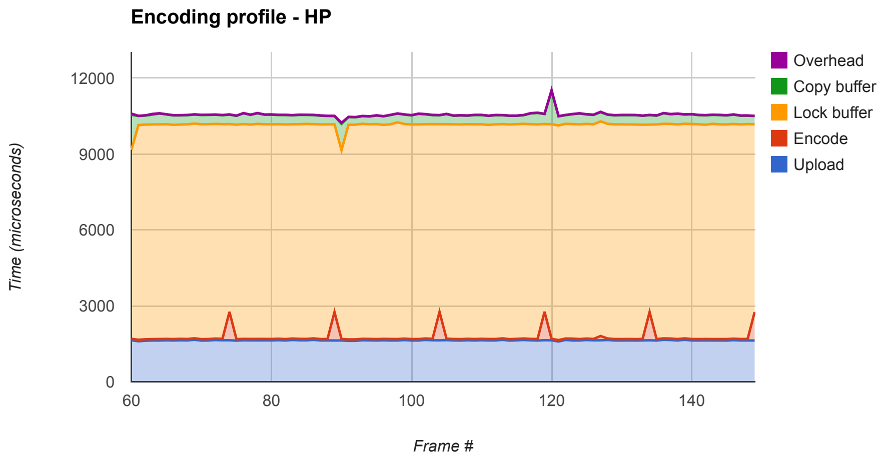


(b) Encoding performance for 1080p

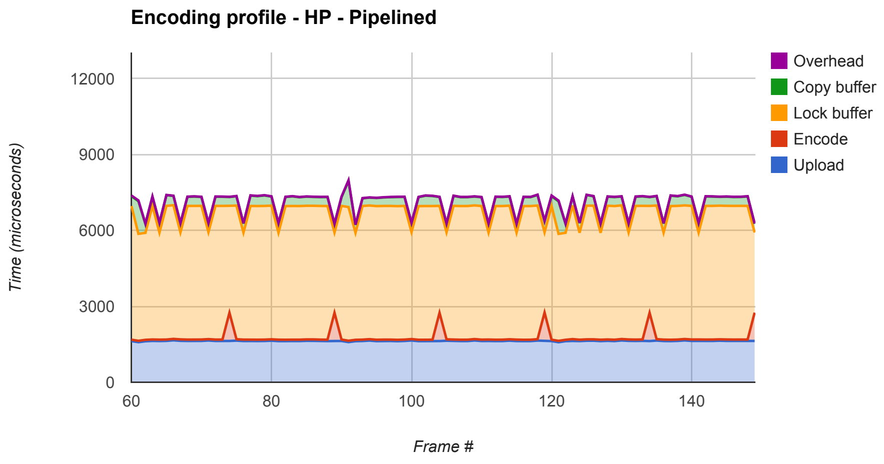


(c) Encoding performance for 720p

Figure 3.4: Encoding performance of NVENC compared to x264



(a) Kepler encoding profile HP sequential



(b) Kepler encoding profile HP pipelined

Figure 3.5: Detailed profile of encoding on Kepler

3.4.6 Visual Quality

Judging visual quality is challenging, especially using objective algorithms such as structural similarity index (SSIM) [40]. When comparing NVENC to x264 in the context of live streaming we had to select the bit rate. We also did some experiments using only Variable Bit rate mode (VB) for both encoders and got some interesting results. On VB, NVENC create much larger files than x264, but also much higher quality, while x264 chose a lower bit rate with lower SSIM measured quality. We decided not to try drawing any conclusions from the test as it is a well-known pitfall which several commercial encoders have fallen for when they are *proving* that their encoder is better than the competition.

Instead, we selected a series of bit rates to compare the two encoders on. To guide us when selecting bit rates, we used the numbers suggested by YouTube on their advanced encoder settings recommendations page [47]. They suggest 8Mbit for 1080p streams. Our experience is that this may be somewhat low, especially since we are using fast presets for low latency. Because of this we also selected a few higher bit rates which, by judging the SSIM results, provide more adequate results. In each case, we also did a visual inspection of the result and, even though the differences may be difficult to spot, we tend to agree with the result from the SSIM algorithm.

We tested the following presets:

LL is NVENC's low latency preset

HP is NVENC's high performance preset

HQ is NVENC's high quality preset. This preset is only included for completeness and will not be considered for the streaming engine.

ultrafast is x264's fastest preset. This preset is highly focused on fast computation and will sacrifice quality.

superfast is x264's second fastest preset. This preset is also highly focused on fast computation, but focuses more on quality than ultrafast.

medium is x264's default preset. It trades less quality for faster computation than ultrafast and superfast.

To calculate SSIM, we use a modified version of qpsnr [48] which calculates both PSNR (Peak Signal To Noise Ratio) and SSIM. The major modification we did to qpsnr was to vectorize the computation using SIMD (Single instruction, multiple data) so that it could complete in our allocated time frame[6].

In table 3.6 we present the results from 690 frames of the 1080p tractor test clip. We chose to present all the numbers gathered on this resolution because we focused especially on 1080p, and the trends seen in this data closely matched our findings in the other resolutions.

We found that LL actually produces better quality than all the other NVENC presets, something we found in all the tests we performed. The reason for this, we can not explain, and it is really strange because LL does not use CABAC as entropy coder. Another observation we made was that the x264 superfast preset produces much lower quality than the other presets.

The reason we added the medium preset is to show how a preset which trades in more time for doing computations produces somewhat better quality. In the previous section, we showed that medium was too slow to be used in a live streaming scenario. LL and superfast are however strikingly similar in the visual quality they produce for all resolutions.

[6]The extra speed was needed because we ran a lot of tests...

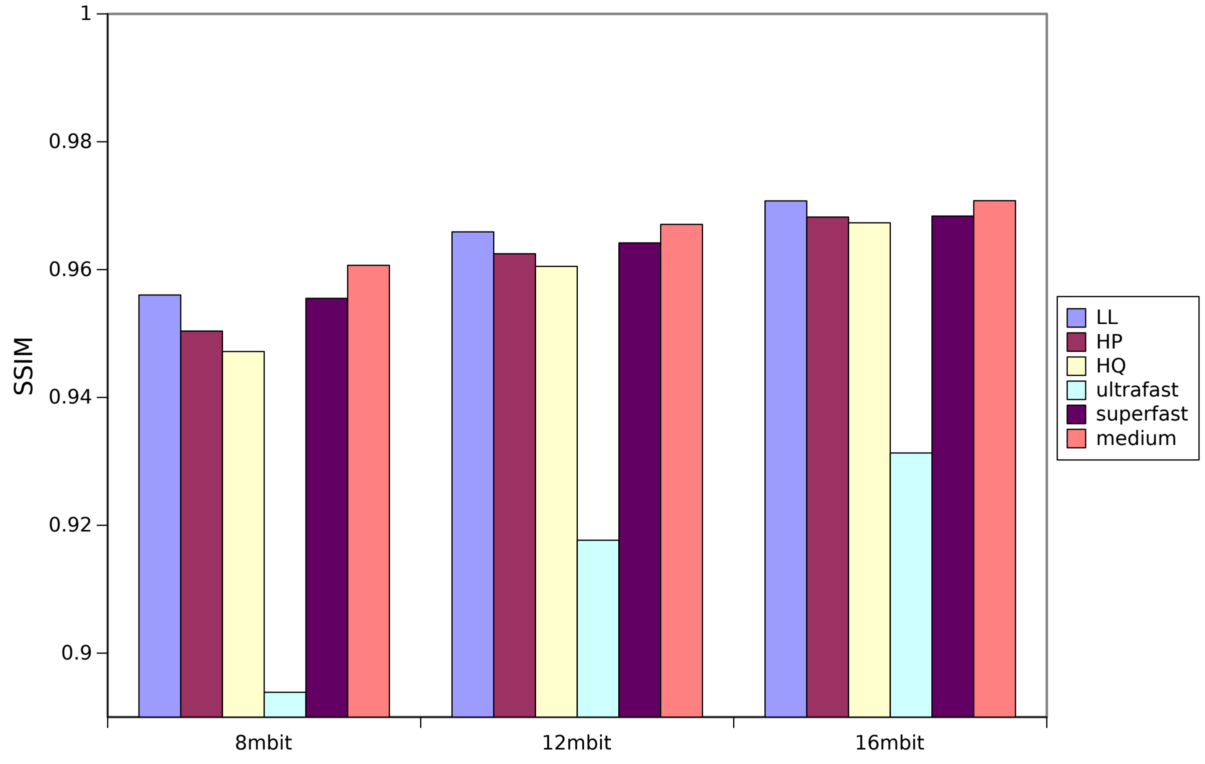
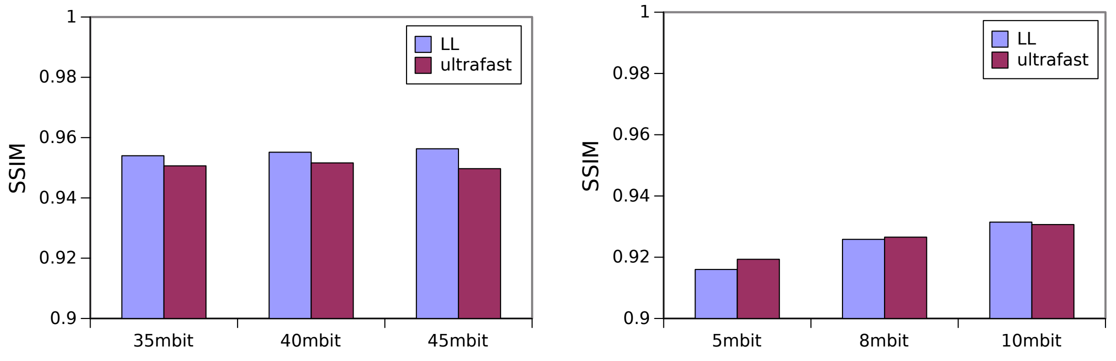


Figure 3.6: SSIM for 1080p, tractor

In figure 3.7 we show the data gathered from 720p (3.7b) and 2160p (3.7a). We see that for 720p, 5Mbit is far from enough. There may be several reasons for this: the encoder is simply not able to produce a satisfactory quality on the bit rates we are testing, or more likely the source video contains a lot of noise which the SSIM algorithm fails to recognize is unimportant when comparing the two.

For 2160p, YouTube suggest bit rates in the interval 35 Mbit to 45 Mbit. We see that the SSIM for those values are generally good and by watching the video we agree. The image is in our opinion crystal clear.

From the results shown here we can conclude that the LL preset produces the best quality out of the three presets provided by Nvidia and that it is comparable to x264's superfast preset setting.



(a) SSIM for 2160p, cobra (b) SSIM for 720p, stockholm

Figure 3.7: Encoding performance of NVENC compared to x264

3.4.7 Resources

Resource usage is very important to our application. We need an encoder which does not use so much resources that the application we are streaming is struggling to run properly. In the case of x264, the encoder will grab almost all the resources available on the system which in our case was 10 out of 12 cores. However, encoding is not only a computational heavy task, it also requires a lot of memory bandwidth, something we have not measured but we can generally say that it is not insignificant.

Table 3.8 shows how much more resources are required by x264 compared to NVENC. This graph shows both running at full speed encoding the 1080p tractor test clip, LL versus superfast. In figure 3.4c we saw that both are running at similar speed. A flaw here is that this actually includes the conversion between YUV420 and NV12 on the CPU, a job which could just as easily been done on the GPU, offloading even more work from the CPU.

Even though NVENC is using almost an entire core, the encoder shell is designed with threads in mind which means that the performance we read is not due to the program saturating the performance of the core it is running on.

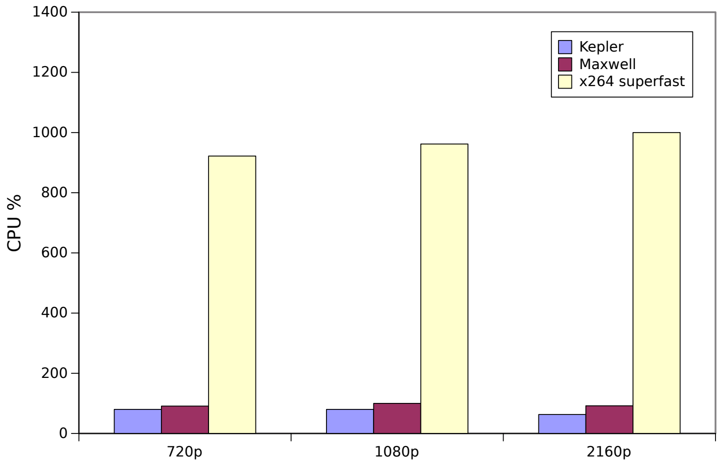


Figure 3.8: CPU resources used when encoding

3.5 Summary

We have shown that NVENC is a good option for live streaming, especially in cases like ours where low encoder latency is key to have an overall low interaction latency. Because x264 is a software encoder it will scale with the performance of the CPU, but a Maxwell was not far behind when comparing LL with superfast. However, running x264 at a high speed required a lot of the available computing power of the CPU which otherwise could be used for running the stream source application.

Additionally, we learned that the visual quality of NVENC's LL preset is comparable to x264's superfast preset. The x264 ultrafast preset did not produce adequate visual quality while x264 medium was too slow to handle 60 fps applications on resolutions higher than 720p.

In the next chapter, we will use what we have learned in this chapter when we try to design and implement a streaming engine capable of streaming OpenGL applications to a web browser.

Chapter 4

Design and Implementation

In this chapter we present the pipeline for streaming OpenGL applications to a web browser using several technologies including OpenGL Framebuffers, OpenGL Compute Shaders, Google Chrome, libx264, Nvidia NVENC and WebSockets. The pipeline is divided into the following three parts:

The inner pipeline runs the application to be streamed. It handles timing issues such as running at a constant frame rate, rendering to an off-screen framebuffer, converting the framebuffer from RGB to NV12, encoding the converted framebuffer, and multiplexing the H.264 stream in a Matroska container (MKV). The stream is then passed to the outer pipeline.

The outer pipeline accepts clients connecting to a custom built HTTP server and serves the HTML front end to the users. It also handles sending a valid MKV stream to the users. This means that it will do book keeping on which frames the user has received and which are missing. Using this information it can decide whether the user needs a new key frame (IDR) or if the stream bandwidth needs to be tuned down.

The HTTP server it runs also accepts WebSocket connections. The messages it receives over WebSocket connection will be parsed into mouse and keyboard inputs which is then passed to the source application.

HTML front-end The HTML front is the web page served to the user when she connects to the custom web server. It contains some JavaScript for opening a WebSocket connection to the HTTP server through which user interaction input such as mouse movement and keyboard keys are sent. It also contains an HTTP5 video element which presents the stream produced in the inner pipeline and later passed through the outer pipeline.

Figure 4.1 shows a simplified overview of what the entire pipeline looks like and how the different parts of it interact with each other.

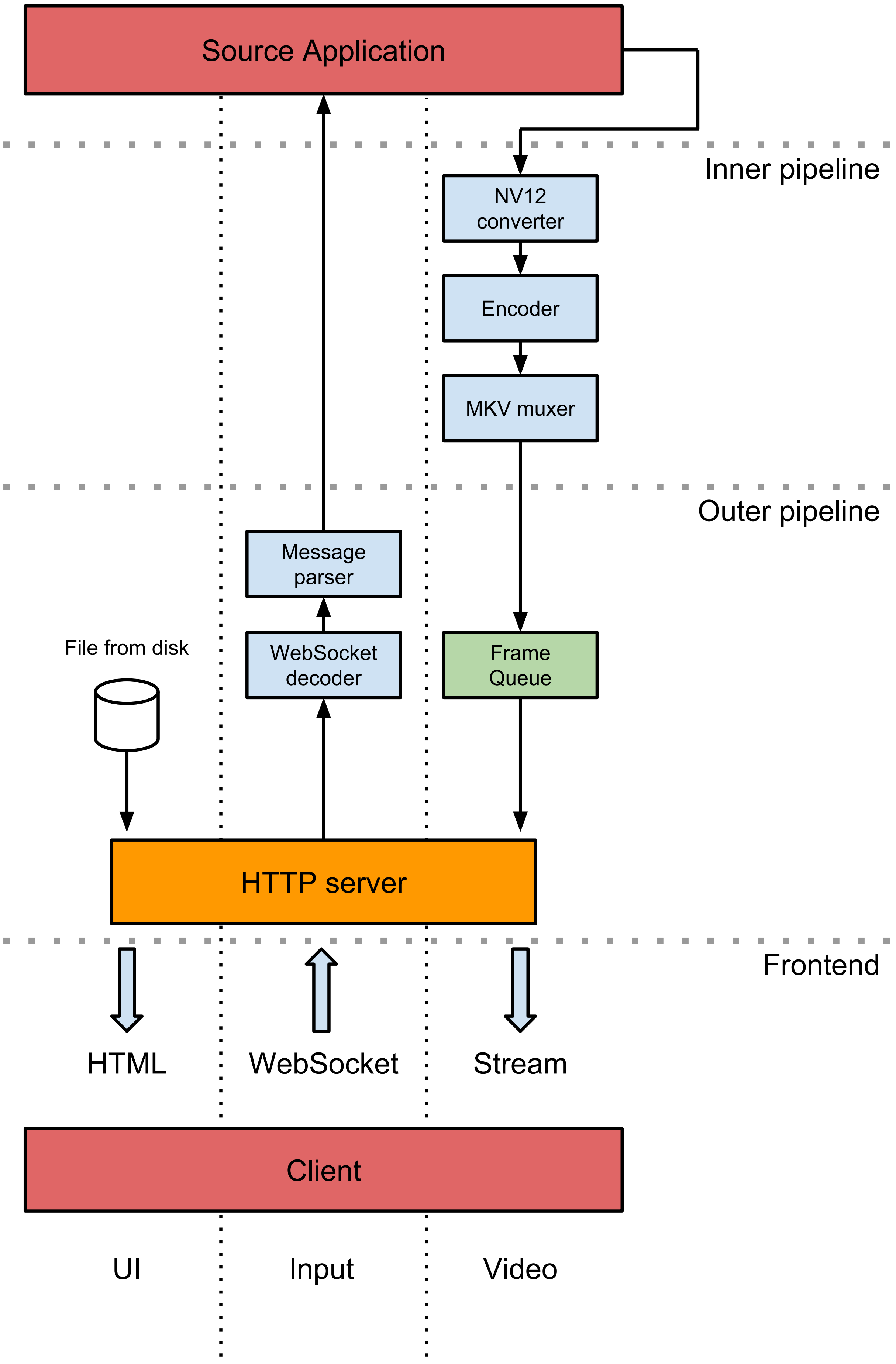


Figure 4.1: Overview of the delivery pipeline

4.1 Delivery Pipeline

Our streaming pipeline was designed to be generic in the sense that it should be able to stream most programs using OpenGL as the graphics library. Generally, OpenGL is not required to make such a pipeline work and one could imagine that using DirectX on Windows could work equally well. In Section 2 we discussed the work of Perez et. al. [26] who suggested a very similar concept. Instead of using OpenGL directly, they used the Windows GDI API [49] to record the content of the desktop. Other possible application also exists, e.g. a cloud video player which plays remote videos encoded using formats not supported by the browser. At the core, all sources of images may be used as input to our streaming pipeline, but applications where user input is important is our main focus.

We designed the engine so that all the source application has to provide is an OpenGL texture and a way to accept the user input. The user input has been divided into three categories: mouse movement, mouse button and keyboard input. The reason for only accepting OpenGL textures is that we are going to work with the data on the GPU. If we instead accept raw data arrays, the pipeline would then have to upload the data to the GPU as most of our operations are performed there. If the source program already was using OpenGL for rendering this would mean that it would have to download the framebuffer, only to be re-uploaded by the pipeline, which would add a lot of overhead.

When streaming to the browser, a stable frame rate is important to get good results. If the source suddenly decided to render at 120Hz when the pipeline was expecting 60Hz the client would notice this very quickly as buffering would occur. If the client started buffering, the stream would no longer be real-time and user interaction would quickly become awkward as the resulting images from input would be lagging far behind.

Keeping a stable frame rate is difficult, especially considering that the images are transferred over a network connection which may introduce variable latency. We overcame much of this by telling the client that the video stream is running at a much higher frame rate than it actually is. A dropping frame rate do not matter as the client would just display the frame as soon as it can, but increasing frame rates above the expected rate will initiate more buffering. To better handle this, the delivery pipeline was designed to control the *render loop* of the source application. In addition to the advantage of less buffering, this could potentially lead to lower latency as the pipeline would then be able to adjust the source rendering to match the refresh rate of the client's monitor!

NVENC expects the input images to be in the NV12 format, while OpenGL produces RGB images. To convert from RGB to NV12, we eventually decided to go for an OpenGL Compute Shader. During the design and testing process we also considered using a CUDA kernel or an OpenCL kernel. Both have OpenGL interoperability which means that they can share data buffers between their API's. Out of the two, only OpenCL is a portable API, while CUDA only runs on Nvidia hardware. Using a CUDA kernel would then mean that the streaming engine would be dependent on using Nvidia hardware. Converting the image using an OpenCL kernel was easy, but there was no simple way to transfer an OpenCL buffer to NVENC. It was only later that we discovered OpenGL Compute Shaders, which were perfect for our application.

When encoding using NVENC, we used the LL preset set up with a tweaked GOP length and IDR frame period. IDR frames usually have a lower data compression ratio than P frames. Less data often means less packets sent resulting in fewer packets lost and lower bandwidth usage which again may result in lower network latency. The reason for selecting NVENC and its LL preset was discussed in detail in Chapter 3. When encoding using x264, we used the superfast preset in zerolatency mode. The zerolatency mode tells the encoder to tune for live streaming with low latency. At this mode, no B frames are produced.

The Encoding pipeline stage was designed to be controlled by two other stages: the RGB to NV12 converter which feeds it, and the Frame Queue which can request encoding setting changes and read the encoded frames back. For simplicity, the logical MKV multiplexing stage was actually merged with the Encoding stage. This means that the frames fetched by the Frame Queue are already multiplexed in MKV. Non-multiplexed frames are never needed in our pipeline.

MKV multiplexing is the process of encapsulating the raw H.264 video stream in a multimedia container called Matroska (MKV). MKV is a standard for defining field names for use with multimedia such as video, audio, subtitles etc. It uses EBML, Extensible Binary Meta Language, a tree structured binary markup language, somewhat similar to XML. An analogy would be that EBML is to MKV what XML is to XHTML. Our use case only required a few of the available MKV fields: video codec, frame rate, video resolution, and the frame timing field.

Before any frames would be streamed to the client, the streaming engine sends a MKV stream header. The streaming engine was designed to pre-generate this header when the application boots and reuse it for all consecutive clients. The header contained all the information about the stream, such as frame rate and video resolution.

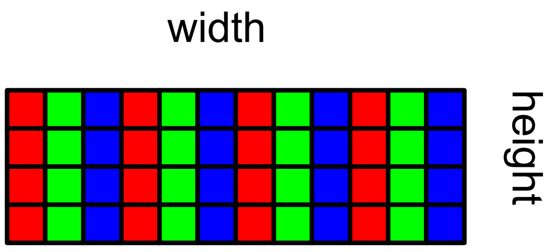
When a frame returns from the encoder, the MKV multiplexer would then encapsulate it in a tiny per-frame header containing time codes and frame byte size. For simplicity, we made it create a new MKV cluster with one block for each frame.

The Frame Queue was the most advanced stage in the entire pipeline. On a high level, it controlled everything about the client's video stream. It was the Frame Queue's responsibility to see to that the MKV header was sent before any frames were sent. It also checked that the client was actually receiving the frames and would tell the encoder that an IDR frame was required for the client to continue streaming if a frame was lost. The ability to drop frames for the client if it lags behind was key to meeting the engine's low latency requirements as it allows for seeking her back on track.

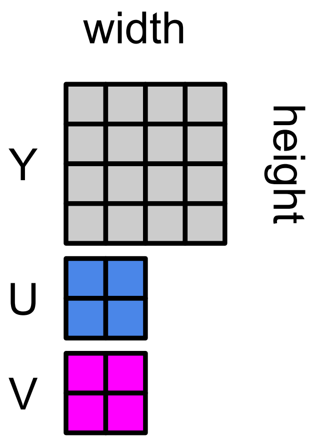
The pipeline was designed to have a built-in HTTP server for serving the client. The server was built using cpp-netlib [50]. When the user points her browser to the server's URL a simple HTML page would be presented. In the HTML there would be an HTML5 video element referring to the location of the video stream located on the same server. In addition, there were some JavaScript code which initiated a WebSocket connection through which all input interaction would happen. Using the jQuery JavaScript library, mouse and keyboard input was hooked and piped through the WebSocket.

On the server side, all HTTP connections asking for `/websocket` would require that the connection was *upgraded* from a normal HTTP protocol connection to a WebSocket protocol connection. We implemented a crude, lightweight, implementation of the WebSocket protocol which supported basic message passing from the client to the server. A very simple ASCII based massage protocol was designed for communicating the user input. Messages received on the WebSocket connection would be decoded using this protocol and translated into calls accepted by the video source application.

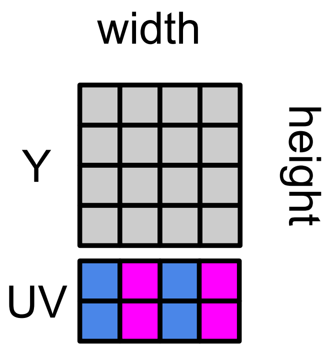
All connections to the HTTP server requesting `/stream/` would be responded by a HTTP 200 (OK) before the socked was handed over to the Frame Queue. The server was designed to spawn a new Frame Queue instance for each and every incoming `/stream/` request.



(a) RGB



(b) YUV420



(c) NV12

Figure 4.2: Color spaces.

4.2 Image formats

4.2.1 RGB

RGB is a color space format where each pixel is stored as a tuple of the three additive prime colors for light: red, green, and blue. When working with colors in OpenGL, all operations are done in the RGB color space, both for textures and framebuffers. The number of colors which can be represented using RGB is depending on the bit depth of each RGB tuple component, e.g. a 24 bit RGB pixel represents 8 bit in each channel which can represent $256^3 = 16777216$, about 16.8 million colors. In addition to the three color components a forth channel is often used called the alpha channel (RGBA). The alpha channel usually represents the translucency used when blending several colors, e.g. mixing two textures. Often, the alpha channel is unused in computation and only serves as padding for making each pixel 32 bit instead of 24 bit as this matches more closely modern hardware.

The OpenGL framebuffer is the destination where rasterization is targeted. It can consist of several planes, but usually only a color buffer stored as RGBA8, RGBA where each channel is 8bit, and a depth buffer used to test whether a fragment from a rasterized vertex will be visible or not. The main advantage of using a depth buffer is that it saves the application some work by not having to sort the triangles it from back to front before rendering each frame.

Internally, OpenGL can store colors as signed integers, unsigned integers or IEEE 32bit floating points. For floats each component is clamped to the interval [0, 1] where the value represents the component's intensity. Often, the framebuffer's color buffer is often a 32bit RGBA where each component is stored as 8 bit unsigned integers having the interval 0 to 255. The depth buffer is often an implementation dependent type only labeled as GL_DEPTH_COMPONENT which can be 8, 16, 24 or 32 bit. It is only single channel and is represented as floats in the range 0 to 1.

Notice the distinction between the words framebuffer and color buffer. In OpenGL a framebuffer is not actually a buffer as it only contains references to the real buffers such as the color buffer and the depth buffer. OpenGL now supports having several color buffers attached to one framebuffer but we will only focus on color buffer zero in our pipeline.

When representing an RGB image in memory they are usually stored as, illustrated in figure 4.2a, a 1D array of pixels where the width and height is meta information used only when doing processing on the image.

4.2.2 YUV

YUV also describes color as a three component tuple containing one luminance channel and two chroma channels, respectively Y, U, and V. This format dates back to advent of the color television where the two chromas were added to the black and white luminosity signal for backwards compatibility. The correct term for digital use of the color space is YCbCr, but YUV is still frequently used. The main advantage of the format today is the fact that it stores the luminosity as a separate channel. Human eyes are much better at detecting changes in brightness than hue which means that when encoding video for humans we can dedicate more bandwidth for the luma. The format where there are 1:1 mapping between Y, U, V is referred to as YUV444. When the chroma channels are down-sampled to four luma values for each U and V tuple luma it is normally referred to as YUV420 if it is stored as a plane for each component, as illustrated in figure 4.2b.

H.264 is made for storing images in the YUV color space, but as input NVENC only supports NV12, which is a mutated version of YUV420 with only two planes. The luma (Y) plane is unaltered while the chroma planes are interleaved where every other value is either U or V, as illustrated in figure 4.2c.

For alignment reasons, encoders usually provide a stride, also known as the pitch, which is the number of bytes between each image line when laid out in memory. Judging by the source of x264 there are no requirements [51] , while for NVENC we believe it to be 512 byte.

4.2.3 Conversion between RGB and YUV

To convert between the two color spaces we are using the formula given by Microsoft [52]. This formula matches well with our use of unsigned integers because we do not have to convert them into floating points in the range 0 to 1.

$$
\begin{aligned}
Y &= ((66r + 129g + 25b + 128)/256) + 16 \\
U &= ((-38r - 74g + 112b + 128)/256) + 128 \\
V &= ((112r - 94g - 18b + 128)/256) + 128
\end{aligned}
$$

Notice that for this formula to work, r, g, and b are casted to signed 32-bit integers. This is a direct mapping from RGB to YUV with one to one sampling otherwise known as YUV444.

To go from YUV444, the one to one mapping from RGB to YUV, to YUV420 we simply down-sample the U and the V channels by taking the average of each 2 by 2 blocks. Strictly speaking, this is the convolution of the $[\frac{1}{4}\ \frac{1}{4};\ \frac{1}{4}\ \frac{1}{4}]$ kernel, then sampling every other pixel in each direction. NVENC at the moment does not support YUV420 as input, only NV12. NV12 is very similar to YUV420 except for how it is organized in memory. While YUV420 consists of 3 planes Y, U and V, NV12 consist of two planes Y, and UV where UV is the two chroma planes interleaved. Back in 2010, when x264 added support for NV12 it was stated in the commit message that this format is better suited for processor caches and should be about 1% faster on the Conroe CPU family (Intel Core 2 Duo) [53].

4.2.4 Implementation

Earlier versions of our pipeline used an OpenCL kernel for converting from RGB to YUV. The main reason for this was to make most of the pipeline portable to non-Nvidia hardware later. OpenCL is an open standard for parallel programming, much like OpenGL is an open standard for graphics programming. Their committees are even under the same umbrella organization, Khronos.

Some advantages of using OpenCL is that it is very portable. Most CPUs and GPUs support it at various levels of performance and it has excellent interoperability with OpenGL. The language is almost identical to ISO/IEC 9899:1999 (C99), the standard even states so, which means that most programmers can read, understand, and most importantly reason about the code. The portability is a key point as the rest of the pipeline is designed to be used portable with minor changes to the code. In an ideal world, only the code actually interacting directly with the encoder is need to be added when changing encoder back-end. As basically all H.264 encoders use some form of YUV as input, this part can be shared among all the back-ends.

However, using OpenCL did not fit well with our use of NVENC as the encoder. The main reason for this is that Nvidia have not created the needed interoperability to pass or copy an OpenCL buffer to CUDA. This meant that we had to first synchronize our OpenGL color buffer with an OpenCL texture, run the conversion kernel which stores the output in another OpenCL buffer. This output buffer would then have to be synchronized with a OpenGL buffer which in turn could be synchronized with a CUDA buffer. The CUDA buffer could then be copied over to another CUDA buffer which was synchronized with an NVENC input buffer to be encoded. Clearly, such a system would be fragile as it involves too many synchronization points which could go wrong. Instead, we decided to skip interop and use system memory as a buffer going from the OpenCL buffer through system memory to the NVENC input buffer. For a while, this worked great and quickly enabled us to work on other more pressing parts on the system. After all, the PCIe bus is fast and a 1080p YUV420 frame is only about 1.3 MB.

Later, it came to our attention that the Khronos Group have added Compute Shaders to a recent version of OpenGL. For our us, this meant that we could drop OpenCL from our pipeline without adding additional dependencies or have a separate code path for conversion when using NVENC. Nevertheless, using OpenGL Compute Shaders was proven to be more challenging than expected. Reading the color buffer was straight forward as it is an OpenGL texture and there are special functions for sampling it using normalized coordinates. The problem was that we can not logically use any of the provided texture formats for our pipeline since we do not want YUV444. What we wanted was the chroma-down-sampled YUV420 format, where, for every four luma values, there was one chroma value of each channel (illustrated in figure 4.2b).

The most obvious method we could think of was to store the output in three different gray scale textures where one was of the same dimension as the original color buffer, for the luma, and the other two were one forth of the original dimension, for the two chroma channels. The advantage with this approach was that the shader would be very simple: spawn one thread for each corresponding chroma. Each thread would then output four luma values to the luma texture and one of each chroma values to the chroma textures. Still, this presented a few new challenges. It would require synchronizing three OpenGL textures with CUDA (actually two, in the case of NV12) to be copied into one buffer in the format NVENC expected it. More importantly, when using other encoders, such as x264, the memory would have to be transferred to main memory using either the OpenGL functions glMapBuffer or glReadPixels. These calls are known for being notoriously slow as they stall the OpenGL pipeline and blocks until the memory is available. Having one of those was bad, having three was worse.

A second version we made used Shader Storage Buffer Objects (SSBO). In this SSBO we allocated $s \times h \times \frac{3}{2}$ bytes of memory to store the converted NV12 image and this worked very good (s is the image stride). The code for converting a 2D coordinate into a 1D coordinate was as simple as $y \times width + x$ which, when hidden inside a helper function did not add much complexity to the shader. The flaw with using this method was not discovered before we found out that the OpenGL Shading Language (GLSL) does not include support for operating on 8-bit integers. The only reason it worked was because we tested it using the Nvidia implementation which provided 8-bit integers as an extension (GL_NV_gpu_shader5). Instead, we would have to write 32-bit integers. This would include using shared memory and a barrier which would add a lot of unnecessary complexity to the shader.

The final approach which we stuck with was to swap out the SSBO from the second approach with a 1D texture backed by a Texture Buffer Object (TBO). This texture is of the format GL_R8UI, meaning one 8-bit color channel stored as an unsigned integer. Basically, this is just a normal buffer which is written to using special functions instead of directly indexing an array in the shading language.

4.2.5 MKV multiplexing

MKV is a standard for containing multimedia information like video, audio, and subtitles etc. The information is stored in a binary tree structure utilizing EBML, Extensible Binary Meta Language, for encoding it. Basic EBML only specifies how branches, keys, and values are to be stored. The basic structure is that every element starts with an id, which also encodes the type of the element, it's size, and the payload. The type may be either some kind of integer, a string, or branch. Ids are not strings but rather binary values.

In MKV's case, it starts of with the basic EBML headers before continuing with the MKV elements. Figure 4.3 shows a dump of the MKV our pipeline produces. The MKV header starts with a doc type, Matroska, and it's version, then continues by specifying the tracks we are going to use.

Our header only contained one single video stream, the video we wanted to deliver to the user. We found that using an extremely high frame rate would counter some buffering in Google Chrome. This meant that would need a high time code scale to encode that each frame should have a very short interval. Because our streaming engine would deliver frames at a much lower frame rate, e.g. 60 fps, telling Google Chrome that the frame rate was much higher, say 300 fps, tricked it into showing the frames earlier than it would otherwise. Still, we were not able to remove all buffering from Google Chrome.

As Codec ID we used `V_MPEG4/ISO/AVC`. AVC, Advanced Video Codec, the old project code name before H.264 was standardized by ITU. Track UID is just a random value we decided, in this case 0xDEADBEEF.

```
+ EBML head
 + EBML version: 1
 + EBML read version: 1
 + EBML maximum ID length: 4
 + EBML maximum size length: 8
 + Doc type: matroska
 + Doc type version: 2
 + Doc type read version: 2
+ Segment, size unknown
 + Seek head (subentries will be skipped)
 + Segment information
  + Timecode scale: 1000000
  + Muxing application: glstreamer
  + Writing application: glstreamer
 + Segment tracks
  + A track
   + Track number: 1
   + Track UID: 3735928559
   + Track type: video
   + Codec ID: V_MPEG4/ISO/AVC
   + Video track
    + Pixel width: 1920
    + Pixel height: 1080
    + Stereo mode: 0 (mono)
    + Frame rate: 900
 + Cluster
  + Cluster timecode: 1.290s
  + SimpleBlock
   + Frame with size 66121
 + Cluster
  + Cluster timecode: 1.290s
  + SimpleBlock
   + Frame with size 66121
```

Figure 4.3: mkvinfo dump of MKV produced by the pipeline

Because our streaming engine was not designed for changing resolution when running, we implemented it so that the MKV header would be created at program boot and be reused for every connecting client. For simplicity, we decided to pack each new frame generated into a new MKV block. It would be possible to add several frames to each cluster, but then we would have to do the multiplexing process per client instead of once per frame.

4.3 Inner Pipeline

The inner pipeline of our video application streamer is where the video is generated, i.e., rendered, encoded, and passed to the outer pipeline for broadcasting. It was designed to run OpenGL programs which, instead of rendering to the default framebuffer, rendered to an off screen framebuffer configured by the pipeline. When a frame is fully rendered, the target framebuffer will be converted from RGB to NV12 (a YUV420 derivative) then passed to either a hardware encoder or a software encoder back-end. We created two back-ends, the NVENC back-end, and the x264 back-end. NVENC is the hardware encoder provided by Nvidia which utilizes application-specific integrated circuits (ASIC) residing on the same chip as the GPU. x264 is the popular free and open source software encoder known for its speed and visual quality.

4.3.1 OpenGL

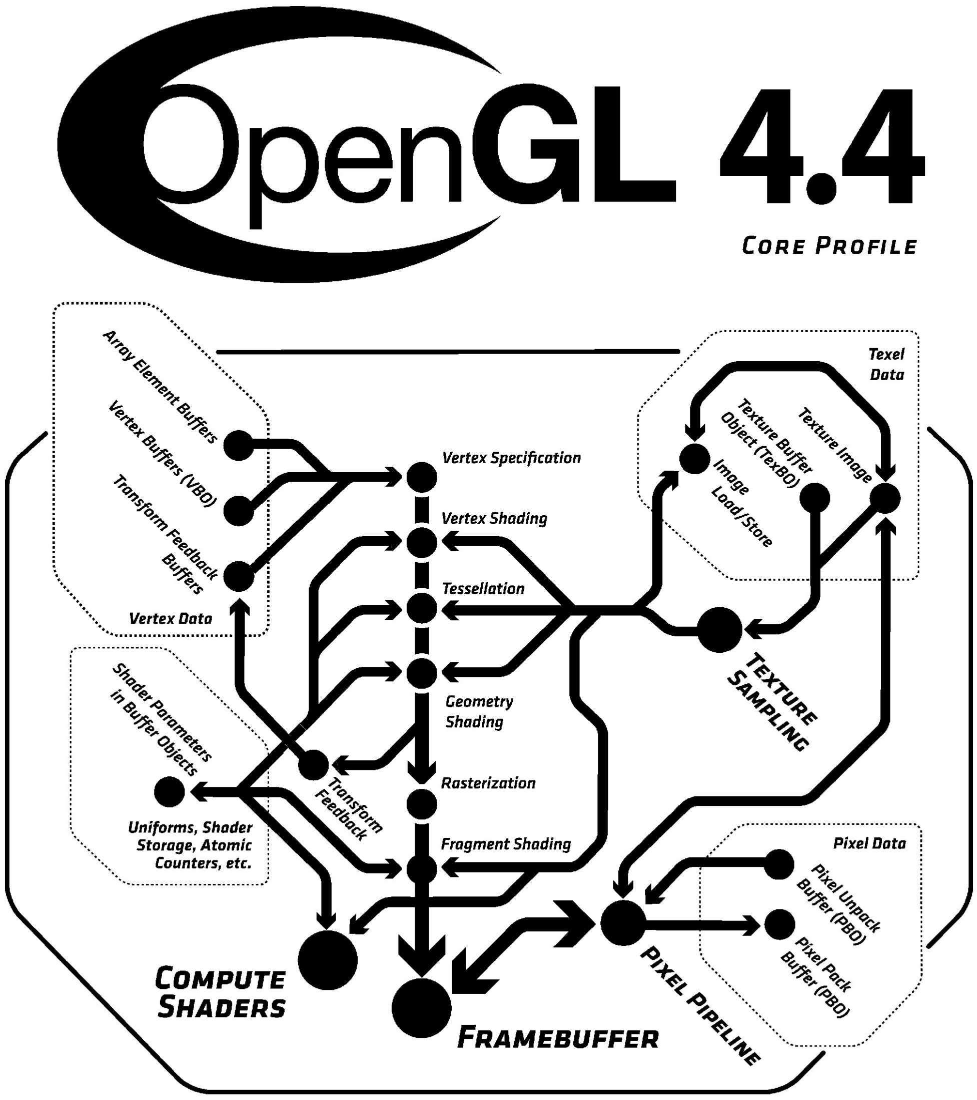


Figure 4.4: OpenGL 4.4 Pipeline. [54]

OpenGL (Open Graphics Library) is an API standard for controlling graphics hardware developed by the Khronos Group. Early versions only supported rasterization of geometry through a fixed function pipeline. Over time, more configuration options were added to this pipeline which eventually was dropped when more and more of the pipeline was made programmable using shaders. This is perfectly illustrated by comparing the original "Programmer's View of OpenGL" section from version 1.0 to how it is stated today, 20 years later, in version 4.4:

> To the programmer, OpenGL is a set of commands that allow the specification of geometric objects in two or three dimensions, together with commands that control how these objects are rendered into the framebuffer. For the most part, OpenGL provides an immediate-mode interface, meaning that specifying an object causes it to be drawn. ... [55]

> To the programmer, OpenGL is a set of commands that allow the specification of shader programs or shaders, data used by shaders, and state controlling aspects of OpenGL outside the scope of shaders. Typically the data represent geometry in two or three dimensions and texture images, while the shaders control the geometric processing, rasterization of geometry and the lighting and shading of fragments generated by rasterization, resulting in rendering geometry into the framebuffer. ... [54]

The most important changes happened in version 1.5 (2003) and 2.0 (2004) when respectively Vertex Buffer Objects and the OpenGL Shading Language (GLSL) were introduced. Vertex Buffer Objects (VBO) broke OpenGL free from immediate-mode only rendering, where to draw, one would have to specify all geometry for every frame. This was very slow when using GPUs because one would have to send the all the geometry over the PCIe bus for every frame. By using a VBO, the programmer only needed to transfer the vertices once and point to that buffer when issuing draw calls.

OpenGL version 2.0 introduced the use of shaders to control vertex transformations and pixel fragment processing. This allowed the programmer to have full per pixel control of the rasterization to the framebuffer which made it easier to create effects like bump mapping, parallax mapping, soft shadow mapping, and using height maps for geometry. Later versions now enable programmable shaders for all the rendering pipeline stages such as tessellation and geometry in addition to compute shaders.

Normal interaction with modern OpenGL includes uploading vertices and textures, compiling shaders for usage in the rasterization pipeline and specifying draw calls. The OpenGL specification only defines the final effect of the issued OpenGL commands and not their specific implementation. This is to keep the API portable without getting vendor lock-in. It is still important to remember that the Khronos Group consists of the hardware vendors, such as AMD, Apple, Intel, and Nvidia, and the big users, such as Pixar, Valve, Google, and Mozilla [56]. This helps the specification keeping in sync with current hardware while not being too specific, making it future proof as the hardware is evolving.

In common implementations of OpenGL, when issuing a draw call, the function returns immediately and does not block until the result is drawn to the framebuffer. This is often done by creating a command queue which is executed out of sync from the program running on the CPU. To utilize this efficiently, it is common to have two, or even three, framebuffers to which OpenGL renders the geometry. This lets the CPU push all the commands for drawing one frame to one framebuffer then swap framebuffers and continue pushing commands while OpenGL is working on the commands from the last frame. If the performance of the program is CPU bound the GPU will finish rendering the frames faster than the CPU can supply the commands. If the performance of the program is GPU bound or asked to synchronize with the display's refresh cycle, the program will at the swap buffer give the GPU some time to catch up with the CPU.

4.3.2 Designing the pipeline

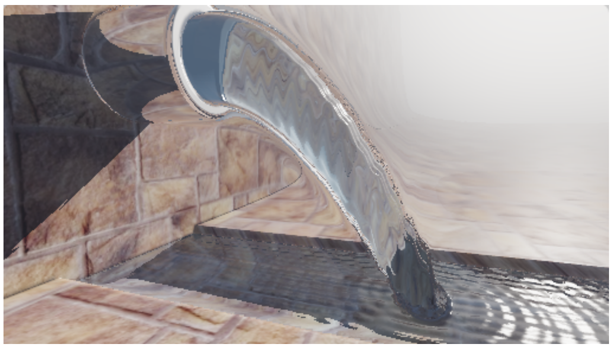


Figure 4.5: Spout by P_Marlin.

Before design our pipeline we first had to select a program or scene to render while testing. To not add too much noise to our test results from CPU computations we went with one of the shaders from shadertoy.com. The shader we selected is called Spout [57] and is created by Paul Malin also known as P_Malin, currently the technical director at Activision. The shader features a running water tap where the view point is controllable using mouse input. It is implemented as a single fragment shader in GLSL and uses a technique called Ray Matching of Distance Fields [58]. We only modified the shader slightly to use a fixed color instead of a texture.

Our primary test GPU, the Nvidia 750Ti, was able to run this shader rendering directly to the default framebuffer connected to a display, with vertical synchronization turned off, at about 140 frames per second. We considered this the target frame rate for our pipeline. It is naïve to believe that we will hit this frame rate, but we do expect to be close. We do add some additional work for the GPU when converting the framebuffer and even more so when download it for the software encoder back-end. For the hardware back-end, the process of OpenGL interop with CUDA may add some synchronization overhead.

At 140 frames per second, every frame takes about 7.14 milliseconds to render. This is well within the peak performance of NVENC, Nvidia's hardware encoder. On the other hand, the software encoder, x264, will struggle to keep up with the GPU in addition to requiring the frame to be downloaded.

When designing a pipeline for encoding the OpenGL framebuffer, one has to keep in mind the parallel nature of the API. Depending on the implementation, it is common that OpenGL commands are dispatched to a queue for execution on the GPU. With a single GPU, only one such task will ever run at a time. This means that if one creates two OpenGL contexts with resource sharing and starts dispatching commands from two threads it may not increase throughput. It may actually lower the throughput, as we will see later in this chapter, because the driver will need to do extra work to keep the two contexts in sync and swapping in and out their states when interleaving their execution.

The pipeline can be divided in 6 logical stages which are illustrated in figure 4.6.

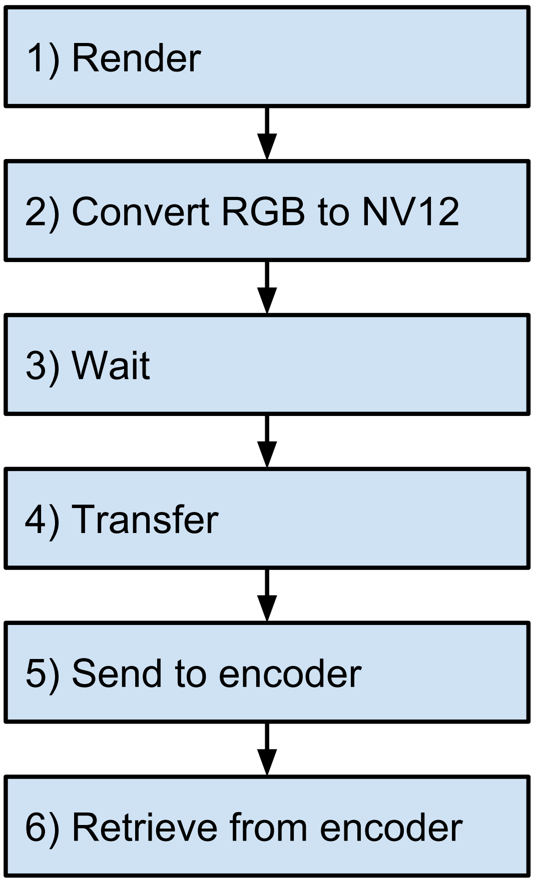


Figure 4.6: 6 stage frame flow.

Stage 1 is rendering the frame using OpenGL. This work is done entirely by the source application. When the source application has finished rendering the frame, the inner pipeline grabs the framebuffer and converts it from RGB to NV12. Then, the inner pipeline has to wait for OpenGL to finish all commands queued up to that stage before it can transfer the frame to an encoder. In the case of a hardware encoder on the GPU the frame is copied from an OpenGL buffer to an encoder specific buffer which is a very fast operation. When using a software encoder, the buffer is transferred from the GPU buffer to a buffer local to the CPU which then can be used by x264. Copying from GPU memory to host memory is a slow operation.

When a frame returns from the encoder, it is multiplexed in MKV, as explained earlier in this chapter, and transferred to the Frame Queue in the outer pipeline.

Pipeline configurations

We have constructed 6 pipeline setups to find the peak performance of the system.

T1B1 is running one thread which renders, converts, and encodes operating on a single framebuffer. This test is fully sequential from our point of view and is the way of running the pipeline with the lowest interaction latency.

T1B2 is running one thread which alternate between two framebuffers. It renders to n, then waits for n to complete and gives it to the encoder. Then it waits for $n - 1$ to finish encoding.

T1B3 is running one thread which alternate between three framebuffers FIFO. It renders to n, then waits for $n-1$ to complete and gives it to the encoder. Then it waits for $n-2$ to finish encoding.

T1B4 is running one thread which alternate between four framebuffers FIFO. The purpose with this is to see how adding framebuffers scales the performance.

T2B2 is running two threads. One thread renders to framebuffer n and issues the conversion on the associated NV12 buffer, then it waits for that buffer to be complete and transfers it to the encoder. The other thread waits for the encoding of $n-1$ to finish.

T2B3 is very similar to T2B2. The first thread issues the rendering and conversion of n, but then waits for buffer $n-1$ to complete and transfers the buffer. The other thread waits for encoding of $n-2$.

T3B3 the final test uses three threads where the first is only rendering to n, the second is only waiting for sync of $n-1$ and transfers it to the encoder. The last is only waiting for encoding of $n-2$. As the second thread is accessing OpenGL, it requires an extra OpenGL context set up with resource sharing.

The first test, T1B1, is optimized to be highly responsive to user input. It is reasonable to say that using the NVENC hardware encoder with this setup should not be a problem as the encoder is very fast. Using x264 may struggle as there is little time for encoding after waiting for the frame to finish rendering. Another flaw with this test is that if submitting several draw calls to OpenGL takes time, this will decrease the frame rate, e.g., computing game logic before actually drawing anything. Adding another buffer to this will probably solve the situation as the GPU gets time to render while the program is doing its thing and submitting more commands. x264 will still have to run encoding in the same thread as the draw calls are issued.

The other tests add more threads which enables concurrent encoding. When using x264, this added concurrency may aid it in finishing faster.

Rendering

Our inner pipeline is designed for easy integration with any OpenGL program witch have a simple render loop. When initiating, the inner pipeline will create the framebuffers which will be used as target for rendering. For each framebuffer we attach a texture as GL_COLOR_ATTACHMENT0 and a render buffer as the depth buffer. The reason for using a texture as color buffer is obvious as we need to access the resulting colors from a render when encoding. We will never need to access the depth buffer and because of this we attach a render buffer. Render buffers are specialized buffers for use with framebuffers [59].

At each render cycle, we bind to a framebuffer before giving control to the client program. There are some flaws with this approach. One is that if the client program is doing any post processing of its frames it will use its own framebuffer for this and then most likely rebind to framebuffer zero. OpenGL does not specify that glBindFramebuffer should function as a stack where it binds to the previous buffer. Instead, binding to framebuffer 0 is binding to the default framebuffer, e.g., the one displayed in the desktop window manager (DWM). We do not try to solve this in this thesis.

Latency

Depending on the configuration we are running in, we are selecting the framebuffer to bind to from a FIFO stack. This way, we can run more OpenGL commands while waiting for the previous frame to render, download, or encode, depending on the number of framebuffers we start with. By adding more framebuffers to the stack, we may utilize or hide more operations while rendering, but it also introduces more latency from a frame is queued for rendering until we are able to fetch it and send it out.

Added latency from adding framebuffers is not linear because it may increase the throughput. For instance rendering a frame to framebuffer A, waiting for framebuffer A to be complete then convert and encode framebuffer A may produce 80 frames per second. Instead, rendering to framebuffer B, waiting for framebuffer A then convert and encode framebuffer A can produce 110 fps. Respectively, a frame will take 12.50 ms and 9.09 ms to render.

In the worst case, the user sends some input just as a frame started rendering. In the case of 80 fps, the user then will have to wait for 12.50 ms + 12.50 ms + convert time + encode time + network transfer + decoding + local display sync. If the frame rate was 110 fps, she will have to wait 9.09 + 9.09 + 9.09 + convert time + encode time + network transfer + decoding + local display sync. If the time it takes to convert, encode, transfer and decode is unchanged between the two, the difference between the two is only about 2.27 ms.

For this reason we introduce the time $L = L_i + L_o$ where L_i is the inner latency and L_o is the outer latency. L_i is the worst case latency for user interaction: $L_i = 2L_f + \text{encode}$, where L_f is the latency for rendering a frame and encode is the time it takes a frame from being rendered until it leaves the inner pipeline.

For a sequential pipeline configuration, the time L_i is simply $\frac{2}{\text{fps}}$, but for the more advanced configurations with several framebuffers the logic is: let the current frame i finish, render the frame with the user input to $i + 1$, wait for $i + 1$ to be rendered and put into encoder which should be $\frac{n}{\text{frame rate}}$, the wait for $i + 1$ to be encoded and retrieved. For simplicity, we are only working with full encode cycles.

$$L_i = \frac{1}{\text{fps}} + \frac{1}{\text{fps}} + \frac{n-1}{\text{fps}}$$

This can be simplified to

$$L_i = \frac{n+1}{\text{fps}}$$

where n is the number of buffers.

4.3.3 Optimizing for hardware encoders (NVENC)

The main difference between the hardware encoder and the software encoder is the fact that the hardware encoder does not have to transfer data across the PCIe bus. Instead, we have to transfer the buffer from an OpenGL and to an NVENC buffer. This means that the output from the converter should be exactly in the format the encoder expects. The input buffers NVENC can process are NV12 buffers with a pitch between the lines aligned at 128byte.[1] To make sure we design an agnostic pipeline, our goal is to create a portable OpenGL Compute Shader to convert from RGB to NV12.

[1]Incidentally, this is exactly the same format AMD's hardware encoder, VCE, expects.

As mentioned earlier, using an SSBO as back-end for the buffer and using a regular buffer array as input to the shader worked well until we found out that indexing individual bytes in this array is not portable, but rather an extension Nvidia provides (GL_NV_gpu_shader5). We ended up using a Texture Buffer Object (TBO) as backing for a texture. The process of creating a TBO is exactly the same as with any other OpenGL buffer object: glGenBuffers, glBindBuffer with TBO as target then glBufferData. The size used for glBufferData is stride $\times$ height $\times$ $\frac{3}{2}$. To bind a texture with this TBO, we used the glTexBuffer with TBO as target and GL_R8UI as internal format. GL_R8UI means that each pixel is only one 8bit byte, stored in the red channel as an unsigned integer. This gives us a 1D textures 1 to 1 mapping of each byte in the TBO. Inside the compute shader we use the imageStore function to store the converted data like so: `imageStore(out, index, uvec4(value, 0, 0, 0))`.[2]

OpenGL interop

Interoperability between OpenGL, CUDA, and NVENC is a complex, but at the same time easy operation. NVENC does not have any concept of OpenGL buffers, only NVENC buffers and NVENC buffers backed by a CUDA 2D buffer. Therefore, we have to use CUDA interoperability with OpenGL to transfer data from OpenGL into this NVENC buffer backed by a CUDA buffer, i.e., a logical three stage operation.

Creation of an NVENC buffer registered to a CUDA buffer is done by first allocating some pitched CUDA memory using the cuMemAllocPitch function. This function takes the image dimensions as input and returns a CUdeviceptr and the pitch of the allocated 2D memory. This device pointer is then used to register a new NVENC input buffer. All TBOs used as target for conversion is then registered into CUDA space using the cuGraphicsGLRegisterBuffer function. This returns a CUgraphicsResource handle which we store in a tuple containing the CUDA buffer used as backing for the NVENC input buffer, the NVENC input buffer, the TBO, and the CUDA resource handle for the TBO.

Each time a frame has been converted and stored into a TBO, the resource handle associated with TBO is mapped using cuGraphicsMapResources. We can then get a CUdeviceptr representing the memory in the TBO using cuGraphicsResourceGetMappedPointer. Then we issue a copy from the mapped memory into the CUDA memory used as backing for the NVENC buffer and unmap the resource handle. In theory, it may be possible to skip this copy stage by directly associating the NVENC input buffer with the pointer received from the CUDA resource handle for the TBO. We did not test this as it depends on all the buffers staying in one place at all time. We do not know if OpenGL is doing optimizations which may change the buffer.

To encode the frame residing inside the CUDA buffer, we first have to map the buffer into the NVENC input buffer. This is simply done using the NVENC API call nvEncMapInputResource which returns an NVENC input buffer which can be used by the nvEncEncodePicture function. As noted in the chapter about NVENC, nvEncEncodePicture is an asynchronous call which starts encoding the frame. On Linux, there is no way to know when the frame is finished other than locking the output buffer. Locking is a blocking call which waits until the frame is encoded and transferred into system memory. When the frame is retrieved, we can unmap the CUDA buffer from NVENC and use it as copy target from another TBO.

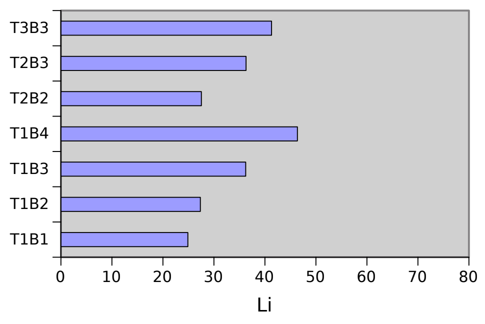
1. Map TBO into NVENC
2. Copy NVENC handle of TBO to CUDA buffer
3. Unmap TBO

[2]It is interesting that even though GLSL supports overloading based on argument types and the type of `out` is an uimageBuffer with r8ui layout it is not able to infer that the value should only a scalar.

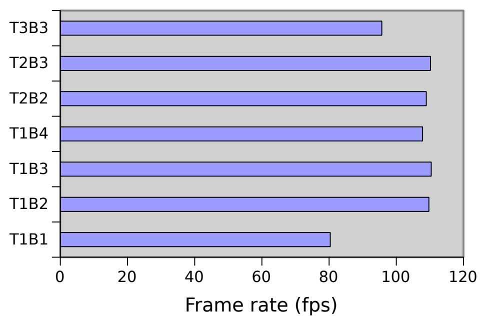
4. Map CUDA buffer into NVENC input buffer
5. Encode frame
6. Retrieve encoded frame
7. Unmap CUDA buffer

Test results: GPU bound performance

Each test is done by rendering 5000 frames of the test program Spout, as described in section 4.3.2, on an Nvidia 750 Ti. The computer is an Intel i7-2600 running at 3.40 GHz with 16GB of system memory.



(a) Latency



(b) Performance

Figure 4.7: NVENC pipeline configuration latency and performance when GPU bound

Looking at figure 4.7 it is clear that in this test, to get peak frame rate one would need three buffers, but if one want the lowest interaction latency, the fully sequential **T1B1** test outperforms the others. The frame rate and latency enabled by having two threads and two framebuffers **T2B2** seems to be a good trade-off.

Test	Render	Convert	Sync	Transfer	Encode	Retrieve	TS	T1	T2	T3
T1B1	0.09	0.02	8.79	0.05	0.14	3.31	12.53	12.53	-	-
T1B2	0.04	0.01	8.87	0.02	0.11	0.07	9.21	9.21	-	-
T1B3	0.10	0.02	5.45	0.05	0.15	3.24	9.14	9.14	-	-
T1B4	0.10	0.02	0.03	0.05	0.16	8.90	9.35	9.35	-	-
T2B2	0.03	0.01	9.00	0.19	0.10	3.25	9.27	9.27	3.26	-
T2B3	0.10	0.02	5.21	0.05	0.16	9.05	9.15	5.64	9.05	-
T3B3	0.20	0.02	4.13	0.09	0.15	9.96	10.54	0.29	4.45	9.98

Table 4.1: Detailed performance of the different pipeline configurations (ms)

In table 4.1, TS is the average for a cycle and T1 .. T3 is the time the work in each individual thread takes on average over the tests. For the multithreaded configurations, we can see that TS does not necessarily match the time from any of the individual threads. This is in small part due to the synchronization which is done using `boost::barrier`, which in turn is implemented using kernel semaphores. The main reason for this, we believe to be that the threads sometimes require much more work than the average. TS here is therefore the average of slowest thread of each cycle.

Below is a description of the various fields in the table.

Render The time it takes to issue the render draw calls.

Convert The time it takes to issue the RGB to NV12 conversion shader.

Sync The time it takes to synchronize with the completion of the last draw calls. For multi-buffered configurations, this is synchronization with the n-th previous draw. In the case of the software encoder, this also includes downloading the frame.

Transfer The time it takes to transfer a frame from a GL buffer to a CUDA buffer. In the case of the software encoder, this is the time it takes to make a copy of the buffer with the right stride for the U and V channels.

Encode The time it takes to start encoding.

Retrieve The time it takes to retrieve an encoded frame from the encoder. This involves waiting for it to finish.

Test results: CPU bound performance

This test is similar to the previous test, but we add 10ms of sleep before each draw call to simulate additional work, e.g., game logic. The flaw here is that a sleep does not create additional real work for the CPU which may be unfair for hardware encoders vs software encoders, as this would inhibit resources which the encoder could have used. It is hard constructing a test case which can mirror real world applications and we are more interested in knowing which pipeline configuration is best for low latency user interaction and which is best for high frame rate.

With 10ms synthetic latency on each frame, the maximum theoretical frame rate is 100 fps.

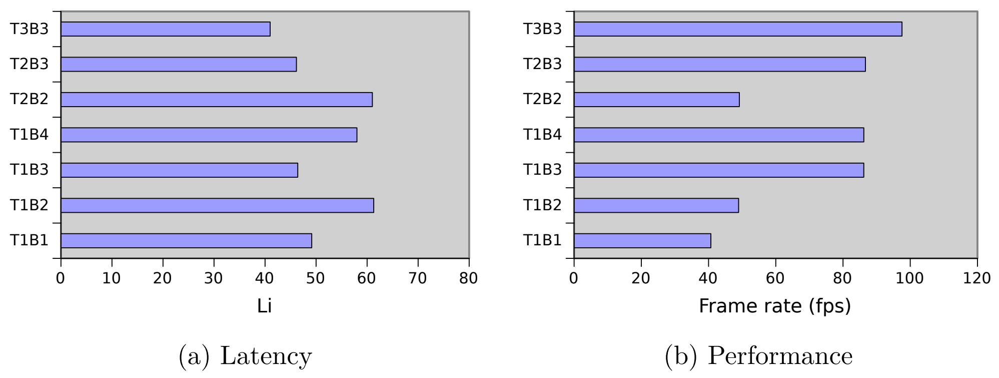


Figure 4.8: NVENC pipeline configuration latency and performance when CPU bound

Test	Render	Convert	Sync	Transfer	Encode	Retrieve	TS	S1	S2	S3
T1B1	10.20	0.02	9.93	0.04	0.14	4.23	24.67	24.66	-	-
T1B2	10.20	0.02	9.93	0.04	0.13	0.09	20.52	24.52	-	-
T1B3	10.20	0.02	0.00	0.05	0.16	1.12	11.69	11.69	-	-
T1B4	10.20	0.02	0.00	0.05	0.16	1.12	11.69	11.69	-	-
T2B2	10.20	0.02	9.93	0.04	0.13	3.82	20.43	20.43	3.83	-
T2B3	10.20	0.02	0.02	0.06	0.16	11.47	11.62	10.56	11.47	-
T3B3	10.20	0.02	8.79	0.05	0.15	2.33	10.34	10.32	9.07	7.99

Table 4.2: Detailed performance of the different pipeline configurations when CPU bound (ms)

4.3.4 Optimizing for software encoders

As the software encoder back-end, we are using libx264. x264 is known for being a fast encoder utilizing all the newest instructions available on the AMD64 architecture. In our case, we are using an Intel i7-2600 running at 3.40GHz with 16GB RAM. This processor was released early 2011 and is therefore not a state-of-the-art CPU anymore, but is still able to encode our stream at an impressive rate. Out of the SIMD instructions available in today's processors, the i7-2600 is missing AVX2 and FMA (Fused Multipy-Add). It does however provide MMX, MMX2, SSE, SSE2, SSSE3, SSE4.1, SSE4.2, and AVX. When starting our build of x264, it claims to use MMX2, SSE2Fast, SSE4.2, and AVX.

SIMD instructions were added to Intel-like processors starting with 3DNow! on the AMD k6 in 1998. What makes SIMD (Single Instruction, Multiple Data) instructions different from other instructions is that they perform processing on vectors instead of scalars. On newer x86, they work with 128-bit and 256-bit vectors[3] which can consist of for example 4, 8, or 16 single precision floating point values. Operations on these are performed in parallel, which can significantly increase throughput, but because of the instruction level parallelism (ILP) available in all x86 implementations today and the fact that many programs are limited by either not being inherently data-level parallel or are bound by memory, most programs do not benefit from SIMD. However, video encoding is one such task that really can take advantage of vector instructions with good results.

The most crippling part of using x264, common to all software encoders, is that it requires the rendered frame to be transferred back to system memory. We had to explore several ways of doing this before settling with a performance we found satisfactory.

Fast read back

In OpenGL, there are several ways to download a buffer and we are of the impression that we tried all of them when trying to get peak performance.

The problem we tried to solve was downloading the data in the framebuffer after a frame has been rendered. The normal way of doing this is to run glReadPixels after a frame is done. This will stall the OpenGL command queue pipeline until the scene has been rendered and the pixels have been transferred to the system memory. Doing so defeats all the advantages from having multiple framebuffers as described earlier in the chapter. Using glReadPixels in the conventional way does not allow us to convert the frame from RGB to NV12 before downloading, which would require the CPU to perform the conversion. Because RGB to NV12 is a task performed much faster by the GPU than the CPU, this is not preferable.

Our first solution was to perform the download in the most naïve way we could. We allocated a TBO and used it as target for the OpenGL Compute Shader, just as we did for the GPU encoder. Then, when the frame was finished, we downloaded the buffer using glMapBuffer. When timing this we found that it took 24 ms to perform the buffer mapping, far more than we expected. A 1080p NV12 frame is only about 3MB, meaning that the download speed was only 136 MB/s. The bus speed for 16 lane PCIe 2.0 is specified to be 8 GB/s.[4]

[3]and 512-bit, currently only available on Intel's Xeon Phi

[4]Note that all times in this section includes the time of rendering the shader!

We then changed how the TBO was allocated from the regular glBufferData to glBufferStorage which was introduces with OpenGL 4.4. glBufferStorage supports additional usage flag hints which lets OpenGL create a persistent coherently mapped buffer where system memory can be used as backing. The idea was that we could use this as target for the conversion shader. This buffer is only mapped once and as long as we wait for the compute shader to complete, the data in the buffer will be coherent. When benchmarking this we found that it did not change download time significantly compared to the normal glMapBuffer approach.

The third and final solution was to instead of writing to the texture backed by a TBO, we create another framebuffer and used it's color buffer as target for the convert kernel. Directly after dispatching the compute shader, we bind to this new framebuffer. Then we bind to a buffer allocated as an GL_PIXEL_PACK_BUFFER and call glReadPixels. Pixel Buffer Objects (PBO) was introduced as an extension by ARB in late 2004 specifically for downloading pixel data from framebuffers. Framebuffers support 1D textures which would have mapped very well to the code written for the hardware encoder, but 1D textures have some restrictions on the size which were too low for our NV12 frames. Instead we had to use a texture with the frame stride as width and $height \times \frac{3}{2}$ as height and only write to the red channel. To accomplish this, we used glReadPixels, which supports swizzling, i.e., reading only one channel.

To pass a copy of the buffer to system memory, we mapped the PBO used with glReadPixels. The data should then be readily available in system memory for us to copy into the x264 input buffer. Using this approach we were able to download a frame in about 10 ms or 300 MB/s. This means that the upper limit frame rate we can run the pipeline using the software encoder on is 100 fps.

Encoder configuration

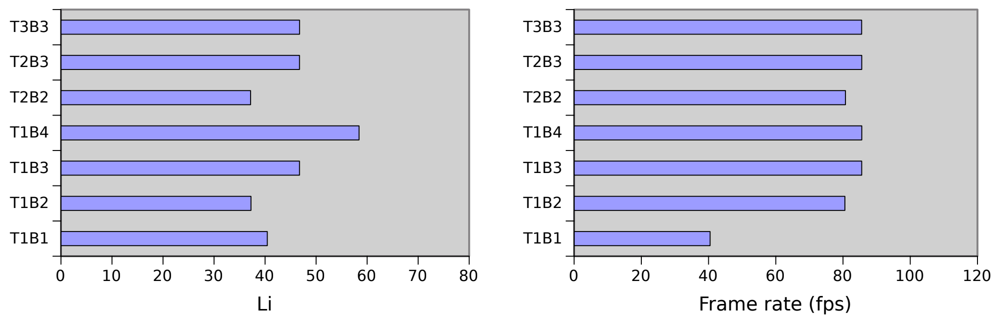
As discussed in a previous chapter, to get x264 to encode at a similar quality as NVENC we have to use the superfast preset. The quality produced when using the ultrafast preset was unsatisfactory because the resulting image suffered from visual blocking effects. To make sure the encoder is aware it is used for real time low latency streaming, we provide zerolatency as tuning and X264_KEYINT_MAX_INFINITE as key frame interval.

Encoding a frame

When passing a frame to NVENC it starts encoding it right away asynchronously while libx264's encode call, x264_encoder_encode, blocks until the encoder has finished. This gives NVENC an obvious advantage as there will be some time between a frame was downloaded and the next frame cycle begins when using a threaded pipeline configuration and constant frame rate. We emulate this behavior by having a separate thread to which downloaded frames are passed. This thread then starts encoding incoming frames as soon as the previous frame has finished or the new frame arrives.

Test results: GPU bound performance

Note that the time it takes to download the buffer is recorded in the sync column. This time corresponds to the 300 MB/s we found when optimizing for read back in section 4.3.4. We see that the frame rates when using the software encoder is somewhat lower, mostly due to having to download the framebuffer.



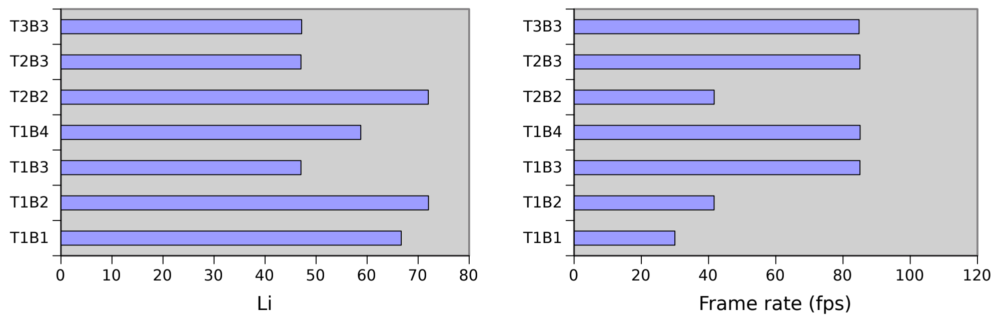
(a) Latency (b) Performance

Figure 4.9: x264 pipeline configuration latency and performance when GPU bound

Test	Render	Convert	Sync	Transfer	Encode	Retrieve	TS	S1	S2	S3
T1B1	0.15	0.18	11.59	0.53	0.01	7.72	20.32	20.32	-	-
T1B2	0.17	0.19	11.57	0.46	0.00	0.00	12.50	12.50	-	-
T1B3	0.17	0.26	10.70	0.46	0.00	0.01	11.77	11.77	-	-
T1B4	0.17	0.34	10.45	0.47	0.00	0.16	11.77	11.77	-	-
T2B2	0.15	0.19	11.57	0.45	0.00	5.53	12.47	12.47	5.54	-
T2B3	0.15	0.26	10.71	0.46	0.00	5.54	11.77	11.69	5.57	-
T3B3	0.20	0.32	10.96	0.65	0.02	0.01	11.77	0.64	11.71	5.63

Table 4.3: Detailed performance of the different pipeline configurations when GPU bound (ms)

Test results: CPU bound performance



(a) Latency (b) Performance

Figure 4.10: x264 pipeline configuration latency and performance when CPU bound

Test	Render	Convert	Sync	Transfer	Encode	Retrieve	TS	S1	S2	S3
T1B1	10.25	0.18	15.60	0.46	0.00	6.81	33.28	33.27	-	-
T1B2	10.26	0.17	13.08	0.46	0.00	0.00	24.09	24.09	-	-
T1B3	10.25	0.22	0.56	0.66	0.01	0.00	11.84	11.84	-	-
T1B4	10.25	0.29	0.47	0.67	0.01	0.00	11.83	11.83	-	-
T2B2	10.45	0.18	13.08	0.46	0.00	6.84	24.08	24.08	6.83	-
T2B3	10.25	0.22	0.59	0.66	0.01	8.71	11.84	11.83	8.72	-
T3B3	10.25	0.23	11.05	0.64	0.00	5.40	11.88	10.60	11.78	5.41

Table 4.4: Detailed performance of the different pipeline configurations when CPU bound (ms)

4.4 Outer Pipeline

The outer pipeline is the interface between the client and the stream. Basically, it is a custom web server which serves connecting clients the web interface and the stream. It takes special care of the stream connection where it always knows which frame the client last received. If the client is lagging behind, the outer pipeline will store a buffer of the upcoming frames. In cases where the client lags too far behind, it will reset the frame queue and ask the inner pipeline to create a new keyframe (IDR) which will put the client back on track. If this happens too often, it can ask the inner pipeline to decrease the frame rate at which it is generating the queue and the encoding bit rate.

4.4.1 HTTP server: CPP-netlib

To save us the hassle of implementing our own HTTP parser and socket library, we decided to use The C++ Network Library Project (cpp-netlib) [50]. Cpp-netlib is a C++ library which provides easy setup of basic HTTP servers. All it does is listen to the TCP port asked by the user where it will parse HTTP commands of incoming connections. When a connection has successfully provided an HTTP header, it will transfer control of it to a callback function defined by the library user. Parameters for this callback is a structure of the HTTP commands and a shared pointer to an object representing the TCP connection.

When starting cpp-netlib, one first creates a thread pool of threads which should handle the incoming connections. This means that the server will not spawn more threads to handle requests and when the requested is passed to the callback it is still running in one of the threads from the thread pool. If handling a request takes a long time it will deplete the thread pool leaving no threads to handle new incoming connections. Instead, the callback function is expected to handle a connection quickly and not do any blocking operations. Writes are done asynchronously by passing the data to be written and another callback to be called when the write has gone through. The actual call to write returns immediately, but the function is expected to return shortly after.

For easy interfacing with cpp-netlib's asynchronously write mechanism, we created a wrapper object we called a generator streamer. This generator streamer is a self-owning object, a term introduced with boost/C++11 shared_ptr. Self-owning objects are defined by inheriting from std/boost::enable_shared_from_this. When a shared pointer pointing to such an object is created, the object functions inside the object is able to request a copy of the reference counted pointer owning it. Using this idiom lets other parts of the code just fire and forget about object lifetime, a weird though in a language which used to be known for it's notoriously hard memory management.

When creating this generator streamer we supply a generator which implements four object functions: the overloaded call operator, `done`, `available`, and `post`. The generator streamer which first probe it's encapsulated object if it is done. When a streamer is done, the generator streamer will just let itself and the encapsulated object get cleaned by not rescheduling itself. If the generator has available data, the generator will call the overloaded call operator and send the data with a copy of it's shared this pointer bound inside a functor as callback. In modern C++, this can be done by simply creating a lambda which fetches a copy of the shared pointer, while in the older C++03 this was usually done using boost's bind-library [60] . If there is no data available yet, we call the post function with a function the callee should call when it wants to reschedule itself. In most cases it is OK to just call this function immediately.

We created an object which handles all incoming HTTP requests, then switches depending on the requested resource: `/`, `/stream/[0-9]*`, or `/WebSocket`. If the user requested `/` we just read a file called index.html from disk and write it to the connection. Because this file is so tiny, there is no performance loss associated with re-reading the file every time. Instead, it allows us to edit the index.html file while the server is still running. Creating some mechanism for caching and checking if the file is modified is not a subject for this thesis.

4.4.2 WebSocket

Connections requesting the `/websocket` resource will start a new generator designed for WebSockets. As the client requesting this resource is aware that it is requesting a WebSocket [61] connection it will include a header token asking the client to upgrade the connection context. This is done by it sending a key challenge to which the client must respond properly. To solve the challenge, the server appends 258EAFA5-E914-47DA-95CA-C5AB0DC85B11 to the given challenge, then SHA1 hashes the result and send it back as hex encoded ASCII with status code 101: Switching Protocols.

Communication over WebSockets follows a standard specified by W3C. All data sent is put in a container frame together with the data length and other attributes about it. For our simple use case, all we care about are text frames less than 127 bytes long. This simplifies our implementation greatly as we can parse the frames with only a few lines of code.

When the browser sends a message it will encapsulate it in a WebSocket frame. WebSocket frames, as illustrated in figure A.1 (appendix), have several fields, but we implement only a few of them and put some requirements on how the browser should behave. We do not support frame fragmentation so the first bit in the frame must be 1, FIN. We only support text frames so op code must be 1. The messages we want to send are always small which means we can limit the support to frames less than 127 bytes long. Google Chrome uses the masking so we added support for decoding masked messages. Masking is a really simple process where the frame includes a 4 byte masking keys which is XOR-ed with the message to produce the unmasked message. The purpose of such a weak encryption is not known to us, but it may counter some very basic attacks.

The messages received through the WebSocket represents commands to the streaming application. To encode these we used a very simple language, as described in figure 4.11.

We did some testing of the latency of WebSockets in Google Chrome, and it was consistent withing less than one ms of the latency measured using the "ping" system application.

```
⟨messages⟩        ::= ⟨messages⟩ ⟨command⟩
⟨command⟩         ::= ⟨mousemove⟩ | ⟨mousebutton⟩ | ⟨keyboardinput⟩
⟨mousemove⟩       ::= 'MM' ⟨coord⟩ ' ' ⟨coord⟩
⟨mousebutton⟩     ::= 'MK' ⟨keycode⟩
⟨keyboardinput⟩   ::= 'KB' ⟨keycode⟩
⟨coord⟩           ::= integer
⟨keycode⟩         ::= integer
```

Figure 4.11: Grammar for messages passed from HTML front end to outer pipeline

4.5 HTML front end

Instead of having a stand alone custom built client, we are using a standard web browser. The solution is only known to work properly in Google Chrome, currently the most frequently used web browser by some measurements. Google Chrome has excellent support for MKV and H.264 which allowed us to tune down the number of frames it will buffer.

4.5.1 HTML and CSS

The HTML page presented to the user is very simple. It contains only one single element inside the `<body>`, the HTML5 `<video>` element. The video element is set to auto play the video source available at `/stream/` once the page is loaded, as shown in figure 4.12. Browsers tend to add some default margin to the `<body>` element [62, 63] which we remove by setting the body's margin to 0 pixels. If the video element is too large for the client's view port usually a scroll bar will appear. For our application, this does not make much sense as we will, using JavaScript, hijack all keyboard and mouse input. We disable the scroll bar by setting the CSS directive `overflow` to hidden.

```
<body>
    <video autoplay id='video'>
        <source src='/stream/' type='video/x-matroska'>
    </video>
</body>
```

Figure 4.12: HTML body presented to the user

```
body {
  margin: 0px;
  overflow: hidden;
}
```

Figure 4.13: CSS presented to the user.

4.5.2 JavaScript and WebSocket

The JavaScript is very simple. It uses jQuery for portable hooking of the mouse and keyboard and is passing this back to the server through a WebSocket.

```
ws = new WebSocket(
    "ws://" + window.location.host + "/websocket"
);
ws.onopen = function(e) {
  \$(document).mousemove(function(e) {
                    ws.send("MM" + e.pageX + "␣" + e.pageY); })
             .mousedown(function(e) {
                    ws.send("MK" + e.button); })
             .keydown(function(e) {
                    ws.send("KB" + e.which); })
}
```

Figure 4.14: JavaScript for sending user input.

4.6 Assessment of the entire pipeline

The time it takes from the draw commands are sent to OpenGL until a frame is fully encoded is possible to calculate as we know all the details concerned with the process. However, as we send the frame out to the client, things are not as easy to work with. There are two main reasons for this: Network and Client side.

4.6.1 Network limitations

Sending anything over a network is always associated with some unpredictability, especially when sending over longer distances. If we were able to send data at the speed of light in vacuum (3×10^8m/s), sending a packet from Oslo, Norway to Bodø, Norway (a flight distance of 8370km) it would take about 3ms. The speed at which light is able to travel through a modern fiber optics cable is about 60% of the speed of light in vacuum, which would make it 5 ms. Of course, the distance between Oslo and Bodø only feels long for us humans but is not really that far compared to the distance between Norway and New Zealand, or even Earth and Sol. Still, the point is that using modern technology, the speed of light is not the retarding factor of our networks.

The main issue with networks are the hardware switching packages. We did an experiment to see how fast we could ping a server located in Nydalen, Oslo, Norway (www.blix.com) from several locations in Norway and Europe. The results are presented in table 4.5. We can tell from this that the impact of using a WiFi instead of an Ethernet cable has significant impact on the latency (12.577 ms / 8.495 ms). Another thing to notice is that typically, for home networks provided by ISPs such as GET, Canal Digital, and NextGenTel the service is much worse than network provided for schools, such as UiO and NTNU. There is no linear relationship between the number of hops, i.e., the number of routers passed through, and the ping time. Some hops may be invisible due to routing protocols.

4.6.2 Client side decoding

A web browser, such as Google Chrome, was not specifically designed for real time streaming. Instead, it was designed for smooth play back across the Internet. To compensate for the network problems described in the previous section, the web browser will begin by buffering a few frames before actually playing the stream. In our use case, this means that it adds some additional latency which we in general can not afford as low latency is our main goal.

ISP	location	ping time	hops
GET	Oslo	8.495 ms	9
GET + WiFi	Oslo	12.577 ms	9
Canal Digital	Oslo	7.000 ms	11
UiO	Oslo	0.725 ms	11
NTNU	Trondheim	8.356 ms	11
NGT + WiFi	Trondheim	23.329 ms	9
Canal Digital + WiFi	Bodø	27.000 ms	13
Signal + WiFi	Bodø	22.000 ms	11
-	Faroe Islands	74.301 ms	20
k-net	Virum, Denmark	13.000 ms	8

Table 4.5: Round-trip time for network packages at several locations

The source code for Google Chrome's H.264 encoder is closed source and not publicly available. We attempted to understand some of the inner workings of the browser by looking at what is open source through the chromium project, but quickly understood that the source is far too complicated to be understood without serious efforts. We decided to instead handle it as a black box where we can only tune H.264 and MKV parameters and see what happens.

4.6.3 Impact on latency from MKV claimed frame rate

The following tests were run using the **T2B2** inner pipeline setting, as double buffered rendering is usually the way games run, and uses the NVENC hardware encoder. The scene rendered is a modified Spout, as described in 4.3.1, fixed at 60Hz.

Before running the tests we measured the ping round-trip time to average at 0.287 ms. Compared to the time period a frame represent on 60Hz, a number so small represents an insignificant amount of time.

In this test, we are interested in how changing the frame rate given in the MKV header impacts the latency we are able to measure. As mentioned earlier, this must always be higher than the frame rate the inner pipeline is generating frame at or else the latency will continuously increase. In figure 4.15 we can see that the average latency from a frame begins rendering until it is shown in the client browser decreases almost logarithmically. We have observed some sluggish displaying of frames when the frame rate in the MKV header is set to high and because of this we want the frame rate to be as low as possible while still getting a decent latency. The latency gain seems to diminish after 300 fps. This seems to be the best value for 60Hz streams.

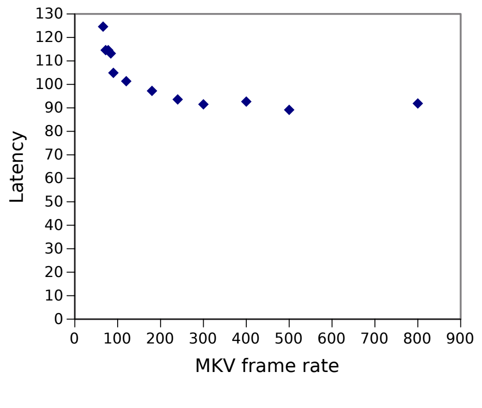


Figure 4.15: Impact on latency on 60Hz when changing frame rate in MKV.

Looking at the box and whiskers plot in figure 4.16, we can see that the latency is not stable. The difference between the high and low medians seems to be less than 20 ms, which is pretty high. A possible reason for this may actually be that the frame rate is not constant as we are using the variable bit rate setting on NVENC.

The numbers here represent the time it takes from a frame starts to render until it is displayed in the browser. Since **T2B2** is using 2 buffers and running at a constant 60Hz, we can calculate the latency of the inner pipeline using the formula given in 4.3.2 minus the first wait $\frac{2}{60}s \approx 33.33ms$. When the MKV claims that the frame rate is 300, the measured average latency is 75.58 ms. By subtracting calculated time in the inner pipeline by the measured average time we get the time from when a frame leaves the inner pipeline until it is displayed in the client browser: $75.58ms - 33.33ms = 42.25ms$. A frame at 60Hz lasts approximately $16.67ms$, this means that the browser is buffering on average 2.53 frames. As counter intuitive as this may seem, the browser may no actually count frames when it buffers, only the number of bytes it expects a frame to be.

Increasing the frame rate in the MKV header does reduce browser buffering and playback latency, but may have an impact on the playback animation. We are able to notice this, but the difference is so small that it does not have a significant impact on the experience.

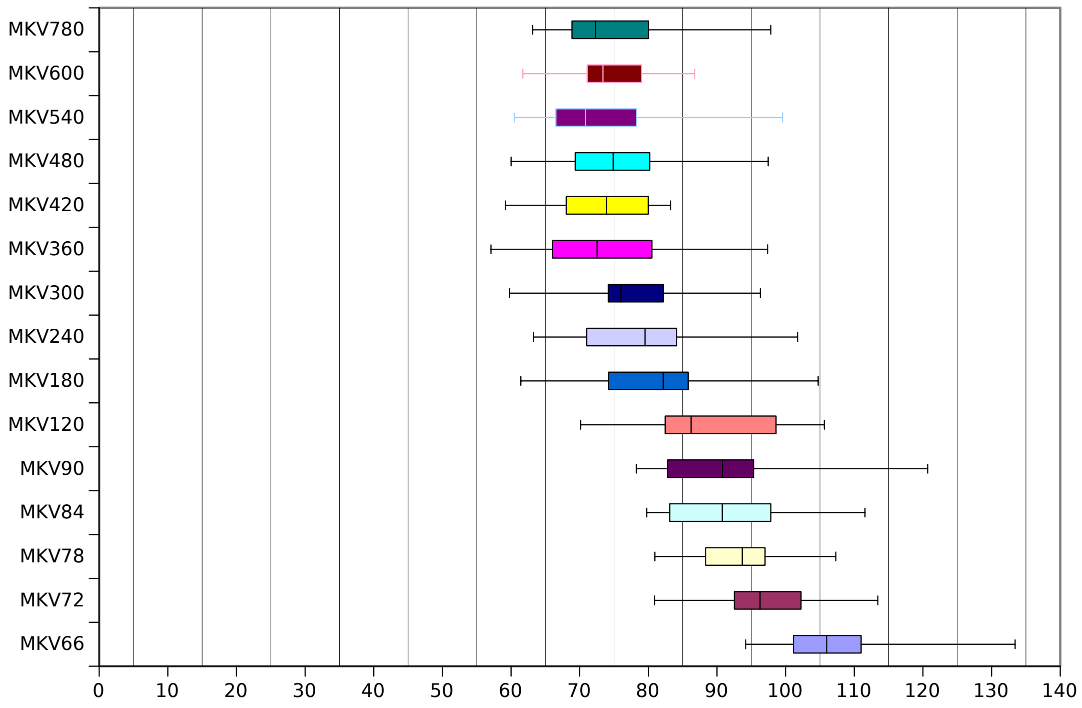


Figure 4.16: Impact on latency on 60Hz when changing frame rate in MKV.

300 frames per second is 500% more than 60 frames per second. With the impact such a high frame rate may have on the playback of the stream, we tested how the effect was when rendering the exact same scene on 30Hz. Our results were very similar to the 60Hz rendering. A claimed frame rate 500% more than the actual frame rate seems to produce the best results in Google Chrome. The numbers from this test is presented in table 4.6. In this case, the average latency for claimed 150 fps was 129.147. The inner latency minus one frame in this case is $\frac{2}{30}s \approx 66.67ms$. The time between the inner pipeline and the client display is then $62.48ms$. But whereas the browser buffered 2.53 frames when reading a 60Hz stream, the browser only buffered 1.89 frames when reading a 30Hz stream.

MKV Frame rate	L_m
36	175.249 ms
60	159.671 ms
120	144.217 ms
150	129.147 ms
300	132.207 ms

Table 4.6: Latency for 30Hz. T2B2, NVENC

The peak performance for **T2B2** was measured earlier to be 108.93. Running the same test on it too provided additional insight on how changing the MKV frame rate influence the latency. We discovered that a when claiming the frame rate to be 1080 fps the browser would simply not try to play the stream. At such a high frame rate, it seems that the curve flattens seems to be earlier than in the 60Hz and 30Hz tests, already at 300% or 400%. The latency from a frame leaves the inner pipeline until it reaches the display is $50.57ms - \frac{2000}{108.93}ms = 32.21ms$. The number of frames buffered is 3.5.

MKV Frame rate	L_m
116	60.9623 ms
216	53.0671 ms
324	51.5814 ms
432	50.5663 ms
540	51.4286 ms
648	50.2203 ms
864	49.7936 ms
972	50.0024 ms
1080	-

Table 4.7: Latency for 108.93Hz. T2B2, NVENC

4.7 Latency

The inner latency, L_i, discussed earlier, is the worst case time from user input arrives at the server until the frame where this input is taken into account is fully rendered and encoded. This depends on the number of buffers n and the frame rate f, given as frames per second.

$$L_i = \frac{n+1}{f}s$$

The latency measured, L_m is the time it takes from a frame starts rendering until it is visible on the client display. To measure this, we used a technique where ever frame is tagged with a green dot in the upper left corner, except one frame every two seconds. Using JavaScript on the client, we are able to detect the frames which do have a black dot instead of the green dot and signal this back to the server. The server can then calculate the time it took from it created the marked frame until the client responded.

To copy the video buffer from the `<video>` element to a JavaScript buffer for detecting the dot, we had to make a detour through a `<canvas>` element. `<canvas`-elements support copying data from a `<video>` element, which in turn can be extacted to a JavaScript byte-array.

Because we can not accurately detect the dot using JavaScript, sometimes the dot may be detected later than it has been displayed and other times it will not be detected at all. However, the latency measured will never be lower than the actual latency. If the latency is higher than two seconds, the calculated L_m will be erroneous, but a human will quickly detect such high latencies and abort the test.

The outer latency, L_o, is the time measured, L_m, minus the inner latency and the extra frame taken into account in L_m.

$$L_o = L_m - L_i - \frac{1}{f}$$
$$L_o = L_m - \frac{n}{f}$$

Finally, the worst case interaction latency is the sum of the inner latency and the outer latency, or just the measured latency plus $\frac{1}{f}$

$$L = L_i + L_o$$
$$L = \frac{n+1}{f} + L_m - \frac{n}{f}$$
$$L = \frac{1}{f} + L_m$$

The number of frames buffered, B, is L_o divided by the time each frame represent $\frac{1}{f}$. There are several places a frame may be buffered including the server's TCP stack, a network switch, the client's TCP stack, the client's browser, or even the H.264 encoder.

$$B = \frac{L_o}{\frac{1}{f}}$$
$$B = L_o f$$
$$B = (L_m - \frac{n}{f})f$$
$$B = L_m f - n$$

f	L	B
24 fps	199.74 ms	1.79
25 fps	200.00 ms	2.00
30 fps	161.05 ms	1.83
50 fps	108.31 ms	2.42
60 fps	96.63 ms	2.80
90 fps	66.84 ms	3.02
100 fps	63.49 ms	3.35
109 fps	60.60 ms	3.61

Table 4.8: GPU bound, T2B2, NVENC, $n = 2$

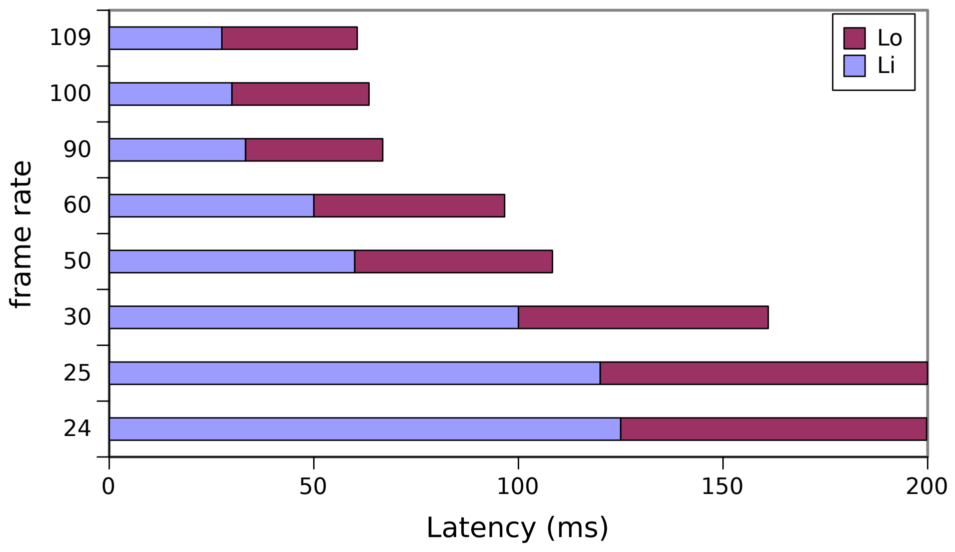


Figure 4.17: GPU bound, T2B2, NVENC, $n = 2$

In figure 4.17, we present latency for various frame rates as a stacked bar. As can be seen from this, at variable bit rate using NVENC with the scene we are testing, a frame rate of at least 60 fps is needed to achieve a full cycle latency of less than 100 ms. With full cycle latency, we mean the time it takes from the user sends input until it is displayed in her browser. Looking at the number of frames buffered in table 4.8, we see that as the frame rate increases the number of frames buffered also increases. This can also be seen in figure 4.17 by looking at how the ratio of L_o and L_i change with the frame rate. In section 4.8.1, we will look into using constant bit rate to lower the number of frames buffered.

We also did the same test using the x264 back-end. Even though the numbers were similar, x264 actually performed somewhat better at the frame rates we test. When measuring the bit rate we found that they were not similar at all. For the 60 fps test NVENC used 926kB/s while x264 used 1589kB/s. To test this further, we tuned up the bit rate for NVENC to 1500kB/s and sure enough, we get a much lower latency, even lower than the one we measured with x264: $L_m = 69.28ms$.

f	L_m
25	137.876 ms
60	73.3011 ms
80	57.8975 ms

Table 4.9: GPU bound, T2B2, x264, $n = 2$

4.8 Resource usage

We see three important factors of resource usage for our system: Bandwidth, CPU usage, and power usage. The problem with bandwidth is not as much a server problem as it is a client problem. We expect the server to be located in some location where Internet access is wast, such as a data center.

In section 3.4.1, we discussed the average bandwidth available in Norway and found two different claims: 6-8 Mbit/s [43] and 8.7 Mbit/s [44]. However, using an average of bandwidth may be flawed because of geographical reasons. It is highly likely that in cities such Oslo, the average bandwidth is much higher than in rural areas such as Rodøy in Nordland. In our opinion, having access to at least 20 Mbit/s in Oslo is not inconceivable, and because of this we use it as the high bar when testing latency based on bandwidth.

CPU usage is a highly important factor for the success of our streaming engine. Our goal in this thesis was to design a streaming engine which uses as little resources a possible leaving most to untouched for the source application to use. Being able to measure this was one of the main reasons why we selected Spout as our test application because it almost uses no CPU resources (see section 4.3.2). This is made possible by the fact that the entire scene is generated on the GPU with only minor interaction from the CPU.

The third important factor is how much power the streaming engine uses measured in Watts. Even though electrical power is generally considered cheap in Norway compared to other countries, power is usually expensive in data centers. This is because the cost of transferring the heat produced by the hardware out of the data center and replacing it with cool air is usually included in the price, in addition to UPS and other backup generator systems.

4.8.1 Bandwidth

When testing latency we discovered that the bit rate has a significant effect on the latency. This put us in a difficult position because bandwidth is i finite resource, especially when streaming over the internet. On one hand we want the latency to be as low as possible, on the other we want the stream to be as small as possible. The fact that the latency drop on higher bit rate strengthens our theory that the browser buffering is not done by counting frames but rather on bytes received. As with all the other tests done with latency, the numbers we present are gathered when using Google Chrome as our client.

Bit rate	L_m	L	B
2.06 Mbit	114.69 ms	131.36 ms	4.88
4.11 Mbit	96.92 ms	113.59 ms	3.82
6.16 Mbit	79.51 ms	96.18 ms	2.77
8.22 Mbit	75.99 ms	92.65 ms	2.56
12.36 Mbit	72.17 ms	88.84 ms	2.33
14.39 Mbit	65.43 ms	82.10 ms	1.93
16.45 Mbit	65.35 ms	82.02 ms	1.92
24.67 Mbit	64.22 ms	80.89 ms	1.85

Table 4.10: Effect of bit rate on latency in Google Chrome on a 60Hz stream, T2B2 NVENC

In table 4.10, we see that as we increase the stream's bit rate, the number of frames B buffered decreases up to about 14 megabit per second. From there we see no or little gain in increasing the bandwidth. To get a full cycle latency L less than 100 ms, we have measured that we need a stream bit rate of about 6 megabit per second. At 1.93 frames buffered, the full cycle latency is calculated to be 82.10 ms which seems to be as low as we can go using Google Chrome as client with a HTML5 video element.

The reason for the impact of bit rate is, as mentioned earlier, most likely due to the browser buffering in bytes, not in number of frames.

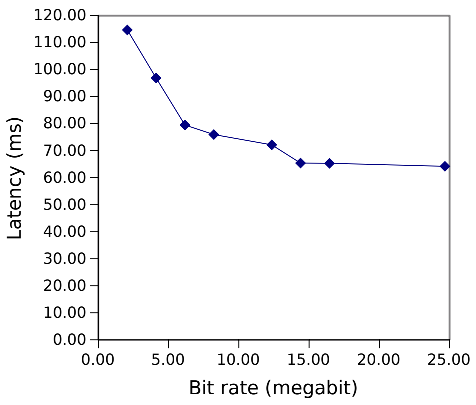
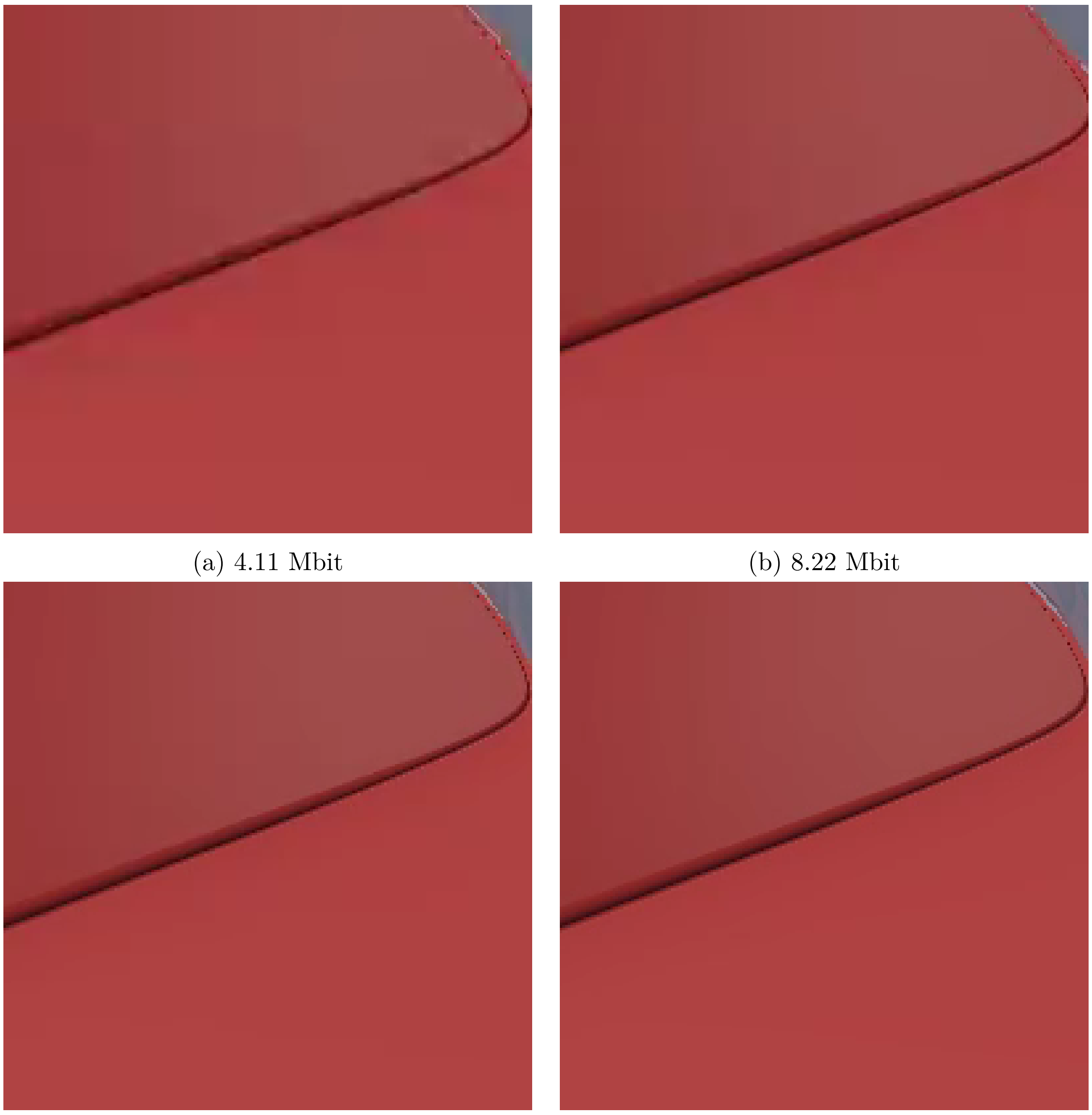


Figure 4.18: Bit rate vs L_m, T2B2 NVENC

Incidentally, the visual quality also seems to reach a peak at about 14 megabits. This is of course an unsubstantiated claim as it depends very much on the type of video streamed and is judged subjectively by us. In figure 4.19, we show a comparison of a region where the difference in bit rate is very noticeable on the visual quality. To save space, we have only included half of the samples.

When comparing 4 Mbit/s (figure 4.19a) to 8.22 Mbit/s (figure 4.19b), we notice that the image is much smoother with less noise from quantization. There is even less of this noise in the 14.39 Mbit stream (figure 4.19c, while it is difficult to spot any increase in quality from 14.39 to 24.67 Mbit/s (figure 4.19d).

The images were generated by cropping a 200x200 region from a full size 1920x1080 frame. They have since been scaled to make the effect seen more accessible on print.



(a) 4.11 Mbit

(b) 8.22 Mbit

(c) 14.39 Mbit

(d) 24.67 Mbit

Figure 4.19: 200x200 extraction at various bit rates, NVENC

4.8.2 CPU resource usage

To measure the CPU usage of a process, we use what the system tool ‘ps’ claims is the usage. The exact command we use is:

```
ps au|grep -i glstreamer|grep -v grep |grep -v sudo| \
 awk '{print $3}'
```

This extracts the CPU usage from the ps-line associated with our program called glstreamer. We then wait for this to stabilize use that as the CPU usage percentage. When the program is running an external computer connects to the stream. This is done using ‘curl’ to make sure the external computer downloads the stream without creating any trouble for the server. Curl is a command line tool and library for handling HTTP connections. We are piping the downloaded stream directly to the null device on the client computer. The CPU used for testing is an Intel Core i7-2600, 4 cores plus hyper threading, running at 3.40GHz. A CPU usage percentage of 100% means that 1 core is running at 100%. This computer has 4 cores, and 8 logical cores as hyper threading is enabled.

The numbers gathered from running NVENC is presented in table 4.11. A bright and observant read may notice the close correlation between the frame rate and the CPU usage. One may even feel compelled to say that the frame rate is equal to the CPU usage. This is, however, a flawed conclusion as it may only hold for this specific processor with this specific scene rendered. CPU usage, as presented here, only tells us that the system does not use much resources and it scales almost linearly with the frame rate.

fps	NVENC	x264
25	25.9 %	111.2 %
30	31.9 %	132.6 %
50	53.1 %	215.3 %
60	61.9 %	211.3 %
80	75.8 %	257.8 %
85	79.2 %	-
108	98.0 %	-

Table 4.11: CPU usage of GPU bound, T2B2, NVENC/x264

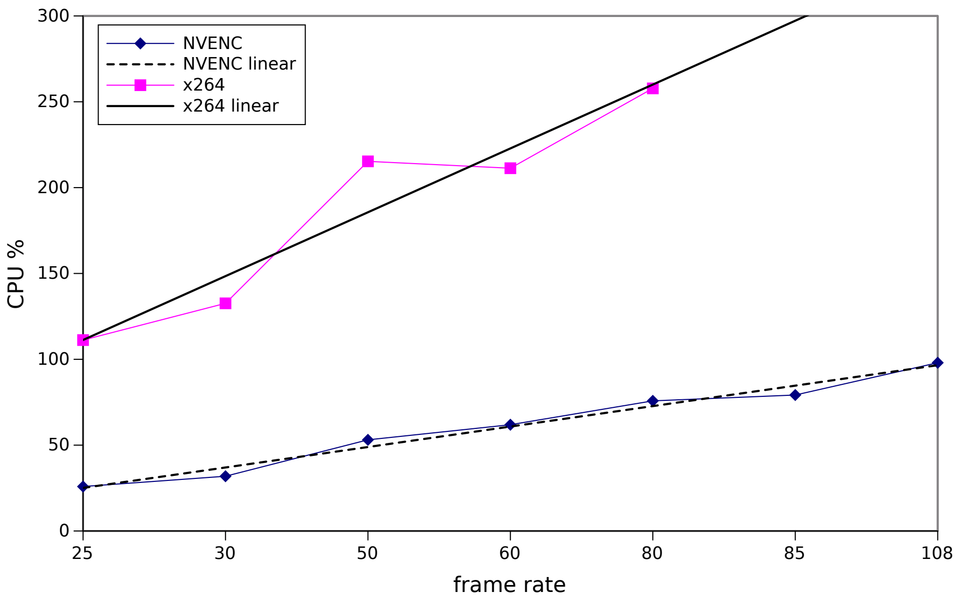


Figure 4.20: CPU usage

4.8.3 Power usage

To measure power usage when running Spout inside the streaming engine, we used an APC Rack PDU (Power Distribution Unit) 2G specified to have an accuracy of ± 3% of the reading. Because this equipment was designed for measuring the power usage of entire racks stacked with computers, it is only able to measure in terms of kW (kilowatt) with two decimals. This means that we only are able to know the usage within tens of Watts.

Yet another flaw with the APC Rack PDU 2G is that it will round every reading less than 0.5 A to zero, or everything less than 120 W. To work around this we connected a lamp specified to use 120 W to the same PDU and let it heat up. When the lamp was warm, we measured to use a stable 0.14 kW which we subtract all the following readings. The next thing we did was to measure the computer's idle power usage which we found to be 0.05 kW. This has also been subtracted from the readings we present in figure 4.21.

As with all the other tests performed on the streaming engine, this test was also run on a quad core Intel i7-2600 running at 3.4GHz with an Nvidia GTX 750 Ti as the GPU.

$$P_{\text{lamp}} = 0.14kW$$
$$P_{\text{idle}} = 0.05kW$$

We then tested running the application at various frame rates using both NVENC and x264 as encoding back-end. What we found was that at low frame rates, such as 25 and 30 Hz, the difference in power usage is insignificant, but as the frame rate increases the difference increases notably. At the highest frame rate we were able to run the streaming engine on x264, the software encoder uses almost twice as much power as NVENC.

The most interesting frame rate will be 30 Hz for low end applications and 60 Hz for high end applications. At 60 Hz we see that NVENC only uses 20W, while x264 uses 50W, more than twice the power usage. This proves that using the hardware encoder as back-end scales much better. If we were to run 10 instances of this application at 60Hz, the NVENC back-end is expected to use only 200W while the x264 back-end would have used 500W. At a cost of NOK 1200 per kW/month, the NVENC would solution would cost about $1200NOK/kW/month \times 0.2kW/month = 240NOK$ while the x264 solution would cost about $1200NOK/kW/month \times 0.5 = 600NOK$. We conclude that there is a significant amount of money to save when using a hardware encoder versus a software encoder.

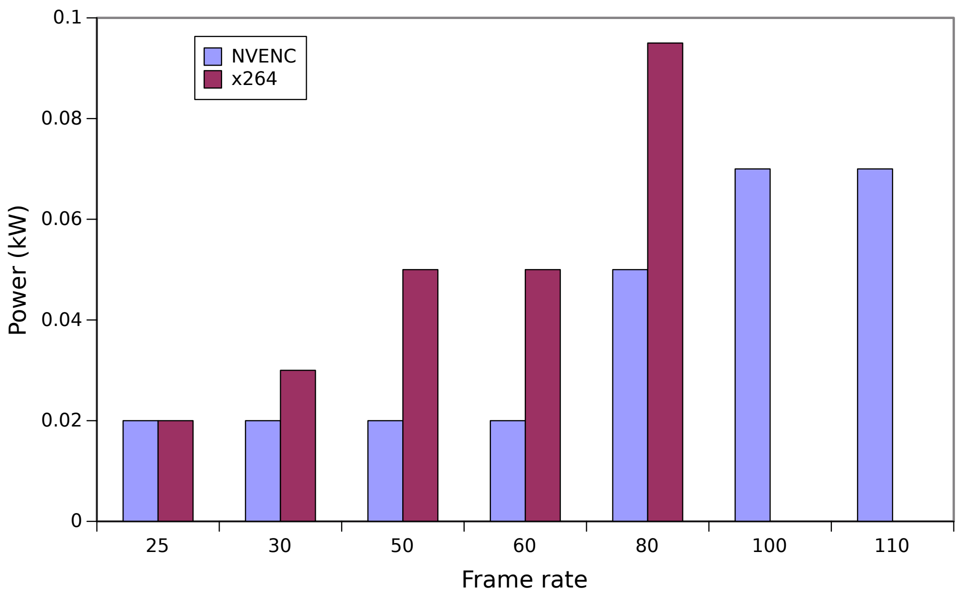


Figure 4.21: Power usage of the streaming engine at various frame rates

4.9 Summary

In this chapter we have presented the design for an engine capable of streaming OpenGL graphics to a client web browser while receiving user input through a WebSocket. The design was divided into three connected parts: the inner pipeline, the outer pipeline, and the HTML front-end.

The inner pipeline took care of reading the source application's frame buffer, converting it from RGB to NV12, encoding it and passing it along to the outer pipeline. The outer pipeline's responsibility was to create an HTTP server and managing the incoming connections, including streaming the video from the inner pipeline to the client. The HTML font-end was just a simple HTML page served by the outer pipeline containing an HTML 5 `<video>` element and JavaScript for capturing keyboard and mouse input and sending it back to the server.

To convert the OpenGL RGB-formatted framebuffer to NV12, we used an OpenGL Compute Shader. We also tried using an OpenCL kernel, but this added unnecessary complexity because of lack of interoperability between OpenCL and CUDA-buffers, needed for transferring the buffer to NVENC.

For encoding, we created a back-end for both the NVENC hardware encoder, and the x264 software encoder. Transferring a NV12 buffer to NVENC was done through OpenGL/CUDA interop, while transferring a NV12 buffer to x264 included downloading the buffer from GPU memory to system memory. To ensure high frame rate when downloading the NV12 buffer, we tested several ways and ended up with using OpenGL Pixel Buffer Objects (PBO) and the glReadPixels function.

Testing showed that for different scenarios, different setups of the pipeline was preferable for best performance. For both encoders with GPU bound applications, T2B2 excelled, while for CPU bound applications T2B3 was a good compromise.

When streaming to the browser, we found that increasing the frame rate which the MKV claimed the stream to be decreased the latency. The golden number seemed to be 500% of the actual frame rate. The browser buffer was also affected by the bit rate of the stream, where higher bit rate was preferable. When streaming 60 frames per second, the lowest bit rate was reached around 14 megabit.

When we explored the scalability of the streaming engine, we discovered that using NVENC as back-end used much less CPU resources and power.

In the next chapter, we will put the streaming engine we designed and implemented in this chapter to the test by running a few case studies on it. The first case study will be to enable the Bagadus virtual camera to run using our streaming engine, and look into how it scales. Later, in chapter 6, we will try running Quake III Arena and measure the latency we are able to achieve on a fast paced first person shooter game.

Chapter 5

Case Study: Bagadus

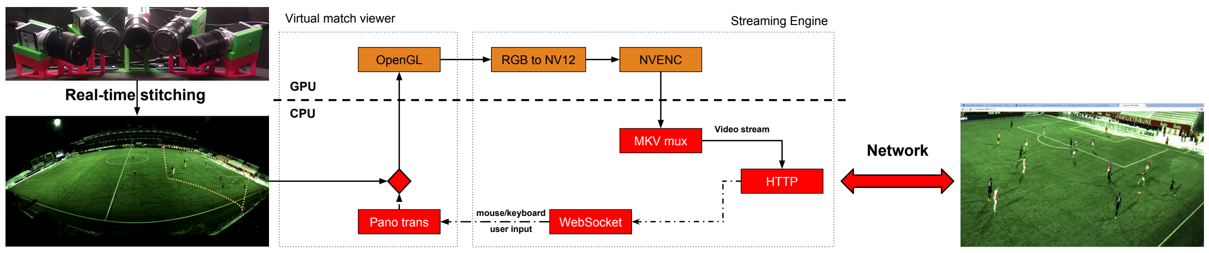


Figure 5.1: Virtual match viewer

The Bagadus project [6, 64–66] is a research project in soccer analysis using player position tracking technology and video recording. The Bagadus recording rig consists of five cameras in a matrix. Video streams from the cameras are stitched into a panorama image [7], which in turn is used for creating a virtual camera playback of the matches [8].

Every stage in the Bagadus pipeline have been optimized to run in real-time. The stream we did the tests on for this case study was recorded at 30Hz, about half the frame rate which the streaming engine was designed for. At such a low frame rate, latency is expected to be high.

In this case study we modified the virtual camera player by Vamsidhar et. al. into using our streaming engine as the video sink. The performance of the system was measured in terms of CPU usage, encoding latency, and power usage. We included tests using both the NVENC encoder back-end and the x264 driven back-end. A conceptual flow diagram for the entire system is included in figure 5.1.

The test system is the same as used during testing of the pipeline in chapter 4. An Intel i7-2600 running at 3.4GHz with an Nvidia GTX 750 Ti.

5.1 CPU Usage

Just as in section 4.8.2, we probed the CPU usage reported from the ‘ps’ system tool on Linux. When testing CPU usage, we ran the virtual camera viewer inside our streaming engine at both 720p and 1080p. Just as expected, we found that CPU usage when using the x264 back-end was almost double that of when using NVENC.

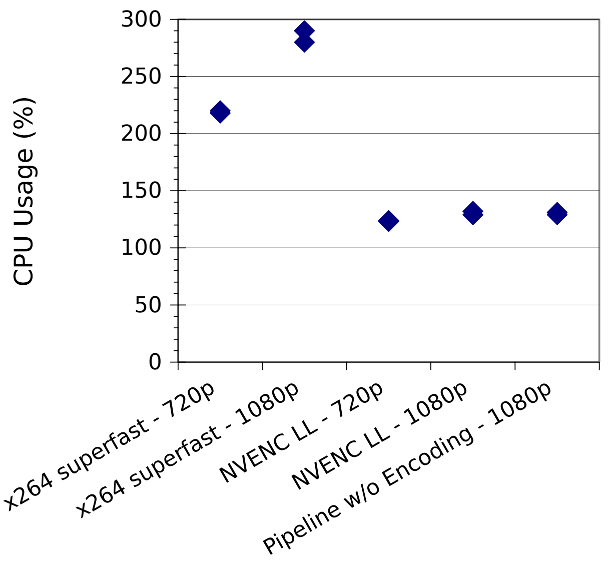


Figure 5.2: CPU usage

For comparison, we were unable to measure any difference between using NVENC as encoder and not encoding anything at all. This was expected, as encoding using NVENC is fully offloaded to the GPU and the CPU is not affected by it. We also see no significant difference between 720p and 1080 when using NVENC. The minor difference there may be connected to actually handling a larger output stream on 1080p, i.e., sending the encoded stream to the client. However, that does not explain why the pipeline without any encoding uses more CPU than a 720p stream. We must therefor conclude that the difference is caused by errors when measuring.

For applications with similar CPU utilization to the Bagadus virtual camera, encoding using x264 almost doubles the CPU requirements. Running several instances clearly scales much better on NVENC.

We also tested the scalability of the system by running several instances of it on the same computer. With the x264 software encoder back-end, we were able to deliver to streams in parallel while maintaining the 30 frames per second frame rate. When using NVENC on the same machine, we were able to deliver four concurrent sessions while still maintaining the frame rate.

5.2 Power Usage

The power measurements can be seen in figure 5.3. The data is collected by monitoring the entire system while running our delivery pipeline. To measure the power consumption, we used an APC Rack PDU 2G specified to have an accuracy of ± 3% of the reading. From the measurements, we observe that the NVENC pipeline uses considerably less energy compared to the software encoder. We also observe that the power usage using NVENC does not increase much when we change the resolution.

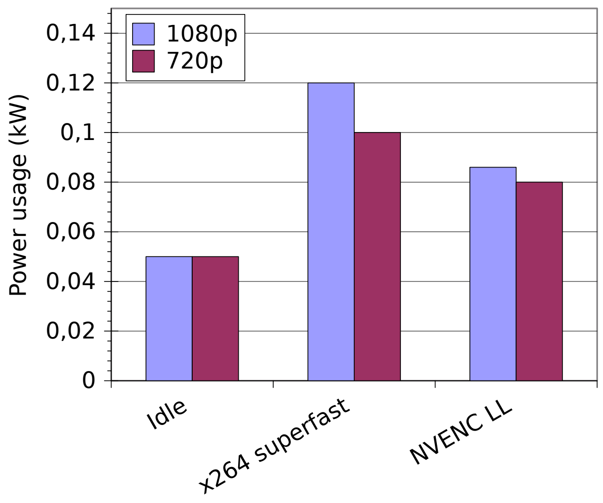


Figure 5.3: Power usage

5.3 Encoder Latency

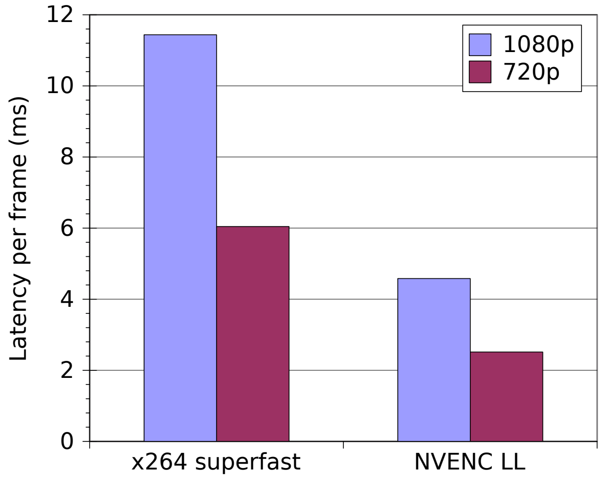


Figure 5.4: Encoding latency

In this case study, we did not look into the user input latency, only the latency from we start encoding until we have the encoded frame in system memory. For x264, we do not include the time it takes to download a frame, we only look include the time the encoder is actually encoding. Because this test is running at a very low frame rate, we were able to hide the framebuffer transferring in the rendering stage of the pipeline.

The latency of the encoder is measured inside the running system, i.e., it comprises the time it takes from when a frame is sent into the encoder component until it is ready. In figure 5.4, we observe, as expected, that NVENC is more than twice as fast compared to x264 superfast.

In both cases, we see that encoding 1080p takes about twice as long as encoding 720p. This is expected as there are more than 2 times the amount of pixels in a 1080p frame compared to a 720p frame (2.1 megapixels / 0.9 megapixels).

5.4 Quality

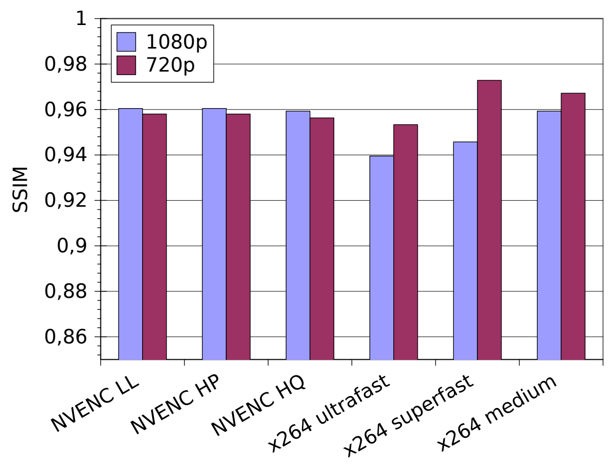


Figure 5.5: Visual quality

The test was performed by first recording the output from running the application to a raw YUV420 file. The recorded video was then run through the encoders with the same settings as if they were done online in the streaming engine. Visual quality was then measured using the SSIM algorithm, as explained in chapter 3. The encoders were configured to output a 14 megabit video stream for 1080p stream and 7 megabit for the 720p stream.

Results from the test can be seen in figure 5.5). We found that the quality of all three NVENC presets are similar, and they perform between x264 superfast and medium. We also used subjective testing of the videos, and we found that x264 ultrafast did not produce adequate video quality compared to the other presets. There was no noticeable difference between the other presets.

We noticed that x264 produced much better quality on 720p than 1080p, while NVENC, in contrast, produced somewhat better quality on the 1080p.

5.5 Summary

While this case study was very basic, it served as a verification of the system in that it works and it does so efficiently. We found that in all the tests we performed on the system, except for the quality tests, NVENC performed better than x264.

The CPU usage of our streaming engine is in general very low, especially when using the NVENC encoder. Using NVENC, we were able to run twice the amount of instances of the service, while still maintaining the frame rate. Power usage when using NVENC is also considerably lower than when the x264 encoder is used. As this test does not include transferring the framebuffer to system memory, what we see is only the latency from encoding a frame. If we were to include the transfer, x264 would be much slower.

The only test we ran on which the x264 encoder performed better was when measuring visual quality. When streaming on 720p using the x264 superfast preset, we measured somewhat better quality than NVENC on the same input. On 1080p, SSIM values were consistently higher for NVENC.

In the next case study, we will look into if the overhead and latency of our streaming engine is low enough to stream a computer game. Focus will be on the interaction latency experienced by the user, and compare it to previous studies in latency perception.

Chapter 6

Case Study: Quake III Arena

In this case study we will look into how our streaming engine performs on a real life application, such as Quake 3 Arena. Our focus will be on how latency is experience on home networks using consumer laptops, such as the MacBook Air from mid 2012.

6.1 Limitations

There are several parts of the system that can be tested, but we have selected only a few. We are only experimenting the T2B2 pipeline setup from section 4.3.2 which means that there are two threads running. The first thread is responsible for rendering the game, converting the framebuffer and sending it off to the encoder. The second thread is waiting for a frame to return from the encoder, multiplexing it in a MKV container and adding it to the output queue of the client. The threads are running in lock step, which means that both threads have to finish before the next iteration.

Also, in this experiment we will only be using the Nvidia NVENC hardware encoder and not x264. Doing both would be too time consuming as running a test takes a while and requires human interaction, i.e., playing the game. While the game is fun for a while, it gets old after a few weeks.

In chapter 3, we found that NVENC is not able to encode 2160p at 60 frames per second with sequential encoding. Because our interest is in low latency, we will only include 1080p and 720p in our tests. Instead, we will be looking at how the engine performs in an environment closely matching what we are convinced is a common Norwegian home. We are interested in seeing how latency is depending on the network connectivity using both WiFi and Ethernet cable at different frame rates and resolutions.

As for bit rate, at all our tests we try as good as we can to run at 14 megabit per second. This was selected due to our findings in section 4.8.1.

6.2 Quake 3

Quake III Arena (Q) is a first person shooter by Id Software released in 1999 which was, as with all Id Software games, considered technically advanced for the time. The source code was released under the GNU General Public License (GPL) in 2005 and has since been modified by the community in several ways, e.g., making it possible to compile with modern versions of GCC (The original required GCC 2.95) [67].

By today's standards the game looks dated but is still relevant as it is to this day one of the most high-paced action games where low latency is vital to the experience and playability.

We decided to modify ioquake3 [68], a modified version of the original Quake3 engine where support for the AMD64 architecture and a proper SDL back-end are added. Because of the high quality of the code, modifying the engine to use our streaming back-end instead of SDL was very easy. The platform dependent parts of the code is clearly separated from the game logic and the render code which meant that all we had to do was implement a few function calls, mostly by removing all SDL code. As we are using a modern GPU, the functions which query for available OpenGL extensions was set to always return true if the extension string included ARB.

The biggest difference in how our streaming engine is designed and how Q3 works is that our streaming engine is meant to control the rendering application while Q3 was designed to control the OS layer. This was not a major hindrance as we could just swap out the main entry function of Q3 with our own and run the game loop as we see fit. Listing B.1 shows how the original main function looks while Listing B.2 shows how this was fitted into our framework.

As Q3 is a first person shooter (FPS), we had to do some changes to the client side to capture the mouse properly. Using the JavaScript Pointer Lock API [69] and browser fullscreen, we were able to capture mouse input properly. The issue with the normal mousemove event is that as the mouse moves to the border of the view port it will not be able to move any further which means that the game is basically unplayable as the user has to plan mouse movement. With the Pointer Lock API, the mouse becomes invisible and events only send relative mouse movement.

6.3 Lab tests

To establish a base line of how well the system possibly can perform through the internet, we first performed a series of preliminary tests in our computer laboratory on a local area network (LAN). The game and the streaming engine was run on the same hardware as all the tests in the previous chapter: a four core Intel i7-2600 running at 3.40 GHz. The Google Chrome web browser client run on a computer with similar hardware in Linux.

In test, the game was running at 120 frames per second, somewhat more than is expected from a high end modern computer running a modern triple-A game. To measure latency we tagged all frames with a green dot in the upper left corner, except for some frames which are tagged with a black dot. Using JavaScript, we can detect when the dot is black and send a message back to the server which then computes the latency from it started rendering that frame until the frame was displayed and the message was sent back. This is exactly the same technique as described in section 4.7.

Our willing test subject was Asgeir Mortensen, who used to play a lot of Q3 in his younger days. He was asked to play the game for several minutes while the system was logging the latency. The values from this test run are presented in table 6.1. We found that the median L_m was 46.72 ms. Using the formula established in section 4.7, the interaction latency with the system is $L = 1/f + L_m = 1s/120 + 46.71 \times 1s/1000 \approx 55.04ms$.

Avg L_m	Med L_m	Med L_m lo	Med L_m hi
47.98 ms	46.72 ms	43.75 ms	51.06 ms

Table 6.1: Measured latency for Mortensen's quake 3 game at 120 fps

We did a few more tests with different hardware in the lab before starting with the remote tests. The results from these can be seen in figure 6.1 together with the Mortensen test. While the outlier in Ljødal WiFi probably is mostly due to WiFi connectivity problems, the outlier in the Mortensen test will be explained later in section 6.5. The general trend in the lab environment is that median latency L_m is about 47 ms.

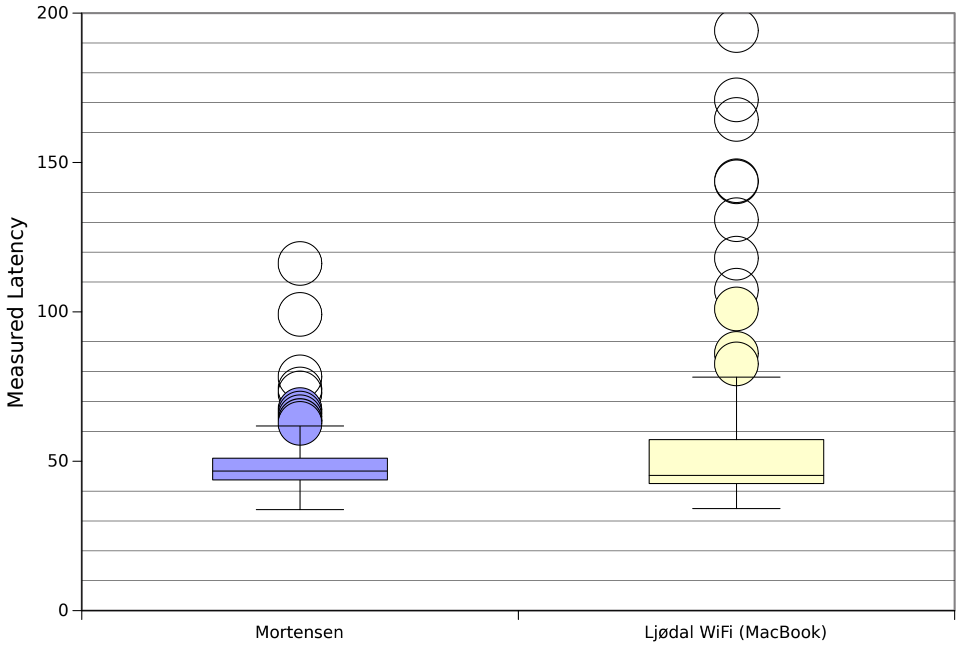


Figure 6.1: L_m from playing ioq3 in the lab

6.4 Remote testing

Because of a rigid network policy at the University of Oslo, to perform our remote tests we co-located the server used for all the previous tests with Blix Solutions [70] in Nydalen, Oslo. Blix Solutions was kind enough to offer us this service free of charge while working on this thesis. The server's IP connectivity is 1 gigabit/s which means that the server's internet uplink should not be an important factor for the latency measured.

The testing location is a residential apartment on Grünerløkka, central Oslo. Internet here is provided by GET which is a very popular internet provider in the Oslo area. We measured the internet connectivity by using `http://www.speedtest.net` to be 51 megabit/s downstream and 11 megabit/s upstream. Incidentally, the server this test was run against on speedtest.net is also hosted by Blix Solutions.

We measured the ping round-trip latency between the location and the server we were running the streaming engine on to be 7.103 ms when connected using an Ethernet cable and 10.025 ms when connected using WiFi. This means that the WiFi added three additional milliseconds to the round-trip. Tracing the route from the testing location and to the server using the tracepath tool commonly available on Linux distributions, we found that there were 8 hops. Two of those were inside GET's network, one was portlane located at the NIX at the University of Oslo, and three were routers inside Blix' network.

To perform the tests we used two computers. One is an older AMD system (Phenom II 1050T) which is connected to the home router through an Ethernet cable, the other is a MacBook Air from mid 2012. The AMD system is running Linux and has an AMD GPU supporting H.264 decoding enabled in Google Chrome. The MacBook Air is running OSX and has H.264 decoding enabled in its GPU.

6.4.1 720p

WiFi (MacBook)

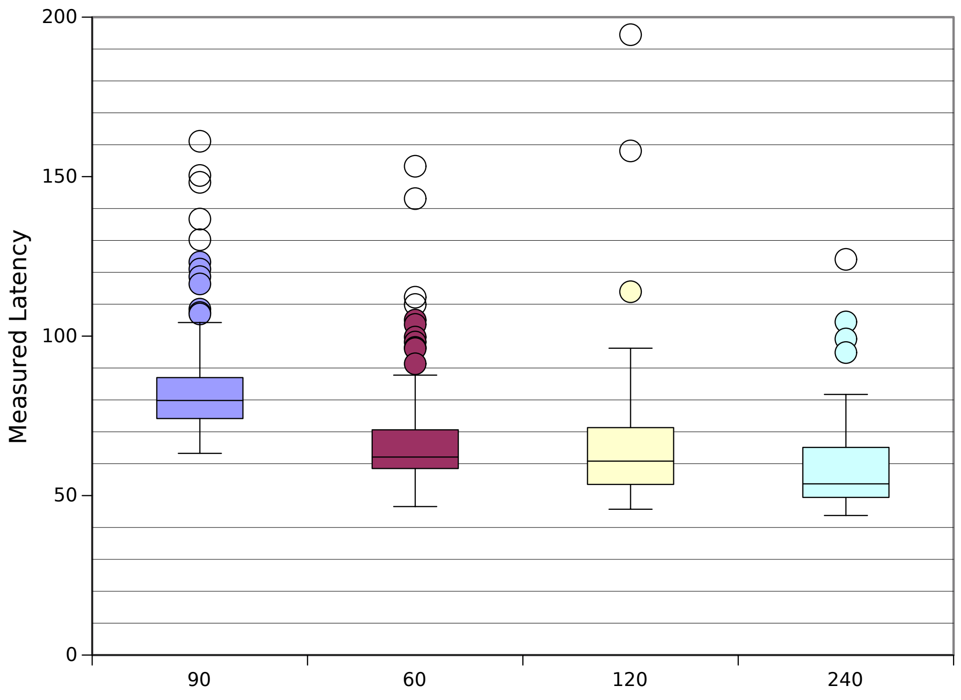


Figure 6.2: L_m from playing ioq3 remotely at 720p over WiFi

Normally, a graphical computer application runs at 60 frames per second. We see here that at that speed, the median measured latency is 80 ms. Adding the 16 ms for the input wait, we end up with an interaction latency of about 96 ms. In the next chapter, we will discuss how this affects the human perception of the game.

As with the lab tests, we that there are a lot of outlier when we are testing WiFi. When this happens, mostly because of packet loss, the user can experience some “lagging” before the stream catches up with the live stream.

What is more interesting is that as we increase the frame rate, the measured latency decreases. We know from the discussion about latency in section 4.7 that the interaction latency should decrease with the frame rate as the interval one frame represents is shorter, i.e., $1/f$ seconds. In our engine, when a frame has finished rendering and is encoded it will be sent to the client at once without waiting for the rendering thread to finish it’s next frame. In that sense, frame rate should not affect L_m, only L. We think the reason for what we are seeing is that the browser is more “willing” to display frames faster if it receives more frames to always keep up with the stream.

Since Q3 is an old game, modern hardware have no problem running it at very high frame rates. We were even able to run and stream the game at 480 frames per second on the lab hardware, but on the MacBook that proved challenging. However, running the game at 240 fps worked extremely well and with a very impressive L_m of 50ms. At that frame rate the interaction latency is 54.17 ms. Still, we see it as unrealistic that modern games would be able to run at that frame rate as most are optimized for either 60 or 30 fps.

Ethernet (AMD)

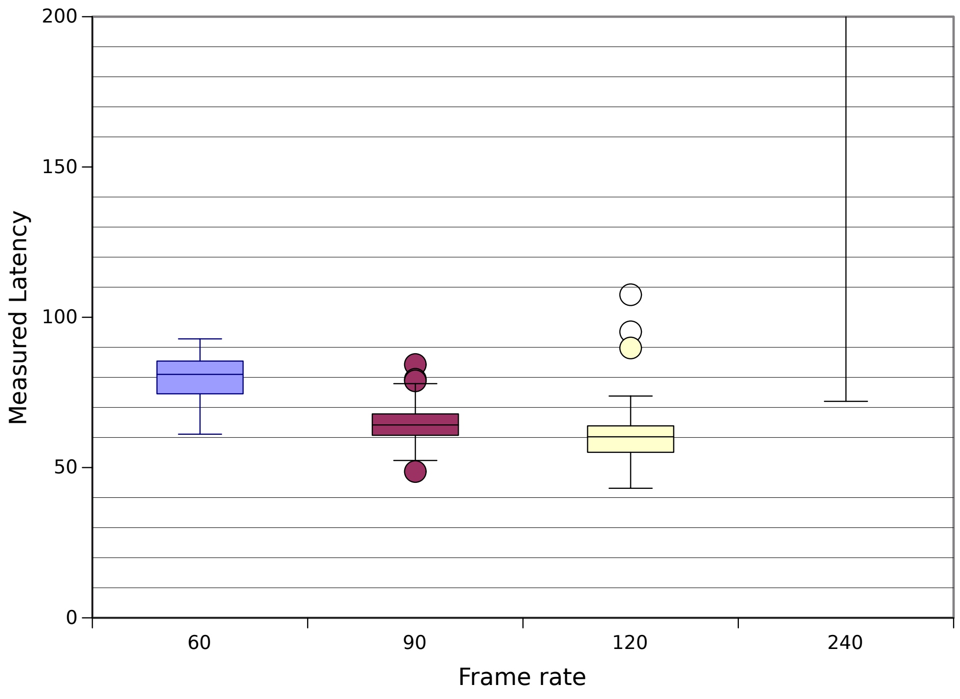


Figure 6.3: L_m from playing ioq3 remotely at 720p over Ethernet

In this test we were running the game on a AMD Phenom II 1050T, bought in 2010. At the time writing this, the computer is 4 years old and the only upgrade done on this machine is to upgrade its GPU. The GPU on the system is a AMD HD7970, which is really fast although it produces a lot of heat. The test is run on Linux, and has the advantage of being connected through a Ethernet cable which, in the ping tests, had a round trip time to the server of 3 ms less than the WiFi.

Even though it had the advantage of better connectivity, the L_m we measured was somewhat higher than in the WiFi tests. There are several reasons why this may be. One is that the CPU is older and does not have all the newer features of modern Intel CPUs. Another is that the H.264 decoder is on a discrete GPU, while the MacBook's decoder is built into the CPU (QuickSync). However, we have found no sources to back this claim.

In the case of 240 fps, the AMD decoder was not able to keep up with the stream and the latency we measured was in order of seconds, not milliseconds. Still, the 120 fps test ran very smoothly while having an L_m of almost 10ms more than the MacBook on WiFi.

6.4.2 1080p

WiFi

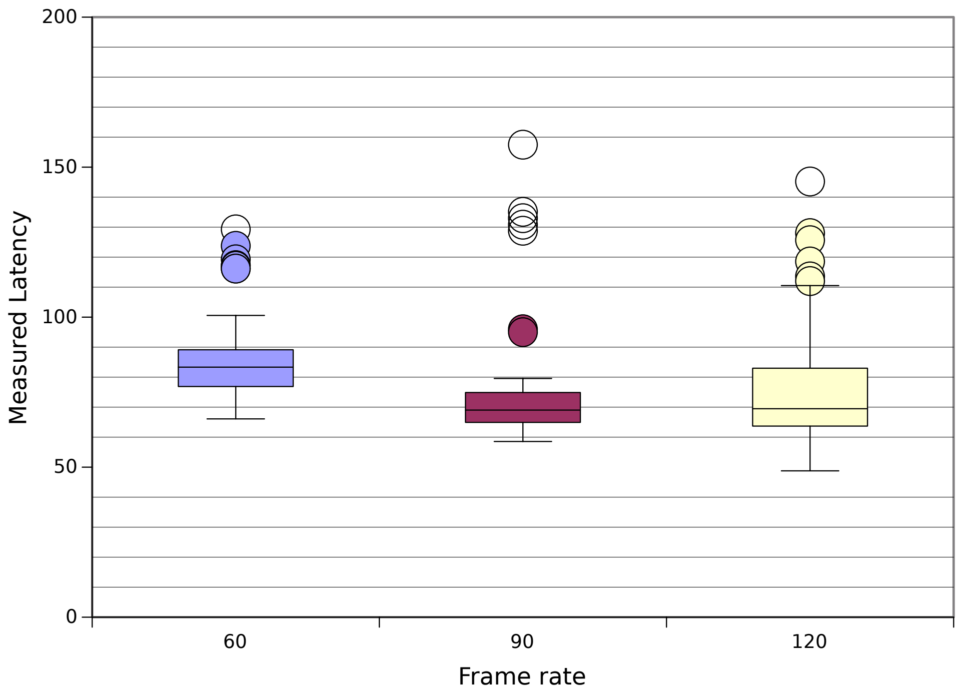


Figure 6.4: L_m from playing ioq3 remotely at 1080p over WiFi

In the 1080p test, none of our test computers were able to decode the stream at any higher frame rate than 120 fps. Even at 120 fps, we experienced a lot of jitter which also is visible in figure 6.4 when looking at the difference in upper and lower median.

In general, we found L_m for 1080p to be about 5 to 10 ms higher than for the equivalent frame rate on 720p.

Ethernet

In this test, we can see the same gradual improvement by frame rate until we reach 120 fps. There we see that the system is not able to play back the stream as fast as with the lower resolutions. At such a high frame rate, the data is actually $1920 \times 1080 \times 3 \times 120 = 711 megabytes$ per second when the stream has been decoded to RGB.

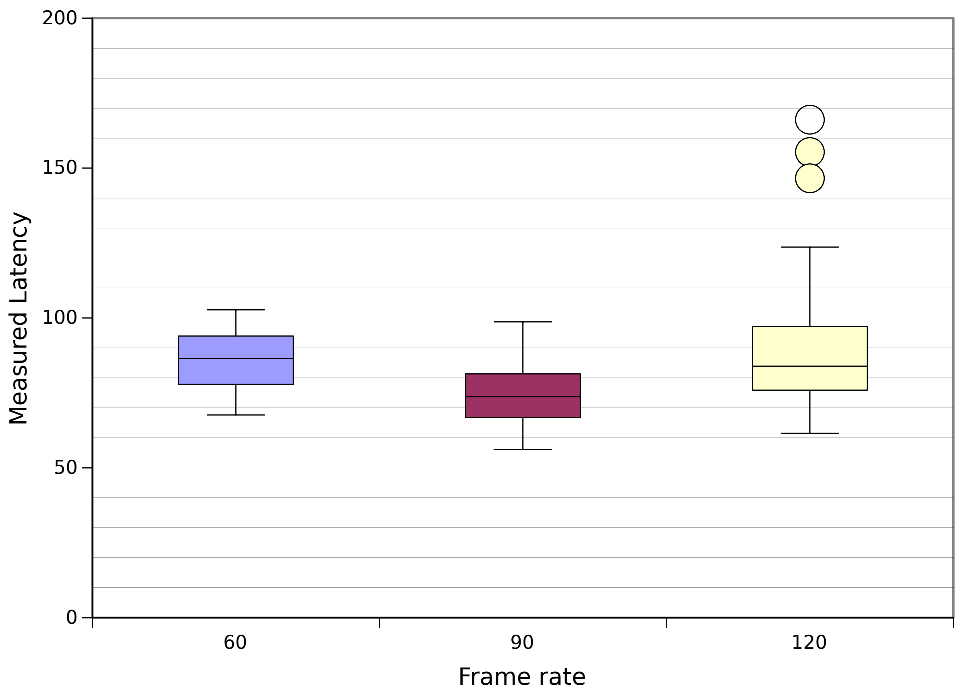


Figure 6.5: L_m from playing ioq3 remotely at 1080p over Ethernet

6.5 Challenges observed during testing

6.5.1 Rate control

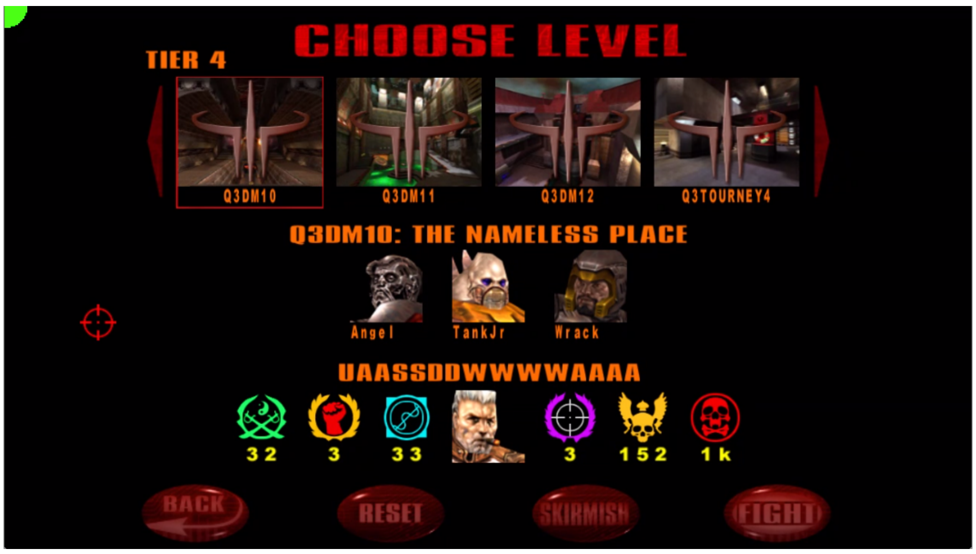


Figure 6.6: The Quake III level selection menu

In the previous section we saw some outlier that were hard to explain why happened. When looking at the stream of L_m samples while they took place we were able to pin-point exact types of scenery causing them. The most obvious place where they happened was in the game menus which were very unresponsive and changes in mouse position lagged far behind the actual mouse movement.

At first, this was though to be due to the game using an old rendering technique when drawing the menus where it would only update sections with changes. Such a technique would match poorly with the way our engine expects the game to render to several framebuffers.

After reading through the source code of ioq3 we were able to dismiss this theory. It was not before measuring the bandwidth used when playing the game that we discovered exactly why we were seeing the effect. The reason for it was that because each frame when rendering the menu is so similar to the previous, the H.264 encoder was able to encode it at a very low bit rate, even though the encoder was asked to encode at constant bit rate. This is related to the bandwidth issue we discussed in section 4.8.1, but to a more extreme degree.

While recording L_m values we removed all samples from when we were in the menus, and only selected those from when playing the actual game. When looking at the data, we noticed that there were still some unexplainable outlier in the data set. Those are from when there are no motion in the scene, e.g., when the player dies and the score board is rendered while the camera is facing a wall or the ground.

Solving the challenge may be problematic as we then have to interfere with the encoder's rate control mechanism. A simple and naïve solution would be to detect that the output of the encoder decreases and then add some padding in the MKV to overcome the browser buffering. We see a few issues by doing that which includes sudden bursts in bit rate if the frame detected to be tiny was just a fluke and not a part of a series of similar frames. If we then added the padding, the encoder would still think that as it saved some bits in the previous frame it can use those in the next one to increase the visual quality.

A better solution seems to be using temporal prediction scheme where the engine detects a trend in well compressed frames and then starts to add padding. We have not looked into the details of how this algorithm would be constructed.

6.5.2 Visual Quality

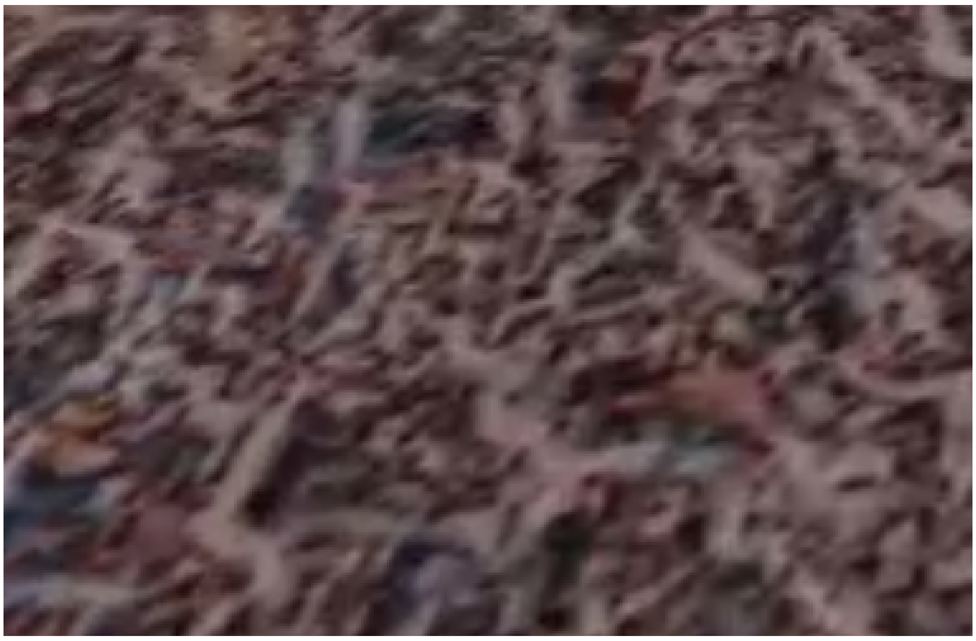


Figure 6.7: Floor texture which the encoder struggles with

Some maps in Q3 have textures which the encoder struggles to encode properly and results in poor visual quality of the stream when there is motion. This results in a rather funky visual effect where some places in the map have very bad visual quality while other places have superb. Figure 6.7 shows an example of the texture messing with the quality.

Another strange effect the player may notice is that while he is moving fast while looking around, the visual quality may be low. But if the player stops, even for less than a second, the quality progressively gets better. This is of course due to the use of constant bit rate where the encoder is not able to cope with the major changes in scenery between the frames.

6.6 Perception of latency

How humans perceive the latency of a system is still not fully understood. There have been several studies on the subject, but none, as far as we know, have been fully conclusive.

When talking about game latency there are two very different situations they are mentioned. One is the client/server latency, usually important when playing a game locally where it connects to a remote server. In this kind of gaming, the game is just a client which renders the frame while the server keeps all state. There have been a some studies into how the latency between the client and the server affects this and methods of reducing the latency by doing local calculations mirroring the ones happening on the server.

The second type of game latency is more related to the one we are studying in this thesis. It is the time it takes from the user interacts with the game until he or she can see the effect on their display. Usually, the frame rate at which the game run at is an important factor. Everybody have tried playing a game which the computer running the game is too slow to handle, which results in a choppy game play and it is not very pleasant experience. Solving the frame rate challenge is often done by removing some complexity from the rendered scene.

For our streaming engine, the latency from the application rendering is of course inherited. In addition to that time, we have added some latency from encoding the frame, sending it to the client over a network, and then decoding it before displaying it. Because our client is just a simple H.264 decoder, we can not utilize the local rendering technique used for multiplayer games to lessen the effect of latency.

6.6.1 Previous Work

One recent study done by Jota et. al. [71] on mobile touch devices, strengthens the theory that as latency increases, productivity decreases. They also found that the performance is more severely affected by latency on smaller targets which matches closely the prediction made by Jake Cannell in his blog post about latency in cloud gaming [72].

In Cannell's blog post he discusses the feasibility of running games in the cloud and how much latency is acceptable for the user. His key point is that gamers on consoles have been playing games running at 30 fps for some time now, and while hardcore PC gamers used to play at 60 fps would notice the difference, players seems to be contented. His findings is presented in table 6.2.

300ms>	games become unpleasant, even unplayable
200ms>	delay becomes palpable
100-150ms	limit of delay detection for full scene actions - camera panning and so on
50-60ms	absolute limit of delay detection - small object tracking - mouse cursors

Table 6.2: Cannell's latency table

An article presented by Jarschel et. al. [73] concludes that the slower the gameplay in a game is, the less latency affects the user. In figure 6.8 we can see that fast games, such as racing games, have a much steeper slope on latency than slow and medium paced games such as soccer and role playing games. However, they do not have any data for delays less than 80ms. Because this figure have measurements at zero delay, it must represent the addition delay, and not the actual delay of the game. Jarschel described the different MOS values, where 5 is best and 1 is worst, as:

> While the delay is recognized in the racing simulation and rated with a MOS value of 3, only some people detected it in the role play game and the soccer game resulting in a MOS value of 4 for both.

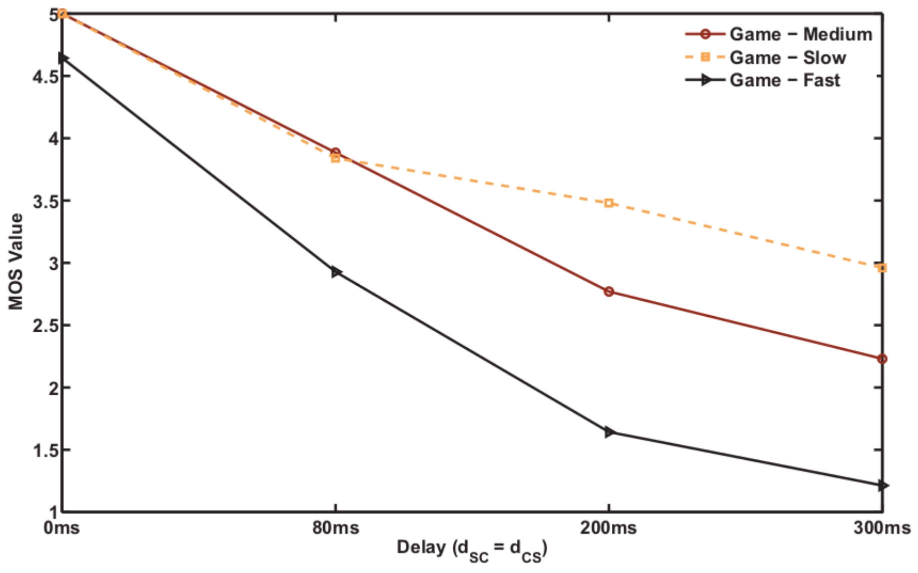


Figure 6.8: Latency / Mean Opinion Score[73]

If we compare the results from both studies, we see that they do not agree fully. While Cannell describes a limit for detectable full scene actions to be in the range of 100ms to 150ms, this would result in a MOS value of 2 - 2.5 for fast paced games according to Jarschel's study.

6.6.2 Considering Quake 3 Latency

Earlier in this chapter, we measured the latency of playing Quake 3 over the Internet. While both the server and the client was located in Oslo we managed to get the latency when running at 60 fps down to 96 ms.

If we make the assumption that Quake 3 is a fast paced game, we find that while this is well within the limit of delay detection from Channell's table, it scores somewhere between 3 and 2.5 in Jarschel's MOS scale. If we were to remote the delay from local play, e.g., $\frac{2}{f}$ at 60 fps, the delay would be 64 ms, which probably would yield a MOS of 3.5.

On the 120 fps WiFi game, the measured latency minus one frame would yield a delay as low as 53 ms. If the numbers we have gathered from measuring the latency of Quake 3 are comparable with the delays from Jarschel's study, we must say that our system is indeed able to keep the latency within an acceptable range between the recognizable 3 MOS and detectable 4 MOS. This fits well with how we would describe the subjective experience.

6.6.3 Errors

There may however be flaws in the data we measure. To measure latency, the server sometimes tags a frame with a black dot in the upper left corner. The client is programmed to detect this black dot using JavaScript and send a message back to the server. The server can then calculate the time it took from it started rendering that frame until it was displayed in the client browser plus the time it took the signal to reach the server. Frames which are not tagged will have a green dot instead, as visible in figure 6.6.

Errors may include that the JavaScript is not able to detect the white dot. We see this especially on higher frame rates, such as 120 fps, where the JavaScript does not seem to keep up with the pace. As the JavaScript is just sampling at what may be random intervals, we know that it will not be able to give accurate results and may most of the time detect tagged frames later than when they actually appeared on the client display.

6.7 Summary

We found that running ioquake3 on a remote server and streaming the video output to a client web browser located in a typical Oslo home environment is indeed possible and is quite enjoyable. It is our opinion that the game ran smoothly at 60 fps, while we preferred the experience of running it at 120 fps.

To get the best performance on the hardware tested, we found that playing the game at 720p is required. Only then was the client able to decode the stream fast enough at 120 fps to get close to an L_m of 60 ms.

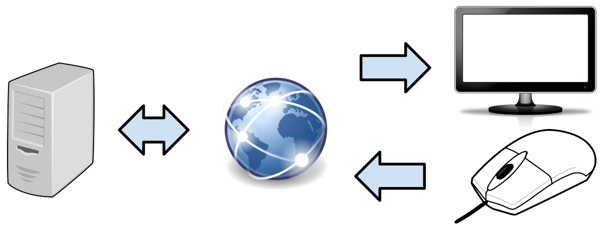
The challenge with dropping bit rate discussed in section 6.5.1 seems like a problem which is possible to fix, but requires some additional research in understanding the exact nature of it. The naïve quick fix by just padding every frame which high compression seems like a good start, but will probably not be the best solution possible.

When comparing the latency we were able to measure of Quake 3 running inside the streaming engine to some studies of perception of latency, we found that our system scores somewhere between 3 MOS and 4 MOS. According to a statement in Jarschell's study, users reported that a MOS value of 3 was recognizable, and a MOS value of 4 was detectable.

Chapter 7

Summary and Conclusion

7.1 Summary



In section 1.2, we defined the problem which we tried to solve in this thesis to be designing and implementing a high performance, remote computing service for running graphical applications, where all the client needs is a common web browser with no extensions or browser plugins installed.

To achieve this, we decided to use the H.264 video encoding format for encoding the source application framebuffer. The main reason for selecting H.264 was that many web browsers support this video format and can readily decode it when used through the HTML 5 video element.

7.1.1 Hardware H.264 encoder

In chapter 3 we tested NVENC, Nvidia's H.264 encoder, and compared it to x264, a popular open source H.264 encoder. There, we found that in terms of visual image quality, the encoder performed very similar x264 at the superfast preset. We also found that NVENC is able to encode a Full-HD (1080p) at more than 140 fps when encoding in a sequential setup. A sequential setup is a configuration where we always wait for a frame to finish encoding before we start encoding the next one. Using a pipelined approach, we were able to push the encoder up to more than 200 fps. However, in a live streaming scenario with the latency requirements we have, only sequential encoding is viable. Figure 3.4 shows the encoding performance for 720p, 1080p, and 2160p. Since both NVENC and the software x264 encoder were able to encode at frame rates higher than 60 fps, we were able to conclude that it is viable to use H.264 as video encoding format for our streaming service.

In terms of visual quality, we discovered that NVENC produces a visual frame quality of about 0.96, judged by the SSIM algorithm (figure 3.6). This value was later confirmed in the Bagadus case study (figure 5.5). We also confirmed our suspicion that the NVENC, as a GPU encoder, uses much less CPU resources (figure 3.8).

7.1.2 Streaming Engine

The main challenge in chapter 4 was to actually design a streaming engine able to read an OpenGL framebuffer, encode it, and send the resulting H.264 stream to a client web browser. Because the OpenGL framebuffer is stored in RGB, we had to translate it into NV12 which is the format both NVENC and x264 accepts as input. We first used an OpenCL compute kernel to do the conversion, but later we settled with an OpenGL compute shader in an effort to keep the API interops to a minimum as NVENC must be accessed through CUDA.

A difficult task we managed to tackle was to transfer the converted NV12 buffer from GPU memory to system memory for the x264 encoder. We tested several approaches which all were slow. In the end, we determined that using a pixel buffer object (PBO) and glReadPixels was the only way to get a sufficient frame rate of 100 fps at 1080p.

To enable the web browser to stream the video from our engine and send user interaction input back to the server we created a simple HTML page. This page included some JavaScript for capturing mouse and keyboard input and mouse input and sending it back to the server using a WebSocket. The HTML page also included a simple HTML 5 `<video>` element which opened a HTTP stream for the streaming engine output. To lower buffering done by Google Chrome, we discovered that if we increase the frame rate claimed in the MKV with 500% of the actual stream frame rate, we were able to save about 10 ms of latency at high frame rates.

The way we managed to measure latency was two-folded. Part of it we were able to calculate from knowing how the pipeline was designed. The other part was measured by tagging some frames sent with a green area in the upper left area of the frames. Using JavaScript, we were able to detect changes in this area and send a signal back to the server when those changes were detected. The server would then know when those changes actually occurred on the server side and calculate difference L_m. The process of measuring latency is described in section 6.3.

When trying to decrease the measured latency, we found that increasing the stream bit rate gave good results. Going from 8 to 14 mbps, we were able to shave of another 10ms of latency. Increasing the bit rate any further did not have any significant effect on the measured latency (section 4.8.1). Obviously, increasing the frame rate also decreased the latency. However, it did also increase buffering. This effect can be seen in table 4.8.

After having implemented the streaming engine, we measured that it did not use much resources. On the quad-core hyperthreading enabled CPU we ran the tests on, we found that on a 60Hz stream it would only use about 60% of one core when NVENC was used for encoding. When x264 was used, the streaming engine with its encoder used more than 2 cores. Without having profiled the application properly, we concluded that the streaming engine did not inhibit the source application at a high degree when using the NVENC encoder.

7.1.3 Bagadus virtual camera

In the Bagadus test-case we verified that the system actually worked by running the Bagadus virtual camera application by Vamsidhar. The program ran smoothly at 30 frames per second, and we were able to send both mouse and keyboard input to the application.

We found that using NVENC both reduced CPU and power usage. The CPU usage of running the system through the NVENC and running it through a dummy no-operation encoder was exactly the same. The lowered CPU usage meant that we were able to run more instances of the program on the same machine, compared to using the streaming engine's x264 back-end. As for power usage, we found NVENC used almost 40W less than x264, which could lower the cost of running the service.

When we tested the video quality produced by the two encoders, we confirmed the results from chapter 3. x264 produced somewhat lower SSIM results than the previous tests, but still x264 superfast and NVENC produced comparable quality.

7.1.4 Quake III Arena

In the Quake 3 test, our goal was test the latency as experience by the user of the system. To do this, we tagged some frames with a distinctive black dot in the upper left corner which the JavaScript client was able to detect. When the JavaScript code detected the dot, it sent a signal back to the server which then could calculate the time from marking the frame until the JavaScript detected it.

The results from running it in the lab were promising for further testing outside the lab. Latency measured there was below 50 ms. The remote test was conducted from a residential apartment in central Oslo, while the server was located in Nydalen Oslo.

We tested streaming both 1080p and 720p with and without WiFi, and found that the latency was highly dependent on frame rate. At 720p, we were able to get the measured latency down to 50 ms when running Quake 3 at 240 frames per second.

During testing we observed a few new challenges. One was that when streaming the menus, the latency increased as the bit rate dropped. The reason for the bit rate dropped was due to the small difference between each frame rendered resulted in very high compression from the encoder. We already knew of the effect of low bit rate in the browser, as discussed in section 4.8.1, but we had not experienced bit rate drops in our earlier tests. We made no attempts to solve this, but believe that padding small frames with null-data may be a plausible, but naïve, solution.

The other problem with the system we discovered was that when the scene contained high entropy textures, the visual quality of the encoded stream lowered. When this happens, a user may experience that the image becomes blurry where there is motion. The quality quickly regains if the motion stops. The problem of lowered quality is due to constant bit rate where the encoder increases quantization to compensate for the higher entropy in the frames. We did not attempt to solve this problem.

7.2 Contributions

During this thesis we have made several contributions related to low latency streaming and high performance computing.

The streaming engine
Our main contribution is the design, implementation, and analysis of the streaming engine, as described by chapter 4. This engine is able to stream OpenGL applications running at a server to a web browser client with very low latency at high frame rates.

Analysis of NVENC
We performed extensive testing of Nvidia's NVENC hardware encoder on two hardware generations: Kepler and Maxwell. We compared the encoder to the open source x264 encoder in terms of visual quality, speed, and power consumption.

Downloading of OpenGL buffers
For the x264 powered back-end, we needed to download an OpenGL buffer containing the frame encoded as NV12 from the GPU to the system memory. We found that using a pixel buffer object with the asynchronous glReadPixels call was much faster than mapping the buffer using glMapBuffer or similar.

Techniques for streaming to the browser
To be able to stream video to the browser we had to come up with a series of techniques for mitigating the browser buffering strategy. This included study of the effect of MKV frame rate / time codes, and bit rate.

Running Quake 3 in the web browser through a video stream
Others have compiled Quake 3 for running directly in the browser. This uses the local GPU to render the game. Instead, we stream the game, rendered at a remote computer, to the client web browser.

Measuring video latency
To measure the latency of our system between the server and the client's display, we had to come up with a way of tagging frames and detecting the tags in the browser. This was achieved through visual tags and reading a section of each video frame from the `<video>` element through a `<canvas>` to the JavaScript code, then signaling this back to the server.

Poster at GTC 2014
In march of 2014, we entered Nvidia's GPU Tech Conference (GTC) with a poster [17] describing the process of streaming OpenGL framebuffers to a client web browser. The web browser used was Google Chrome without any extensions or plug-ins. The only requirement to the web browser was that it supported the HTML 5 video-element and WebSocket connections.

Paper in proceedings
We have a paper in proceedings for the ICM conference of 2014 [18]. This paper is very similar to chapter 5, and focuses on scaling systems using our streaming engine and NVENC.

7.3 Conclusion

Our conclusion is that it is possible to create a streaming engine able to stream graphical heavy programs from a remote server to a client using only commodity hardware and open technologies. We achieved this by using Nvidia's NVENC hardware H.264 encoder, and HTML 5 elements such as the `<video>` element and WebSockets.

Through extensive testing of the Nvidia NVENC encoder, we found that the encoder is able to encode at high speeds with low power usage. When we compared the resulting video stream quality to x264 using the SSIM algorithm, our finding was that NVENC's quality is comparable to x264 running with the superfast preset. Thanks to NVENC being located close to the GPU memory, we were able to run the streaming engine at a much higher frame rate than when using x264. This was due to the challenge of downloading the framebuffer to system memory for use with x264.

Using the streaming engine we designed, we were able to run the Bagadus virtual camera at 30 fps, the frame rate the program was designed for. Through testing, we confirmed that using NVENC resulted in almost no additional overhead for encoding, while x264 used twice as much CPU resources. By using less CPU resources, we were able to run more instances of the program on a single computer than when we used x264 for encoding. We also confirmed the visual quality from chapter 3 tests where we found that NVENC is comparable to x264 running with the superfast preset.

When we tested Quake 3, we discovered that we were able to push the latency from rendering a frame until it was displayed on a client down to 60 ms. We managed this by running the service at 720p resolution at a high frame rate of 120 frames per second. With a frame rate of 240 fps, we were able to decrease this to 50ms. By comparing the latency we measured to studies in human latency perception, we found that our system lies between the detectable and the recognizable. This fits well with our subjective experience of using the system.

When we tested Quake 3 we also discovered two new challenges: dropping bit rate, and dropping visual quality. We made no attempt to solve them, but discussed a plausible solution for one of them.

7.4 Future work

We see several parts of the streaming engine which could benefit from future studies. Below we have included a list of them with a short description.

- The streaming engine does not support audio. Including audio presents several new challenges, including synchronizing it with the video.
- Changes directly to the browser to better facilitate low latency streaming, mitigating the need for the hacks we had to perform.
- Look into ways of better controlling the bit rate post encoding, as discussed in section 6.5.
- Creating a stand-alone client and compare the latency possible to achieve when better controlling the client to the latency in the web browser.
- Finding a better, non-intrusive, way of measuring the latency in the web browser.
- Find ways of automatically enable programs for streaming through the engine without having to compile it into the source application. This could be done by using LD_PRELOAD (man 8 ld.so) and hooking the buffer swapping call.
- Research ways of sharing resources, e.g., textures, shaders etc., so that several clients can be connected to the same server, while still having their own unique "instance". This could enable much better scalability.

Appendix A

WebSocket

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Figure A.1: WebSocket frame. https://tools.ietf.org/html/rfc6455#section-5

Appendix B

Quake 3

```
int main() {
  // ...
  Sys_PlatformInit( );

  // ...

  Com_Init( commandLine );
  NET_Init( );

  CON_Init( );

  while( 1 ) { // game loop
    IN_Frame( );
    Com_Frame( );
  }

  return 0;
}
```

Listing B.1: Original main function for ioQuake3

```
struct Q3 : public RenderClient {
  void init() { // init, run after a GL context is available
    // ...
    Sys_PlatformInit( );

    // ...

    Com_Init( commandLine );
    NET_Init( );

    CON_Init( );
  }

  void mouse(int x, int y) {
    // ... save mouse input
  }

  void key(unsigned char k, int x, int y) {
    // ... save key input
  }

  void render() { // game loop function
    // ... send input to game event queue

    IN_Frame( );
    Com_Frame( );
  }
};
int main() {
  auto client = std::make_shared<Q3>();
  // 1080p@120Hz running at port 8088
  setupGLS(1920, 1080, 120, 8088, client);
}
```

Listing B.2: Modified main function for ioQuake3

Bibliography

[1] Trevor Mogg. *PC sales fall for sixth consecutive quarter as the shift to tablets and phones continues.* Oct. 2013. URL: `http://www.digitaltrends.com/computing/pc-sales-continue-to-fall/` (visited on 03/19/2014).

[2] Larry Walsh. *PC and Tablet Channel Sales Surge in 2013.* Dec. 2013. URL: `http://channelnomics.com/2013/12/26/pc-tablet-channel-sales-surge-2013/` (visited on 03/19/2014).

[3] Ovide Sherr. *Computer Sales in Free Fall.* Apr. 2013. URL: `http://online.wsj.com/news/articles/SB10001424127887324695104578414973888155516` (visited on 03/19/2014).

[4] LucasArts BioWare. *Knights of the old republic.* 2003.

[5] Stainless Games. *Carmageddon.* 1997.

[6] Pål Halvorsen et al. "Bagadus: an integrated system for arena sports analytics: a soccer case study". In: *Proceedings of the 4th ACM Multimedia Systems Conference.* ACM. 2013, pp. 48–59.

[7] Ragnar Langseth. *Master's thesis: Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views.* 2014.

[8] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. "Be your own cameraman: real-time support for zooming and panning into stored and live panoramic video". In: *Proceedings of the 5th ACM Multimedia Systems Conference.* ACM. 2014, pp. 168–171.

[9] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R Wood, and Andy Hopper. "Virtual network computing". In: *Internet Computing, IEEE* 2.1 (1998), pp. 33–38.

[10] Konstantin V Kaplinsky. "VNC tight encoder-data compression for VNC". In: *Modern Techniques and Technology, 2001. MTT 2001. Proceedings of the 7th International Scientific and Practical Conference of Students, Post-graduates and Young Scientists.* IEEE. 2001, pp. 155–157.

[11] NVIDIA Corporation. *NVENC - NVIDIA Kepler hardware video encoder.* Application Note. Sept. 2012.

[12] id Software. *Quake III Arena.* 1999.

[13] *HTML5 video - Browser support.* Mar. 2014. URL: `http://en.wikipedia.org/wiki/HTML5_video#Browser_support` (visited on 03/19/2014).

[14] Erik Zachte. *Wikimedia Traffic Analysis Report - Browsers e.a.* Jan. 2014. URL: `http://stats.wikimedia.org/archive/squid_reports/2014-01/SquidReportClients.htm` (visited on 03/19/2014).

[15] Ruan Paul. *Google Reveals Plan To Remove H.264 Support From Chrome.* Jan. 2011. URL: `http://www.wired.com/business/2011/01/google-reveals-plan-to-remove-h-264-support-from-chrome/` (visited on 03/19/2014).

[16] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. "Computing as a discipline". In: *Communications of the ACM* 32.1 (1989), pp. 9–23.

[17] M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz. *Performance and Application of the NVIDIA NVENC H. 264 Encoder.* URL: `http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4188_real-time_panorama_video_NVENC.pdf` (visited on 07/26/2014).

[18] M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, and P. Halvorsen. "Using a Commodity Hardware Video Encoder for Interactive Video Streaming". In: *Submitted to the 2014 IEEE International Symposium on Multimedia - ISM.* 2014.

[19] Edwin O Blodgett. *Teletypewriter.* US Patent 2,684,745. July 1954.

[20] Microsoft. *Remote Desktop Protocol (Windows).* URL: `http://msdn.microsoft.com/en-us/library/aa383015.aspx` (visited on 05/21/2014).

[21] Robert W Scheifler and Jim Gettys. "The X window system". In: *ACM Transactions on Graphics (TOG)* 5.2 (1986), pp. 79–109.

[22] *TigerVNC Feature Request Tracker 55 X264 encoding support.* Apr. 2013. URL: `http://sourceforge.net/p/tigervnc/feature-request-tracker/55/` (visited on 05/20/2014).

[23] Andrew Wessels, Mike Purvis, Jahrain Jackson, and S Rahman. "Remote data visualization through websockets". In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on.* IEEE. 2011, pp. 1050–1051.

[24] Eric Shepherd. *Uint8Array - Web API Interfaces | MDN.* Oct. 2010. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Uint8Array\$history?limit=all` (visited on 05/21/2014).

[25] Davy De Winter, Pieter Simoens, Lien Deboosere, Filip De Turck, Joris Moreau, Bart Dhoedt, and Piet Demeester. "A hybrid thin-client protocol for multimedia streaming and interactive gaming applications". In: *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video.* ACM. 2006, p. 15.

[26] Pablo Gomez Perez, Wolfgang Beer, and Bernhard Dorninger. "Remote rendering of industrial HMI applications". In: *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on.* IEEE. 2013, pp. 276–281.

[27] *OpenCV.* URL: `http://opencv.org/`.

[28] Ole Mogstad. "A Networked Service for the Remote Execution of Interactive Multimedia Applications". MA thesis. Norwegian University of Science and Technology, July 2008.

[29] *OnLive.* URL: `http://www.onlive.com/` (visited on 03/19/2014).

[30] *Gaika.com.* URL: `https://www.gaikai.com/` (visited on 03/19/2014).

[31] Rakesh Ranjan, Kelvin Leung, Talhah Asharaf, and Xiang Liu. "OnLive cloud gaming service". In: (2011).

[32] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. "GamingAnywhere: An open cloud gaming system". In: *Proceedings of the 4th ACM multimedia systems conference.* ACM. 2013, pp. 36–47.

[33] MainConcept. *MainConcept and AMD Collaborate to Accelerate High-Definition Video Encode.* Apr. 2010. URL: `http://www.mainconcept.com/eu/press/single-view/article/mainconcept-and-amd-collaborate-to-accelerate-high-definition-video-encode.html` (visited on 05/22/2014).

[34] Jason Garrett-Glaser. "OpenCL Acceleration of x264". AFDS 2012. Sept. 2012. URL: `https://www.youtube.com/watch?v=u0OOTqqI18A` (visited on 05/22/2014).

[35] Jason Garrett-Glaser. *OpenCL lookahead.* Feb. 2013. URL: `https://github.com/DarkShikari/x264-devel/commit/3a5f6c0`.

[36] ITU. *H.264.2 : Reference software for ITU-T H.264 advanced video coding.* 2012. URL: `http://www.itu.int/rec/T-REC-H.264.2`.

[37] Intel Corp. *Intel Quick Sync Video Technology on Intel Iris Graphics and Intel HD Graphics family - Flexible Transcode Performance and Quality.* 2013.

[38] Ganesh T S Ryan Smith. Feb. 2014. URL: `http://www.anandtech.com/show/7764/the-nvidia-geforce-gtx-750-ti-and-gtx-750-review-maxwell` (visited on 05/22/2014).

[39] NVIDIA. *Whitepaper, NVIDIA GeForce GTX 750 Ti.* Tech. rep. 2014.

[40] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *Image Processing, IEEE Transactions on* 13.4 (2004), pp. 600–612.

[41] Martin Čadík, Robert Herzog, Rafał Mantiuk, Karol Myszkowski, and Hans-Peter Seidel. "New measurements reveal weaknesses of image quality metrics in evaluating graphics artifacts". In: *ACM Transactions on Graphics (TOG)* 31.6 (2012), p. 147.

[42] Alain Hore and Djemel Ziou. "Image quality metrics: PSNR vs. SSIM". In: *Pattern Recognition (ICPR), 2010 20th International Conference on.* IEEE. 2010, pp. 2366–2369.

[43] *Nettfart.no - Test kapasiteten paa nettoppkoblingen din.* 2010. (Visited on 05/22/2014).

[44] Akamai. *AKAMAI'S STATE OF THE INTERNET Q4 2013.* Tech. rep. 2014.

[45] *Xiph.org Video Test Media [derf's collection].* URL: `https://media.xiph.org/video/derf/` (visited on 07/30/2014).

[46] I Goncalves, Silvia Pfeiffer, and Christopher Montgomery. *Ogg Media Types.* Tech. rep. RFC 5334 (Proposed Standard), 2008.

[47] YouTube. *Advanced encoding settings.* URL: `https://support.google.com/youtube/answer/1722171?hl=en` (visited on 05/27/2014).

[48] Sergey Beduev Emanuele Oriani. *qpsnr - A quick PSNR/SSIM analyzer for Linux.* URL: `https://github.com/bsv9/qpsnr`.

[49] Microsoft. *Windows GDI.* URL: `http://msdn.microsoft.com/en-us/library/windows/desktop/dd145203(v=vs.85).aspx` (visited on 07/17/2014).

[50] Glyn Matthews Dean Michael Berris. *The C++ Network Library Project.* 2012.

[51] *x264 common/common.c.* URL: `https://github.com/DarkShikari/x264-devel/blob/master/common/common.c` (visited on 05/27/2014).

[52] Microsoft Corporation. *Converting Between YUV and RGB.* URL: `http://msdn.microsoft.com/en-us/library/ms893078.aspx` (visited on 05/27/2014).

[53] Loren Merritt. *Convert x264 to use NV12 pixel format internally.* July 2010. URL: `https://github.com/DarkShikari/x264-devel/commit/c9b2fcb`.

[54] Khronos OpenGL Working Group Inc. *The OpenGL Graphics System: A Specification.* Version 4.4 (Core Profile). Mar. 19, 2014.

[55] Silicon Graphics. *The OpenGL Graphics System: A Specification.* Version 1.0. July 1, 1994.

[56] *Khronos Contributing Members.* URL: `https://www.khronos.org/members/contributors` (visited on 05/27/2014).

[57] Paul Malin. *Spout.* URL: `https://www.shadertoy.com/view/lsXGzH`.

[58] Inigo Quilez. *raymarching distance fields.* URL: `http://www.iquilezles.org/www/articles/raymarchingdf/raymarchingdf.htm` (visited on 05/27/2014).

[59] Simon Green. "The OpenGL framebuffer object extension". In: *Game Developers Conference.* Vol. 2005. 2005.

[60] Peter Dimov. *The Boost Bind Library.* 2001.

[61] Ian Fette and Alexey Melnikov. "The websocket protocol". In: (2011).

[62] *mozilla-central mozilla/layout/style/html.css.* URL: `https://mxr.mozilla.org/mozilla-central/source/layout/style/html.css` (visited on 05/27/2014).

[63] *html.css i trunk/Source/WebCore/css - WebKit.* URL: `view-source:http://trac.webkit.org/browser/trunk/Source/WebCore/css/html.css` (visited on 05/27/2014).

[64] Asgeir Mortensen, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. "Automatic event extraction and video summaries from soccer games". In: *Proceedings of the 5th ACM Multimedia Systems Conference.* ACM. 2014, pp. 176–179.

[65] Vamsidhar Reddy Gaddam, Ragnar Langseth, Sigurd Ljødal, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, and Pål Halvorsen. "Interactive zoom and panning from live panoramic video". In: *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop.* ACM. 2014, p. 19.

[66] Vamsidhar Reddy Gaddam, Carsten Griwodz, and Pål Halvorsen. "Automatic exposure for panoramic systems in uncontrolled lighting conditions: a football stadium case study". In: *IS&T/SPIE Electronic Imaging.* International Society for Optics and Photonics. 2014, pp. 90120C–90120C.

[67] id Software. *Quake III Arena Source Code.* URL: `https://github.com/id-Software/Quake-III-Arena` (visited on 05/20/2014).

[68] *ioquake3.* URL: `http://www.ioquake3.org` (visited on 05/20/2014).

[69] W3C. *Pointer Lock.* Dec. 2013. URL: `http://www.w3.org/TR/pointerlock/` (visited on 05/20/2014).

[70] *Blix Solutions AS.* URL: `http://www.blix.com`.

[71] Ricardo Jota, Albert Ng, Paul Dietz, and Daniel Wigdor. "How fast is fast enough?: a study of the effects of latency in direct-touch pointing tasks". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM. 2013, pp. 2291–2300.

[72] Jake Cannell. *EnterTheSingularity: Latency & Human Response Time in current and future games.* Mar. 2010. URL: `http://enterthesingularity.blogspot.no/2010/04/latency-human-response-time-in-current.html` (visited on 03/03/2014).

[73] Michael Jarschel, Daniel Schlosser, Sven Scheuring, and Tobias Hoßfeld. "Gaming in the clouds: QoE and the users' perspective". In: *Mathematical and Computer Modelling* 57.11 (2013), pp. 2883–2894.