BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

ATTRACTIVE EFFECTS FOR VIDEO PROCESSING

LÍBIVÉ EFEKTY PRO ZPRACOVÁNÍ VIDEA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MARTIN IVANČO**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. ADAM HEROUT, PhD.**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/17100/2017/xivanc03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Ivančo Martin**
Obor: Informační technologie
Téma: **Líbivé efekty pro zpracování videa**
Attractive Effects for Video Processing
Kategorie: Zpracování obrazu

Pokyny:

1. Seznamte se s problematikou zpracování obrazu a aplikování populárních vizuálních efektů na obraz a video.
2. Seznamte se s problematikou zpracování videa v reálném čase, zaměřte se na streamující kamery a prohlížení ve webových aplikacích.
3. Navrhněte a prototypujte vhodné množství líbivých filtrů; vypracujte k jednotlivým filtrům jednoduché a uživatelům přístupné uživatelské rozhraní pro jejich nastavení.
4. Vybrané vhodné filtry zoptimalizujte, aby umožňovaly zpracování videa v reálném čase.
5. Navrhněte a implementujte program, který bude v reálném čase zpracovávat video ze streamující kamery a výstup bude streamovat dále.
6. Vytvořte webové rozhraní pro přehrávání a nastavování modifikovaného videa.
7. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Gary Bradski, Adrian Kaehler: Learning OpenCV; Computer Vision with the OpenCV Library, O'Reilly Media, 2008
- Richard Szeliski: Computer Vision: Algorithms and Applications, Springer, 2011
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN-13: 978-0321965516
- Susan M. Weinschenk: 100 věcí, které by měl každý designér vědět o lidech, Computer Press, Brno 2012

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Herout Adam, prof. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
L.S. 612 66 Brno, Božetěchova 2

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

The goal of this work was to create a server solution capable of applying visually pleasing effects to live video streams. The processed video shall then be displayed in a web user interface allowing the user to adjust the applied effects.

This goal is achieved by creating a video processing application connected to a web server using a RPC framework. The video generated by this application is streamed using HLS protocol enabling it to be displayed in the web user interface. The user interface is connected to the web server via WebSocket protocol. This enables the web server to connect the user interface with the video processing application.

The created solution is accessible via a web user interface, which allows the user to submit video stream URL, which then gets displayed. The time required to load the stream is usually around 10 seconds, which could be much shorter if the solution ran on a more powerful machine. The user interface also enables the user to adjust the settings in a separate mode. In this mode, only a single frame from the stream is displayed, but the adjustments made by the user are displayed almost instantly.

Abstrakt

Cieľom tejto práce bolo vytvoriť serverové riešenie schopné aplikovať vizuálne atraktívne efekty na živé video streamy. Spracované video by sa malo zobrazovať vo webovom užívateľskom rozhraní, ktoré by malo taktiež umožniť užívateľovi aplikované efekty zmeniť.

Tento cieľ bol dosiahnutý vytvorením aplikácie na spracovanie videa, ktorá je pripojená k webovému serveru prostredníctvom RPC frameworku. Video vygenerované touto aplikáciou sa streamuje pomocou HLS protokolu, čo umožňuje jeho zobrazenie vo webovom užívateľskom rozhraní. Užívateľské rozhranie je pripojené k webovému serveru cez WebSocket protokol. To umožňuje webovému serveru spojiť užívateľské rozhranie s aplikáciou na spracovanie videa.

Vytvorené riešenie je prístupné cez webové užívateľské rozhranie, ktoré užívateľovi umožňuje zadať URL adresu video streamu, ktorý sa následne zobrazí. Čas potrebný na načítanie streamu je zvyčajne okolo 10 sekúnd, čo by však mohlo trvať o veľa kratšie ak by riešenie bolo spustené na silnejšom stroji. Užívateľské rozhranie tiež umožňuje užívateľovi zmeniť nastavenia v oddelenom móde. V tomto móde sa zobrazuje iba jedna snímka zo streamu, no úpravy ktoré užívateľ spraví sa zobrazia skoro okamžite.

Keywords

Real-time video processing, Video stream enhancement, Colorful filter application, Attractive video effects

Kľúčové slová

Spracovanie videa v reálnom čase, Vylepšenie video streamov, Aplikácia farebných filtrov, Atraktívne video efekty

Reference

IVANČO, Martin. *Attractive effects for video processing.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, PhD.

Rozšírený abstrakt

Cieľom tejto práce je vytvoriť riešenie problému vizuálne neatraktívneho videa streamovaného množstvom IP kamier z celého sveta, vďaka ktorému môžu potenciálny návštevníci skontrolovať aktuálne počasie a atraktivitu daného miesta. Jadrom problému je schopnosť upraviť video v reálnom čase a následne ho streamovať ďalej aby mohlo byť doručené klientovi. Video stream môže meškať niekoľko sekúnd oproti reálnemu času, keďže aplikácia cieli na živé prenosy výhľadov na mestá a krajinu. Na druhej strane, riešenie musí byť schopné poskytnúť užívateľvi okamžitú spätnú väzbu pri kofigurácii efektov aplikovaných na video stream. Ideálne by riešenie malo byť dostupné cez webové užívateľské rozhranie schopné zobraziť spracovaný video stream ako aj poskytnúť užívateľovi možnosť nakonfigurovať nastavenia použité na spracovanie video streamu, aby užívateľ nemusel inštalovať žiaden ďalší softvér.

Navrhované riešenie pozostáva z troch hlavných častí – *server na spracovanie videa*, *webový server* a *webové užívateľské rozhranie*, pričom každá z nich vykonáva rôzne úlohy. *Server na spracovanie videa* vstupný video stream nie len spracúva ale aj opätovne streamuje. Na spracovanie videa používa niektoré metódy odvodené od spracovania obrazu, ktoré sú čo najviac optimalizované pre spracovávanie veľkého množstva obrazov – snímkov videa v reálnom čase. Spracované video následne segmentuje a vytvára HLS stream ktorého doručenie už však má na starosti *webový server*.

Ten je centrom komunikácie celého riešenia – zabezpečuje predávanie informácií medzi webovým užívateľským rozhraním a serverom na spracovanie videa a tiež doručuje samotné webové užívateľské rozhranie ako aj spracovaný video stream užívateľovi. Doručovania streamu je natívne schopný vďaka protokolu HLS ktorý využíva bežné HTTP requesty na postupné doručovanie segmentov video streamu. Na komunikáciu s webovým rozhraním využíva protokol WebSocket ktorý umožňuje obojsmernú komunikáciu medzi webovým serverom a užívateľským rozhraním bez nutnosti vykonávať HTTP requesty. Komunikáciu so serverom na spracovanie videa udržuje pomocou RPC protokolu ktorý umožňuje priamo volať metódy serveru na spracovanie videa z webového serveru a tým pádom kedykoľvek meniť nastavenia aplikovaných efektov podľa požiadaviek užívateľa.

Poslednou súčasťou je *webové užívateľské rozhranie* ktoré všetku funkcionalitu serveru na spracovanie videa sprístupňuje užívateľovi vo webovom prehliadači. Navrhnuté riešenie bolo následne potrebné implementovať. Implementácia bola značne náročná, keďže bolo potrebné použiť viaceré knižnice, ktorých vzájomná kompatibilita bola minimálna. K náročnosti prispela aj skutočnosť, že dokumentácia k niektorým z týchto knižníc bola neúplná alebo neexistujúca. Nakoniec bol server na spracovanie videa vytvorený pomocou knižnice **OpenCV**, ktorá bola použitá pri spracovaní videa a nástroja **FFmpeg**, ktorý zabezpečuje vytvárnie HLS streamu. Pre implementáciu komunikácie medzi webovým serverom a serverom na spracovanie videa sa použil framework **gRPC**, zatiaľ čo komunikácia medzi webovým serverom a webovým užívateľským rozhraním funguje pomocou knižnice **Socket.IO**. Webové užívateľské rozhranie bolo implementované bežnými webovými technológiami s pomocou knižnice **Bootstrap** na vizuálne prvky rozhrania a knižnice **hls.js** na načítanie HLS streamu.

Výsledná aplikácia je schopná načítať video stream zadaný užívateľom prostredníctvom webového užívateľského rozhrania, aplikovať na neho rozličné efekty a spracovaný video stream zobraziť v užívateľskom rozhraní. Takisto poskytuje možnosť aplikované efekty zmeniť a nastaviť. V súčasnej dobe je výsledná aplikácia schopná obsluhovať len jednoho klienta zároveň, avšak toto obmedzenie je možné v budúcnosti odstrániť. Načítanie video streamu v užívateľskom rozhraní trvá približne 10 sekúnd, záležiac od výkonnosti počítača

na ktorom server beží. Pri nastavovaní efektov sa zmeny prejavia ihneď vďaka aplikácii efektov iba na statickú snímku. Užívateľ tiež môže zdieľať spracovaný video stream pomocou permanentného linku. Aj keď aplikácia v súčasnom stave nie je pripravená na použitie v profesionálnej sfére, pri zachovaní určitých podmienok je plne funkčná a je preto validným riešením danej úlohy.

Attractive effects for video processing

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Adam Herout. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Martin Ivančo
May 16, 2018

Acknowledgements

I would like to thank my supervisor prof. Ing. Adam Herout, PhD. for his help.

Contents

Chapter 1

Introduction

This work addresses the problem of visually unpleasant images streamed by many IP cameras around the world used to show potential visitors the current weather conditions as well as the available view at a given location. The core of the problem is the ability to adjust the video stream in real time, and then stream it forth so that it can be shown to a client. The video stream can be a few seconds delayed when compared to real world view, as the targeted application scenarios are dealing with landscapes and cityscapes. On the other hand, the solution must be able to give the user instant feedback in the process of configuring the effects applied on the stream. Ideally, the solution should be accessible via a web user interface capable of displaying the processed stream as well as giving the user an option to configure the settings used to process the stream, so that there is no need for the user to install additional software.

While there are solutions to parts of the problem, none of them addresses the problem in its entirety. If the only requirement would be to receive the stream, process and display it just locally on the running machine, image processing libraries, like those mentioned in Section 5.1, would be sufficient to create a solution quite easily. If, on the other hand, processing of the stream was minimal and the solution would require just to forward the video stream with slight changes, tools like FFmpeg would allow to do this in one command. The proposed problem, however, gets fairly complex when all the requirements are considered.

This paper first describes the theory behind the video processing methods as well as the communication protocols needed to create a complete solution to this problem, which is then proposed in a Chapter 4. This solution is them implemented using several libraries and frameworks. This process is described in Chapter 5. While the solution proposed in this paper might not be as simple and as clean as I would like, it is, to my best knowledge, currently the only solution to this problem as stated.

Chapter 2

Video Processing Methods

This chapter introduces the reader to the theory of video processing methods used in my application. It first explains the absolute basics of video processing – the essential terms used in the rest of this chapter as well as the similarities and differences when compared to image processing. It then focuses on the *tone curve*, a technique used to add or subtract contrast from the video, as well as alter the brightness of only a part of the video. This chapter then goes on to explain how the look up tables work and why they are so very useful, especially for video processing. In the last two sections of this chapter, two more methods for video processing are explained, which are quite different from the ones explained before them. The first one is *unsharp mask* which is used to make the video look more sharp. The second method is called *vignetting* and it adds an artistic effect to the video by darkening it near the edges.

Although this chapter does not mention any specific tools or libraries used to implement the previously mentioned methods (this is discussed in detail in Section 5.1), it tries to explain them in a practical fashion so that it is easily understood.

2.1 Principles of Video Processing

Video processing is at its core very similar to static image processing. The video consists of a sequence of images called *frames* that are quickly shown one after another. Each frame is made up of square points, called *pixels* which bear a certain color represented by RGB value. The color of the pixels is altered according to various image processing methods, which results in a processed frame which is different from the unprocessed one. The processed frames can then be played as a video again. This process is shown in Figure 2.1.

As processing of the frames gets more complicated, the time needed to process each frame gets longer and longer. This is where video processing differs from image processing as not all image processing methods are designed to be so quick that they can be applied in a fraction of a second. This is a problem especially in real-time video processing. Frames are usually presented at a 25 or 30 frames per second rate, which means the processing can not take longer than 4 hundredths of a second. This would be impossible using many advanced and even some basic image processing techniques, and that is why video processing methods need to be as optimized as possible, sometimes even to the point where the quality is compromised. Even with various efficiency tweaks, though, real-time video processing requires quite capable computer to run smoothly.

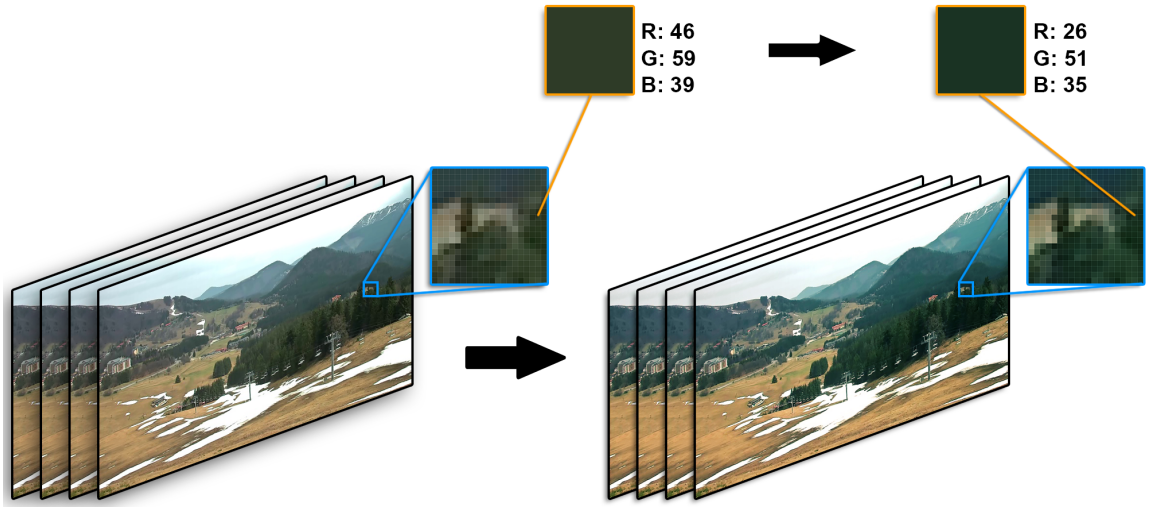

Figure 2.1: The left part of the picture shows frames of unprocessed video, which get processed by altering individual pixel values according to various settings. The processed video frames in the right part of the picture are then played as a processed video.

2.2 Tone Curve

One of the most important image processing methods that is also widely used for video processing is the *tone curve*. It is a curve that represents a mapping function for the brightness values of pixels. Figure 2.2 shows an example of a tone curve. The tone curve usually has 4 control points that determine its shape, but there can be more or less control points depending on the type of use and desired effect. In the case of 4 control points, they are usually named *shadows*, *darks*, *lights* and *highlights*, referring to the visual brightness level of pixels the control point mainly affects. Every tone curve also has two stable points at $[0, 0]$ and $[255, 255]$[1]. This ensures that for each input value, there is an output value defined, and also that the full spectrum of possible brightness is used for the output.

Using these points, the whole tone curve can then be calculated using **Catmull-Rom Spline** [9]. Other interpolation techniques can be used too, but Catmull-Rom spline fits this purpose very well. For an input value x, the output y is calculated using Equation (2.1) [4]. Parameter t controls the the distance of the resulting point $\mathbf{C}$ on the curve from points $\mathbf{P}_1$ and $\mathbf{P}_2$. Plugging in $t = t_1$ results in $\mathbf{C} = \mathbf{P}_1$ while plugging in $t = t_2$ results in $\mathbf{C} = \mathbf{P}_2$. This means that in order to get points on the curve segment we need to interpolate parameter t between t_1 and t_2. Parameters t_0, t_1, t_2 and t_3 can be calculated using Equation (2.3). Unfortunately, this makes it impossible to simply calculate an output value for a given input value (in other words, get y for x). I solved this problem by dividing the curve segment into several parts equal in length and performing the calculations on their boundaries to get a reasonable amount of reference points. With enough reference points, the missing output values can be calculated using simple linear interpolation between two of the reference points whose x coordinates match the input value the closest. Figure 2.3 might explain this solution better.

[1] As with RGB values, brightness values are usually 8-bit too.

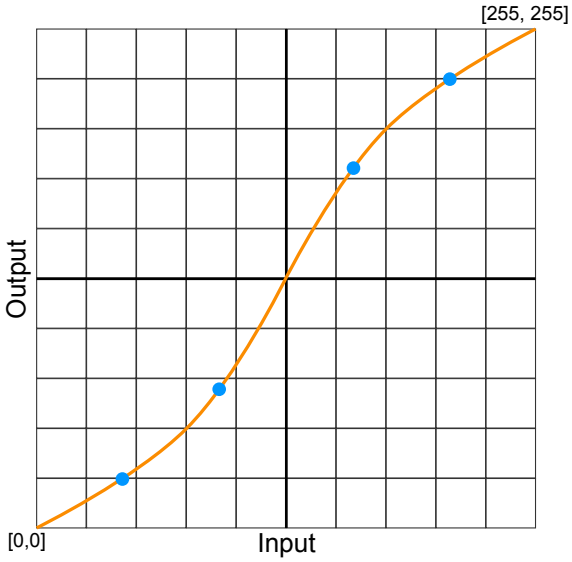

Figure 2.2: Example of a tone curve used to enhance contrast in midtones while reducing it towards the shadows and highlights. This type of tone curve is usually referred to as an S-Curve and it is appropriate to use on most images.

Let $\mathbf{P}_i = [x_i \quad y_i]^T$ denote a point. Curve segment $\mathbf{C}$ can then be produced by:

$$\mathbf{C} = \frac{t_2 - t}{t_2 - t_1}\mathbf{B}_1 + \frac{t - t_1}{t_2 - t_1}\mathbf{B}_2 \tag{2.1}$$

where

$$\begin{aligned}
\mathbf{B}_1 &= \frac{t_2 - t}{t_2 - t_0}\mathbf{A}_1 + \frac{t - t_0}{t_2 - t_0}\mathbf{A}_2 \\
\mathbf{B}_2 &= \frac{t_3 - t}{t_3 - t_1}\mathbf{A}_2 + \frac{t - t_1}{t_3 - t_1}\mathbf{A}_3 \\
\mathbf{A}_1 &= \frac{t_1 - t}{t_1 - t_0}\mathbf{P}_0 + \frac{t - t_0}{t_1 - t_0}\mathbf{P}_1 \\
\mathbf{A}_2 &= \frac{t_2 - t}{t_2 - t_1}\mathbf{P}_1 + \frac{t - t_1}{t_2 - t_1}\mathbf{P}_2 \\
\mathbf{A}_3 &= \frac{t_3 - t}{t_3 - t_2}\mathbf{P}_2 + \frac{t - t_2}{t_3 - t_2}\mathbf{P}_3
\end{aligned} \tag{2.2}$$

and

$$t_{i+1} = \left[\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}\right]^{\alpha} + t_i \tag{2.3}$$

where $t_0 = 0$ and α ranges from 0 to 1 with an effect on the curvature of the resulting spline. For $\alpha = 0$, the resulting spline is a *Uniform* Catmull-Rom spline, while if $\alpha = 1$, the resulting spline is called *Chordal* Catmull-Rom spline. When $\alpha = 0.5$, the result will be the most well-known *Centripetal* Catmull-Rom Spline with a medium curvature, which I also chose to work with, because it interpolates the control points smoothly while making the curves small enough to cope with longer distances between them too. Figure 2.4 shows a comparison between these types of Catmull-Rom spline.

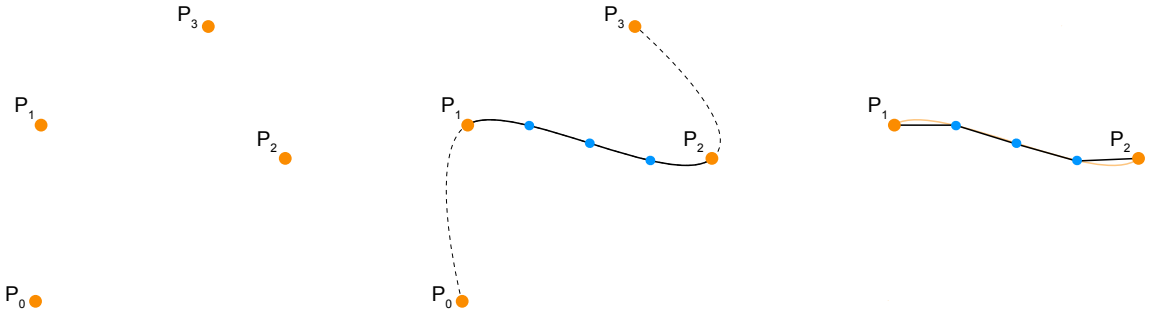

Figure 2.3: Visualization of the process of Catmull-Rom spline interpolation combined with linear interpolation. Blue points are reference points calculated on the curve using Catmull-Rom spline equations [4]. Those are then used for linear interpolation to get an approximation of the curve.

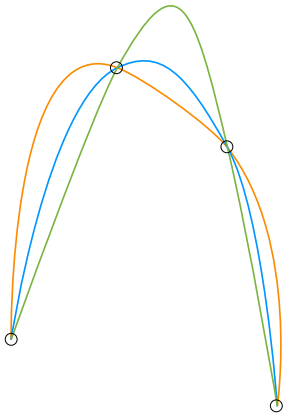

Figure 2.4: Comparison of chordal (highlighted orange), centripetal (highlighted blue) and uniform (highlighted green) Catmull-Rom spline.

2.3 Look Up Tables

Calculations, such as the ones used to calculate tone curve, are not trivial and can really slow video processing down, making it impossible to do it in real time. These calculations, however, are quite repetitive, especially in videos, where the successive frames are similar. The same input pixels are recalculated over and over, which is really inefficient and could be avoided. Look up tables are the solution to this problem. Look up tables are tables containing output values for all or a range of input values calculated in advance. They are designed to speed up the process of getting the output, as reading the corresponding data from memory is usually much faster than calculating the result from scratch, especially if the calculation should be as frequent and repetitive as in video processing. Table 2.1 shows an example of what could be a part of a look up table.

In image and video processing, look up tables are used to transform the overall look of the image by altering colors. These look up tables usually only contain *one column* with output color, as the input color is represented by the *index* of the row. They also do not normally contain an output for every possible input color, instead, they only contain an output for every fourth, eighth or sixteenth possible color. This is done to save memory, as these look up tables are usually exported as text files, and they could reach hundreds of

Table 2.1: Example of a part of look up table

Input	Output
⋮	⋮
2	0
3	1
4	3
5	6
6	10
⋮	⋮

Table 2.2: Difference between a 3D look up table and three 1D look up tables.

3D LUT			R	G	B
⋮			⋮	⋮	⋮
0.953186	0.000000	0.000000	0.132477	0.130524	0.131836
0.954346	0.000000	0.000000	0.158478	0.158630	0.158966
0.956400	0.000000	0.000000	0.187531	0.190857	0.189911
0.000000	0.016144	0.000000	0.219940	0.227753	0.225159
0.016022	0.016144	0.000000	0.256165	0.269226	0.264435
0.032715	0.016144	0.000000	0.296295	0.314880	0.307312
⋮			⋮	⋮	⋮

megabytes in size. Input colors that do not have their output explicitly defined by the look up table are then calculated by linearly interpolating the two closest matching input colors defined by the look up table.

In professional image and video processing color look up tables are usually used in a **3D** format. This means that the look up table contains output RGB value combination for each input RGB value combination, and not output for each input R value, G value and B value separately. Table 2.2 shows the difference more clearly. The use of 3D look up tables versus separate 1D tables for each color channel would only really make a difference if there was a need to change a single color to a completely different one, without touching the other colors. For processing live streamed video, though, the use of 3D look up tables would be superfluous.

However, 3D look up tables are widely used and the possibility to simply export such a look up table from a professional image editing program like *Adobe Photoshop* and then import it to my application to use it to change the look of the live streamed video would be great. Therefore I decided to convert these 3D look up tables to a 1D version which is used to *color grade* (change the look of the color) the video.

2.4 Unsharp Mask

Apart from altering the look of the colors on the video, there are other adjustments that can be made to make it look better, such as image sharpening. One of the most popular

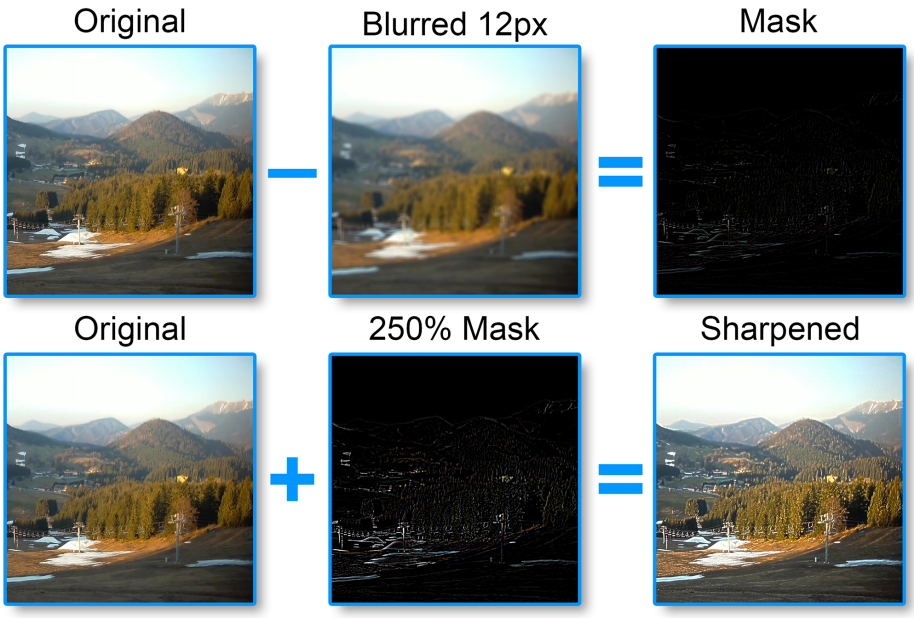

Figure 2.5: An example of sharpening an image using unsharp masking with unusually high parameters, so that the effects are more visible. The original image is blurred with a radius of 10 pixels and then the acquired mask is multiplied by $2,5$ before it is added to the original image.

methods to sharpen an image (or a video in this case) is using a process called *unsharp masking*.

The first step is to make a copy of the original image and apply *Gaussian blur* [11] to it. The chosen *radius* is one of two key parameters of unsharp masking. Radius affects how precise the sharpening will be. The smaller the radius, the more detailed the sharpening will be, but it can also introduce noise to the image. The blurred image is then subtracted from the original image to create a mask. This mask is then multiplied by an *amount* parameter. Amount is the second key parameter of unsharp masking and it controls how strong the sharpening effects will be. The multiplied mask is then added to the original image, which results in a visually sharper image. Figure 2.5 shows how the entire process works.

Another parameter that can be accounted for when using unsharp masking is *threshold*. Threshold controls the minimal brightness difference that should be sharpened. Before the mask is multiplied by the amount parameter, all pixels that are darker than the set threshold are made totally black, so that they are not present in the sharpened image. This can help eliminate noise in areas of the image that should be smooth.

Unsharp masking can also be used to locally enhance contrast [23]. The radius needs to be set really high around 50 pixels, while the amount is usually quite low around 10%. This can enhance contrast around edges while leaving the rest of the image nearly untouched which can be useful if big parts of the image are too dark or too light and raising contrast would cause them to go completely black or white, ergo image would lose detail.

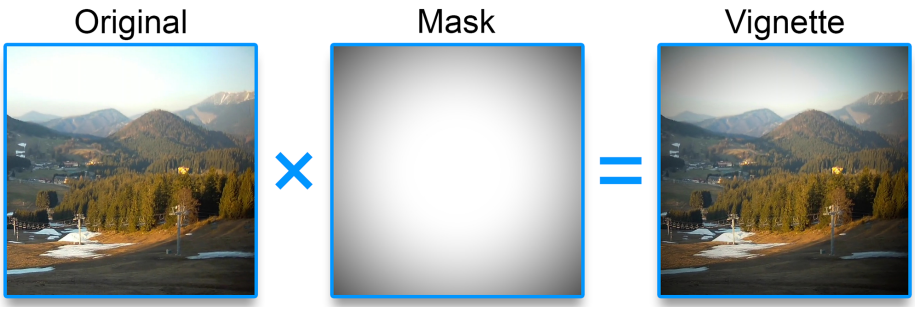

Figure 2.6: Application of vignette with midpoint set to 0.

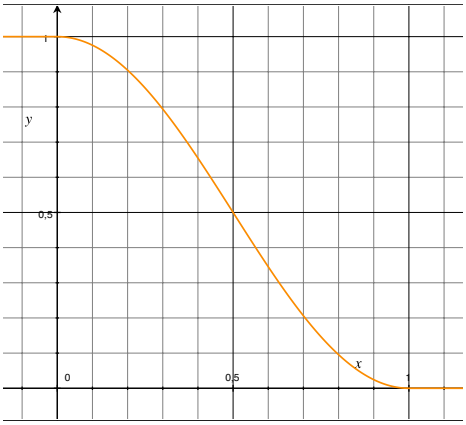

Figure 2.7: Plotted function from Equation (2.4)

2.5 Vignetting

Vignetting is a process which reduces the brightness of an image near its edges. It is sometimes used for artistic purposes, mainly to draw attention to the center of the image and increase the dramatic effect of the image. Figure 2.6 shows an example of vignette application.

First, a mask is calculated using a given *midpoint* parameter. Midpoint is the distance between the center of the image and the point where the vignette will start. The gradient between the inner white circle and the outer black circle of a vignette is usually calculated using a cosine function. Example of such a function is:

$$y = \begin{cases} 1 & \text{if } x \leq 0 \\ 0 & \text{if } x \geq 1 \\ \cos^2(\frac{\pi}{2} \cdot x) & \text{otherwise} \end{cases} \tag{2.4}$$

The gradient is usually as wide as the distance between the center of the image and any of its corners, although this can also be considered the second parameter of the vignette. After the mask is calculated, the brightness of each pixel of the original image is multiplied by the value of the mask at the corresponding position.

Values of the mask can range from 0 (represented as black) to 1 (represented as white). Multiplying the original image by the mask will therefore only darken pixels near the edges of the image while keeping the ones near the center intact.

Chapter 3

Streaming and Communication

This chapter describes the challenges of real-time video streaming and cross-platform communication. It first introduces the available protocols for streaming the processed video and explains why some of them are not suitable for streaming into the web user interface while others are a better option. It then shifts its attention to the protocols used to create communication channels between different parts of the solution. RPC protocols used for creating communication channels between programs are described first. At the end of this chapter, the WebSocket protocol essential for creating web user interfaces is introduced, and it is explained how it connects the web page with the web server.

Similarly to Chapter 2, this chapter tries to explain the protocols in a practical, easy to understand way, without mentioning any specific libraries. These are thoroughly described in Section 5.2.

3.1 Video Streaming Protocols

Streaming media and especially video via network has always been a daunting task. The data needs to be delivered on time, so that the video can be played without much waiting on the client side. Designing a protocol that is reliable enough to deliver the video without losing much data, while being so fast that the video can be played without lagging is a big challenge.

In principle, all video streaming protocols utilize the idea that the video must be sent in *small chunks* to avoid having to download the whole file, which would be very time consuming. After the first few chunks are loaded, the video can start playing, while the successive chunks need to be loaded in the background to be ready for playback when needed. The method of implementation is of course dependent on the specific protocol, but the idea stays the same [22] across all of them. One major difference, though, that divides the protocols into two groups is the choice of the *transport method*.

The first group of protocols uses fast but unreliable datagram protocols, mainly **UDP**. It is very fast and efficient, but the client side needs to be capable of recovering from the data losses [25] caused by the unreliable delivery. The second group uses reliable transport protocols, mainly **TCP**. The reliable transport protocols ensure that the data gets delivered correctly, which means the client side can be much simpler. However, TCP is much less efficient than UDP and therefore the video streaming protocols using TCP might require faster network connection.

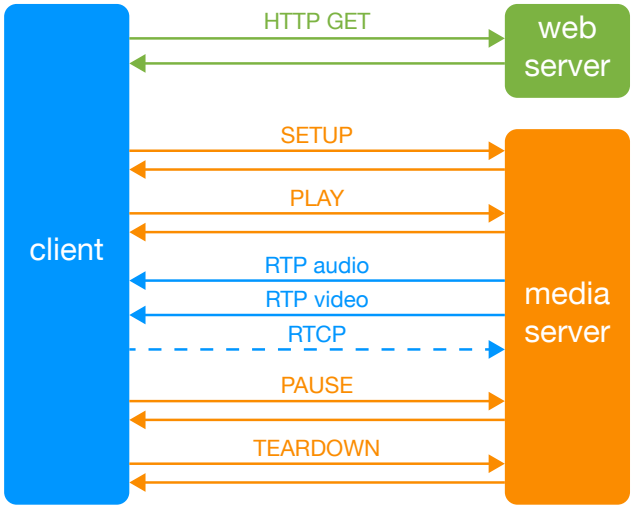

Figure 3.1: Diagram showing an example of streaming a video using RTSP [8]. The client acquires address of the media server from the web server, connects to it and requests a stream. After some time the client closes the connection.

.

3.1.1 Real Time Streaming Protocol

Real Time Streaming Protocol (RTSP) [26] is a protocol designed to control real-time streaming. It is usually used in conjunction with the Real-time Transport Protocol (RTP) and Real-time Control Protocol (RTCP) to enable media streaming via IP networks. RTSP itself does not transfer any media, it only transports meta-data used to control the transmission of RTP packets from the server to the client. RTSP defines a set of commands that can be sent to the server to `SETUP`, `PLAY` or `PAUSE` the stream. This set is quite extensive and covers all the necessary stages from negotiating the options of the streaming, like encoding, bitrate, etc. through controlling the playback while streaming to closing the connection altogether.

The data itself, however is transferred only using RTP. RTCP [15] is used for getting feedback about the quality of the streaming, as well as synchronizing multiple media streams (for example, video and audio tracks are streamed separately and therefore they need to be synchronized using RTCP). RTP defines a header with information about the format, sequence number as well as some other information about the chunk of data transferred in the payload. The specific methods for packetization of the media can differ from format to format. Figure 3.1 shows how RTSP works in conjunction with RTP and RTCP.

RTSP streaming is widely used in situations where low *latency*[1] is important, such as video conferences. However, RTSP streaming is much more complicated to implement than some other streaming protocols and requires a standalone media server capable of serving the stream. This poses a problem especially if the streamed video is created in real time and needs to be transferred to the media server via some sort of special communication channel. The lack of freely available libraries and tools with a reasonably documented API is another downside of RTSP streaming. Thankfully, there are other options for real-time video streaming.

[1]The time difference between real time and the displayed stream on the client side.

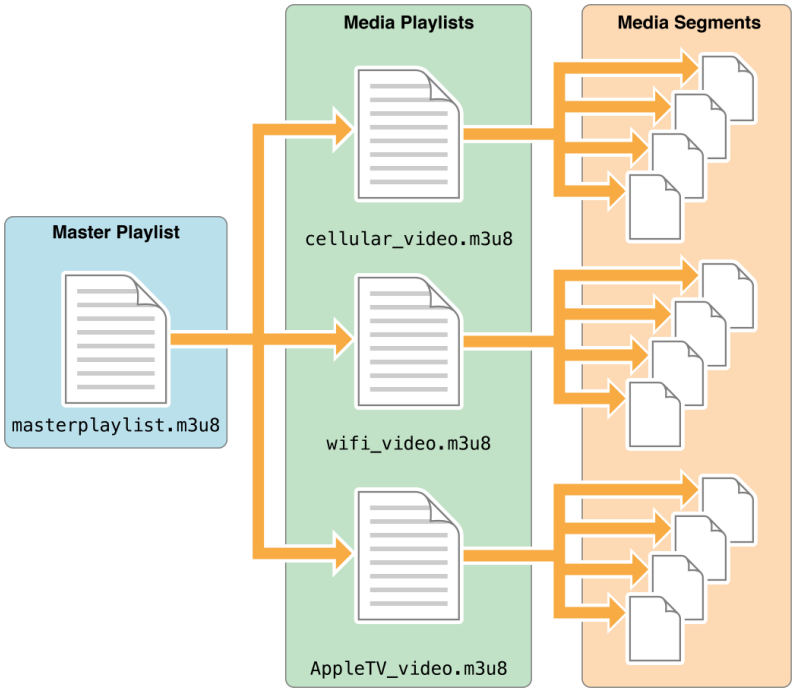

Figure 3.2: Diagram of HLS playlists [1]. The `masterplaylist.m3u8` contains references to multiple media playlists for various use cases, which then reference corresponding media segments.

3.1.2 HTTP Live Streaming

HTTP Live Streaming (HLS) [20] is a protocol developed by Apple as a natively supported way of streaming media on its devices [3]. As the name suggests, it operates over HTTP, which means that the packets are delivered via a reliable TCP connection. This means that the client only needs to support basic HTTP GET requests to be capable of using this protocol. Thanks to its simplicity, this protocol has become quite popular even on non-Apple devices.

HLS is built on playlists referencing other playlists or media segments. These playlists are in the M3U8 format which is a Unicode version of M3U type playlists. Every HLS stream needs to have a master playlist that contains references to available media playlists. The master playlist also contains information about the media playlists. Some of them are compulsory, such as `BITRATE` and `CODEC`, while others are optional. The media playlists contain references to segments of the video or audio in the corresponding format, which can then be requested by the client. Figure 3.2 shows an example of HLS playlists structure. If the video is live streamed, the media playlist gets continually updated and must be reloaded every once in a while by the client.

The disadvantage of HLS lies in latency. The media segments are usually at least one second long, which is an unacceptable delay for video conferences or IP telephony. For these cases, the use of a protocol operating over UDP is inevitable. However, if a few second delay is acceptable, the simplicity of HLS makes it a great choice for video streaming. Although it is not natively supported by non-Apple devices, the support is easily added using existing libraries that are also simple to use.

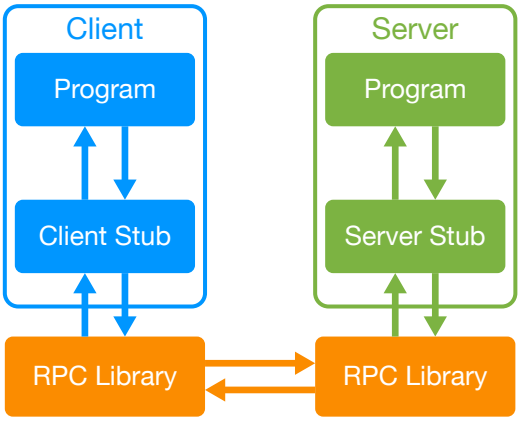

Figure 3.3: Diagram of a common RPC architecture. The *Client* can call methods of the *Server* using the *Client Stub* capable of communicating with the *Server Stub* using the runtime RPC library.

3.1.3 Other Video Streaming Protocols

Apart from RTSP and HLS, there are quite a few other protocols used for streaming media content. Most of them, however, are proprietary and they are fully supported only on their native platforms. A notable example is **Real-Time Messaging Protocol** (RTMP) [21] used by Adobe Flash Player to establish and maintain connection to a media server capable of streaming media using this protocol. This protocol was very popular in the past as Flash Player was the only real option to display live streamed video on the web. With the advent of HTML5 Video, however, Flash Player has become obsolete and therefore RTMP is not a good choice for a modern streaming protocol.

WebRTC [18] is an open-source protocol/technology actively supported and developed by Google, Mozilla and others, which provides a simple API for Real-Time Communication between browsers and mobile applications. This protocol has a great potential to become one of the most widely used streaming protocols on the web. At the time of writing this paper, however, WebRTCs API, although simple, only supports streaming video from web camera and simple data transfer, which is insufficient to achieve the goal of this work.

3.2 Remote Procedure Call Protocols

Remote Procedure Call (RPC) [6] protocols provide a way to call methods of a standalone program from a different standalone program. RPC protocols abstract all the complicated communication between the programs to a simple object method call. Each program using a RPC protocol plays a role of the server – making some of its methods available for others to call; or a role of the client – making use of the servers methods. In theory, a program can play both of these roles at the same time, although this is not very common. Figure 3.3 shows a basic structure of two programs connected by a RPC protocol.

Each client program contains a *client stub*, an object which represents the server and provides methods with the same definitions as the ones the actual server implements. Calling any of the methods will trigger the *runtime library* that implements the RPC protocol, which will then ensure that the call gets transported to the server. The runtime library on the server will process this call and deliver it to the *server stub* which is a part of the server

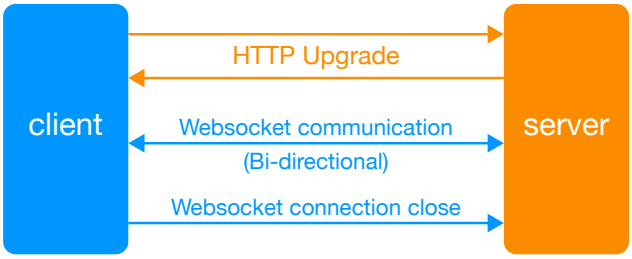

Figure 3.4: Basic diagram showing the process of using WebSocket Protocol from the initiation to the closing.

itself and can therefore call any of its methods. The returned value is passed back to the runtime library by the server stub and the whole process of transporting the method call is now reversed to deliver the returned value back to the client to complete the cycle.

3.3 WebSocket Protocol

WebSocket Protocol [12] is a communication protocol that enables bi-directional communication between a client and a web server. Thanks to WebSockets, the client (usually an internet browser) can communicate with the server without sending HTTP requests. The server can also send messages to the client at any given time, so the client does not need to continuously poll the server in case it is waiting for some specific condition to be met.

Figure 3.4 shows a standard example of communication using WebSocket Protocol from start to finish. The client first sends a **HTTP Upgrade** to the server, asking if it is capable of WebSocket communication. It also sends a *Key* which serves as a verification that the server received the message from the client. If the sever supports WebSocket communication, it then alters the *Key* according to the specification of WebSocket Protocol [12] to get the *Accept* code which it then sends back to the client in a response with the HTTP code `101`. Any other code means that the communication has failed to be initiated. After a correct response from the server, WebSocket communication is established and the server as well as the client can now send messages back and forth. After one of the sides decides to end the communication, it simply needs to send a predefined closing message, to which the other side responds with another closing message to make sure that no more data needs to be sent. After that the communication is considered as closed.

Chapter 4

Proposed Solution

This chapter contains a detailed description of the proposed solution. It aims to explain the individual parts of the solution, their function and purpose. It also explains how the parts are connected to form a functional unit. This chapter does not mention any specific libraries or tools needed to implement the individual parts of the solution described in this chapter. This is discussed in Chapter 5.

The solution consists of three main parts. First of them is the *image processing server*, which handles the reception, processing and streaming of live stream video. The second part is the *web server* responsible for handling the communication as well as delivery of the content to the user. The last part is the *web user interface* that allows the user to control the solution in an easy manner.

4.1 Image Processing Server

The image processing server is the core of the solution that processes input live streamed video and streams the output so that it can be delivered to the user. It also needs to be able to communicate with the other parts of the solution, so that the settings used for processing the video can be changed in the user interface. It should be quite optimized as the video needs to be processed in real time which is a computationally difficult task. That is also the reason why this part of the solution should be a remote server and not run locally on the clients machine.

In my solution, the server has two parts, each of them responsible for handling the image processing for one of the two modes of the resulting application. Figure 4.1 shows their similar parts as well as their differences.

The first part processes the live streamed video according to the arguments passed to it at its start. Apart from the arguments that enable to set all the preferences for the processing, this part is not capable of any communication with the web server whatsoever. It is used along with the *playing mode* of the application that simply displays the processed video stream. To enable this, the processed video is segmented and streamed using HLS protocol. Although this introduces a notable delay between the input video stream and the video displayed in the user interface, it is acceptable as the settings can not be edited in this mode.

The second part uses the same image processing base as the first, but differs in its communication abilities. This part enables for the settings passed as arguments to be changed via a RPC protocol. It can therefore be used along with the *editing mode* that

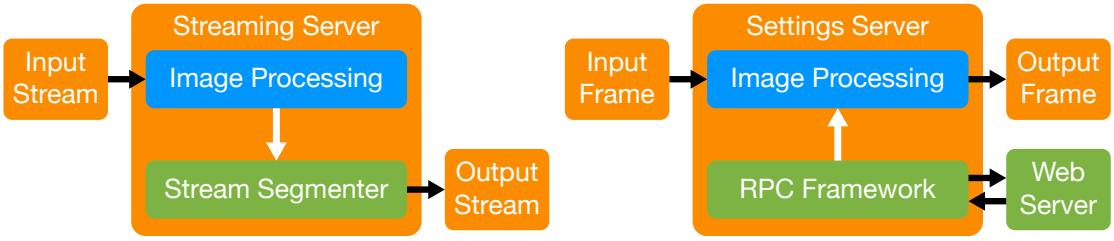

Figure 4.1: Diagrams of the two parts of the image processing server. The *Streaming Server* is used for processing the input stream with parameters set in advance, while the *Settings Server* is used for tuning the settings on a single frame of the input video stream.

enables user to change the settings used to process the input video stream. This part of the image processing server, however, does not stream any video. At the start, it retrieves a single frame from the input video stream which it then processes and exports. The processed frame is then displayed in the web user interface. Anytime the user changes the settings, the frame is processed again. This ensures that the changes get displayed almost instantly so that the user can fluently tweak them. Although displaying only a single frame is not optimal, the delay between the user changing the setting and the change actually appearing in a streamed video would be too long. Using a different streaming protocol, such as RTSP could solve this problem, however, the choice of HLS as a streaming protocol has already been explained in Section 3.1.

4.2 Web Server

The web server is an essential part of the solution as it not only delivers all needed media to the user, but also serves as a communication midpoint between the user interface and the image processing server. It handles running the image server as well as interaction with the web user interface and binds the entire solution together. Figure 4.2 shows a diagram of the entire solution and the role of the web server in it.

After a client connects to the web server, the web server delivers a user interface in which the user can submit a URL of an input video stream. When the user submits the URL, the web server starts the image server - the part used for playing mode, and passes it the URL. Then it delivers the playing mode user interface to the client. When the user decides to change the settings used to process the video stream, the web server stops the running part of image server and starts the other one used for adjusting the settings. It passes all the settings that have been used so far to it, and then delivers the editing mode user interface.

The server is connected to the user interface using the WebSocket protocol. This enables it to react to any changes in the settings or other events the user interface sends messages about. Anytime it receives a message about a change in the settings, it needs to pass the change to the image processing server. It does this via a RPC protocol.

Apart from this the web server also delivers requested media to the client via HTTP. Thanks to the nature of HLS protocol, there is no need for a special media streaming server, and the processed video stream is also delivered this way.

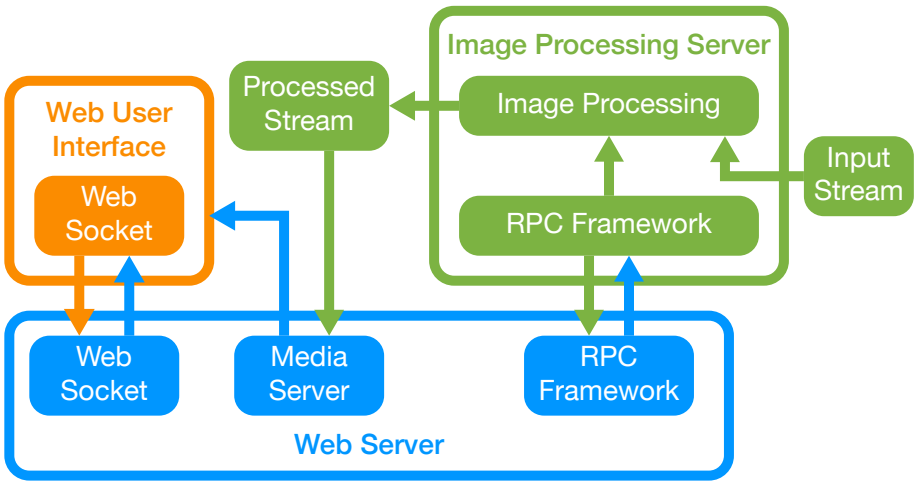

Figure 4.2: Diagram of the proposed solution. It consists of three main parts – *Image Processing Server*, *Web Server* and *Web User Interface*.

4.3 Web User Interface

Since the beginning of this work, I wanted the final solution to be accessible via a web user interface, because an ordinary user does not want to download and install an entire program just to apply some visual effects to a live video stream and share it with a friend. This led me to think about how should the web interface be designed, so that it is as simple and intuitive as possible, as well as how it should be implemented. Details on the implementation part can be found in Sections 5.2.2 and 5.3.

The first idea was based on a one mode concept – there would be a simple welcome screen for the user to submit the URL of the video stream, and then there would be a single screen with the processed video playing on one side, while the settings would be on the other one. After testing the latency of the settings being applied to the video stream, however, there was a need for a two mode user interface – one for playing the processed stream and one for editing the processing settings.

For the two mode interface, I designed a mockup that can be found on Figure 4.3. As mentioned before, I wanted to keep the interface as clean and simple as possible, to avoid confusing the user. That is why the playing mode only contains the playing processed video stream and two buttons – one for opening the editing mode and one for sharing the the processed stream via a permanent link. The editing mode is fairly simple too and it is based on the one mode concept. There is the frame with currently applied settings on the left, while the settings are neatly organized in a box on the right. Most of the settings are edited using sliders, while the filters are organized as radio buttons with image thumbnails, previewing the effect the filter will have on the image. There is one more button located under the processed frame, used to return back to the playing mode with the current settings applied to the video stream.

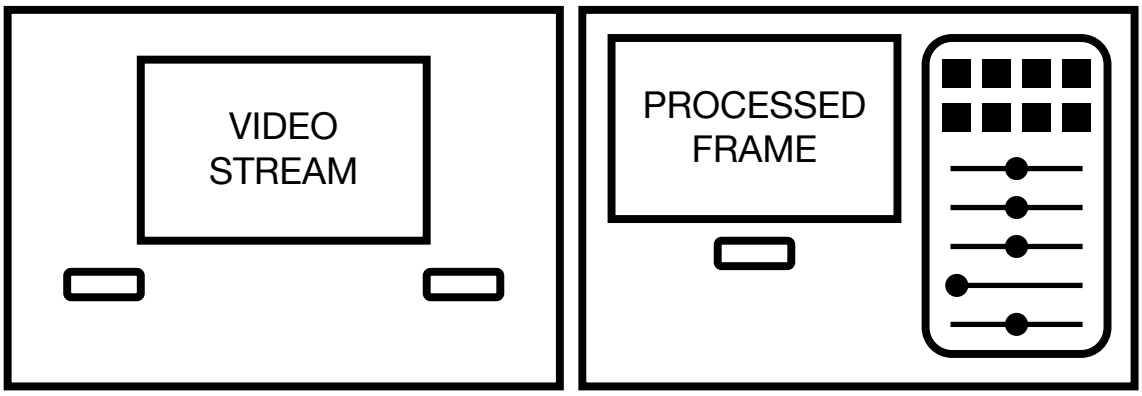

Figure 4.3: Mockup of the two mode user interface. The playing mode used to display the edited video stream is on the left, while the editing mode used to adjust the settings used to process the video stream is on the right.

Chapter 5

Implementation and Results

This chapter discusses available libraries, tools and frameworks considered or chosen for the implementation of the solution. It states the required functionality of the libraries needed for each part of the solution and then focuses on the available libraries individually, discussing how well they satisfy the requirements and how well they cooperate with the other libraries and parts of the solution. It also describes the process of integration of these libraries into the solution. It starts with the libraries used for the video processing part of the solution, then moves on to the ones used for implementing the communication inside the solution and finally it brings up the implementation of the web user interface. In the last section, the current state of the solution is examined in detail.

5.1 Utilized Libraries for Video Processing

For the video processing part of the solution, the main purpose of the libraries should be to enable reception, processing and streaming of video. This turned out to be quite a challenging task as there is not a single library or tool available that is capable of doing all of that. On top of that, the existing libraries do not easily cooperate with the other ones which makes it hard to put together a working unit composed of multiple libraries. There is, however, quite an extensive amount of different libraries and tools with various functions and abilities, so with enough of research, it is possible to combine some of them to create a working solution.

In my solution, I finally decided to use OpenCV for the video processing as well as receiving the streamed video, while FFmpeg handles the streaming of the processed video. This combination turned out to be the only one that worked the way I wanted. Although combining them introduced some non-ideal concepts to the solution (described in detail in Section 5.1.3), it did not cause any major problems in running the final application.

5.1.1 OpenCV

OpenCV [7] is one of the best known actively developed open-source image processing libraries in the world, which is written in C++ with multi-language and multi-platform support. It implements basic image processing operations as well as numerous advanced functions from the field of computer vision and machine learning, all highly optimized and capable of running on GPU if available. Apart from academics, OpenCV is used in many of the biggest companies in the world.

OpenCV plays a major part in the solution as it handles all of the video processing on the input video stream. Some of the adjustments available in the solution were already implemented as a method inside OpenCV, like converting the image to black and white, while others, the more complex ones, needed to be implemented. With the help of existing OpenCV methods, however, the implementation was quite easy except for the ones that were complicated from principle, such as the tone curve which is thoroughly explained in Section 2.2.

OpenCV also handles the reception of the input stream. Thanks to its `VideoCapture` class, OpenCV is able to get frames from any video stream, which can then be processed and displayed locally. However, OpenCV is not capable of streaming the processed frames, which is a feature necessary to display the processed stream in a remote browser user interface. The use of another library capable of putting the processed frames together and streaming them as a video was inevitable.

5.1.2 LibVLC

The search for a library capable of streaming the processed video led me to LibVLC [19], another open-source library, developed mainly by VideoLan as a base for the famous VLC player, which is simply a wrapper on LibVLC with a graphical user interface. Like OpenCV, LibVLC is a massive library with loads of features and functions. Apart from HLS, LibVLC is also capable of streaming using RTSP, which would be a very useful function as it would allow the processed video to be played in the editing mode too, thanks to the minimal latency of RTP transport.

Although most of LibVLCs functionality is available via a well documented, robust API, some of the less commonly used functionality is much less accessible. One such feature is custom input from memory. After researching numerous ways that should theoretically enable raw input of frames I could not get one of them to work in my conditions. Although I was almost certain that there was a way to accomplish this, I decided to look for an alternative as the documentation for this part of LibVLC was poor and the source code too complicated to simply examine.

5.1.3 FFmpeg

FFmpeg [5] is an open-source, cross platform framework for playing, transcoding and streaming media, with a very wide support for almost every media format ever created. It is built on a set of libraries that can be linked to any application to make use of FFmpegs abilities, however, FFmpeg is mainly used as a command line tool by end users. FFmpeg is capable of streaming media in various formats, including HLS and RTSP. For RTSP, however, setting up a `ffserver` is essential, which is a deprecated [13] part of FFmpeg. For HLS, however, FFmpeg provides full support in its command line tool `ffmpeg` including deleting old segments no longer needed for the live stream.

As I wanted to use the processed video frames in memory as a custom input to FFmpeg, I naturally wanted to make use of its set of libraries to accomplish this task. Although FFmpegs command line tool is very well documented, documentation for its libraries at the time of writing this document is virtually non-existent, except for a few unfinished unofficial articles. As the libraries of FFmpeg could not be used, I was looking for another alternative but with no success. Thanks to my supervisor, I found out about a feature of the `ffmpeg` command line tool that can receive raw frames as an input via a standard binary `pipe`.

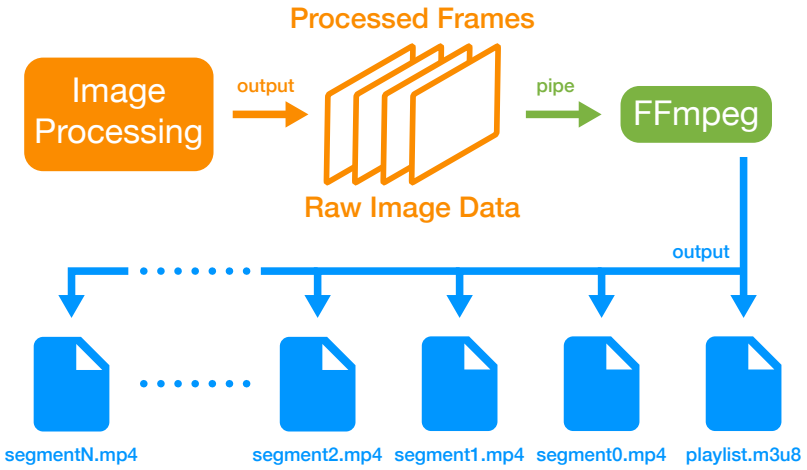

Figure 5.1: Diagram showing the usage of FFmpeg in the solution. Processed video stream in the form of raw frames is piped to the input of FFmpeg which puts the frames together to form a video, segments it, encodes it and writes it to disk. Simultaneously it manages the playlist file which it continuously updates with new references to new segments and removes the old references while also deleting the actual files on disk.

This solution, although not optimal[1], was the only one I found out to be working in my conditions. As it did not cause any problems while testing, I decided to stick with it. The command used in the final solution looks like this:

```
ffmpeg -f rawvideo -pixel_format bgr24 -video_size 640x360 -framerate 25 \
-i - -f hls -c:v libx264 -pix_fmt yuv420p -profile:v high -level 4.0 \
-flags +cgop -g 50 -hls_time 2 -hls_list_size 5 -hls_flags \
delete_segments ../media/output.m3u8
```

All arguments before the `-i` argument correspond to the input, while the ones behind it correspond to the output. For the input, I needed to define the pixel format, which is `bgr24` because OpenCV natively stores pixel color values in blue, green, red order. The video size and framerate is set according to the current settings of output from OpenCV. Input is set to `-` which stands for standard input – the piped data come here.

The output settings are a bit more complicated. The output format is set to `hls` to segment it according to HLS protocol. The video *codec* is set to `h264` so that the stream is widely supported. `yuv420p` pixel format is standard with `h264` and the *profile* set to `high` and the *level* set to `4.0` provide a good balance between wide support and good quality [2]. These settings also determine the `CODECS` field in the master playlist of the HLS stream. The corresponding value is `avc1.640028` in which `avc1` denotes the use of H264 [16] encoding and the `640028` is the `profile-level-id` which is a 24-bit code written in a hexadecimal form. It can be divided into 3 8-bit codes – `profile_idc`, `profile_iop` and `level_idc`. These codes denote the *profile*, array of compliant *constraint flags* and the *level* of the H264 video, respectively. In this specific case, the `64` represents the `high` *profile*, the `00` means that no other constraints are satisfied and the `28` represents the *level* `4.0`.

[1] Piping large amounts of data via a standard `pipe` can introduce a fair amount of lag to the program

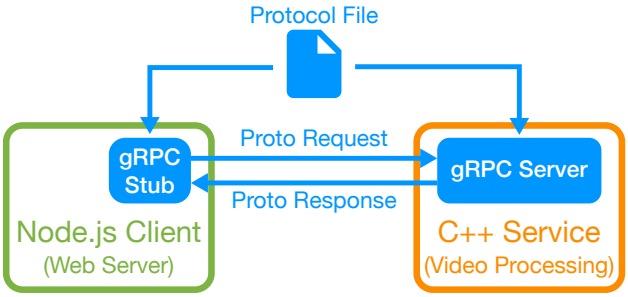

Figure 5.2: Diagram of the use of gRPC in the solution. gRPC uses protocol buffers to serialize the communication. The protocol file represents a contract between the client and server side about the methods that the client can call on the server. This file is then used as an input for creating the gRPC Server and Stub.

The subsequent arguments define that the segments should be 2 seconds long and that an I frame[2] should appear every 50 frames – at a framerate of 25 frames per second this means that it will appear at the beginning of each segment. The last two arguments before the path to the output media playlist referencing the stream video segments define that the playlist should have a maximum length of 5 and that the old segments should be deleted. Figure 5.1 shows how FFmpeg is used in the solution to stream the processed frames.

5.2 Implementation of Communication

With the video processed and streamed, I needed to implement the web user interface to enable the user to adjust the settings and view the processed video stream. Before that, however, I needed to figure out how to establish some kind of communication between the web user interface – essentially a regular webpage and the video processing server. As I already had to use a simple web server to deliver the user interface to the user as well as the HLS video stream from the video processing server, I could as well use it as the communication center of the entire solution as proposed in Section 4.2. For implementing this communication I used two frameworks discussed in the two subsequent subsections.

5.2.1 GRPC

gRPC [14] is an open-source RPC framework initially developed by Google as an internal implementation of the RPC protocol used for many of their projects. gRPCs wide multi-language support makes it an ideal solution for fast communication between applications written in different languages. gRPC makes use of another technology made by Google – Protocol Buffers. They define the way the data transferred between applications is serialized using a platform and language neutral *protocol file*. Thanks to gRPC, calling a function from another application written in a different language and getting the result is as simple as defining a protocol and then calling the function like any other one in the target program. Figure 5.2 shows the usage of gRPC in this solution, however, the diagram would look very

[2]H264 codec encodes the video in a way that only areas of the image that change from frame to frame are stored, while the ones that stay the same are only referenced. However, every now and then, there needs to be a so called I frame that is stored in its full form. The I frame marks the beginning of a GOP – group of pictures – frames that can reference any of the previous frames up until the I frame. The other frames in the GOP are called P frames.

similar in any other use case of gRPC, with the only difference being in the used languages and/or number of clients.

In this solution, the web server plays the role of client as it will call methods of the video processing server, which will play the role of the server. To enable calling the methods of the server from the client using gRPC, I needed to create a *protocol file* that defines the methods available for calling from the clients on the server, and all of the input parameters as well as the output. In this solution, I only needed to call one method to change the processing settings according to the adjustments the user made, and the protocol is therefore simple:

```
syntax = "proto3";
package grpclev;

service GrpcLev {
  rpc SetOption(Setting) returns (Result) {}
}

message Setting {
        int32 option = 1;
        string new_val = 2;
}

message Result {
        int32 code = 1;
}
```

This protocol file defines a so called `service` which is a group of functions that a server can implement and provide for remote calls. Each RPC function must be defined as a part of service. Only one function is defined in this protocol file, and that is `SetOption`. Arguments and return values of RPC functions are transferred as messages which must be defined in the protocol file too. According to this file, gRPC can create a server and a stub. The server then just needs to implement the method defined by the protocol and the client can call it from the stub object as it would call any other method.

5.2.2 Socket.IO

With the web server able to communicate with the video processing server via gRPC, the only problem left to solve to be able to create a functional web user interface is to create a communication channel between the browser and the web server. Socket.IO [24] enables just that. Socket.IO is an open-source *addon*[3] to Node.js which implements the WebSocket Protocol and therefore creates a real-time bi-directional communication channel between a browser (client) and a web server written in Node.js. Socket.IO is one of the best known, easiest to use addons for this purpose.

In my solution, I chose to implement the web server in Node.js, not only because it makes this process very easy, but also because there are plenty of useful addons, like Socket.IO, that provide great features via a simple to use API. In the web user interface, I simply needed to implement triggers when the user adjusts any of the settings and then call the `emit()` method from Socket.IO client library to send a message to the web server, which can then process it and pass it to the video processing server to change the settings accordingly.

[3]Package that can be linked and used in a program written in the Node.js framework.

5.3 Implementation of Web UI

The web UI was implemented using standard web technologies with the help of some useful libraries to simplify this process. The two modes of the solution are at its core simple HTML web pages, including the video in the playing mode which is thanks to HTML5 video implemented as a simple tag, and the use of Flash or other auxiliary technologies is unnecessary. Although HTML5 video does not directly support reception of HLS streams, it does support an extension implemented in most of the widely used browsers, called Media Source Extension [27]. This feature allows JavaScript to generate media streams and play them in the video element. This can be used to add the support for HLS, which has already been done in the hls.js library. hls.js [10] enables playing an HLS stream inside a video tag using a simple to use API.

Another JavaScript library used in the solution was the widely popular jQuery which greatly simplified implementing the editing mode. The editing mode is essentially an image preview of the current settings applied on a frame of the video stream and a HTML form used for the settings panel. Any change in the settings needs to be passed to the web server which can then pass it to the video processing server. This is implemented in a JavaScript script running behind the editing mode on the clients side. jQuery enables to easily create a trigger function that is called anytime a selected element is changed. This is also possible in pure JavaScript, jQuery just makes this much easier and also adds many other functions that simplify JavaScript programming.

The overall look of the UI is implemented using CSS, with the use of Bootstrap. Bootstrap is a popular toolkit to build clean and responsive web user interfaces using predefined CSS classes that can be added to HTML elements to speed up the process of styling. Apart from simplifying the building of the layout of the UI and styling all of its elements, Bootstrap also adds support for many useful concepts, such as Modals – dialog boxes displayed on top of the current page used to inform the user of some event, such as a successful copy of the permanent link to the clipboard in this solution as shown in Figure 5.5.

5.4 Final Application

Putting all of the work described above together resulted in an application capable of receiving a video stream given by the user via a URL on the server, processing it according to user adjusted settings, streaming the processed video and displaying it in a web user interface. Figures 5.3 – 5.5 show several screenshots from the usage of the finished application, while Figure 5.6 shows some results of the video processing server – original and processed frames generated by applying the available effects in the application.

When the user connects to the web server, a welcome screen is displayed, where the user can submit the input stream URL to be processed by the server. After submitting the URL, the playing mode is displayed, as shown in Figure 5.3.

This mode displays the video streamed by the video processing server, currently exactly the same as the original stream as the user has not adjusted any settings in the editing mode yet. The time required to load the video ranges from 5 to 15 seconds, depending on the current CPU load. This delay is suboptimal and could be improved by using a different streaming protocol, however, the reasons for choosing HLS over other protocols with lower latency has already been stated in Section 3.1. The user can now click the button on the left to get to the editing mode.

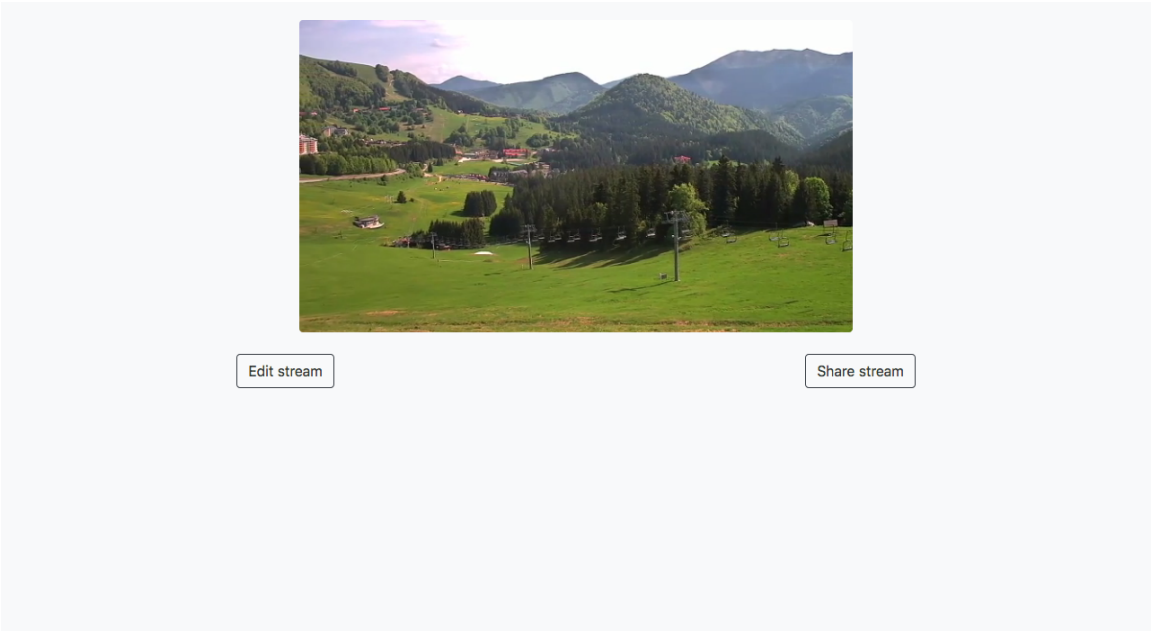

Figure 5.3: The playing mode of the finished application displaying the video stream outputted by the video processing server.

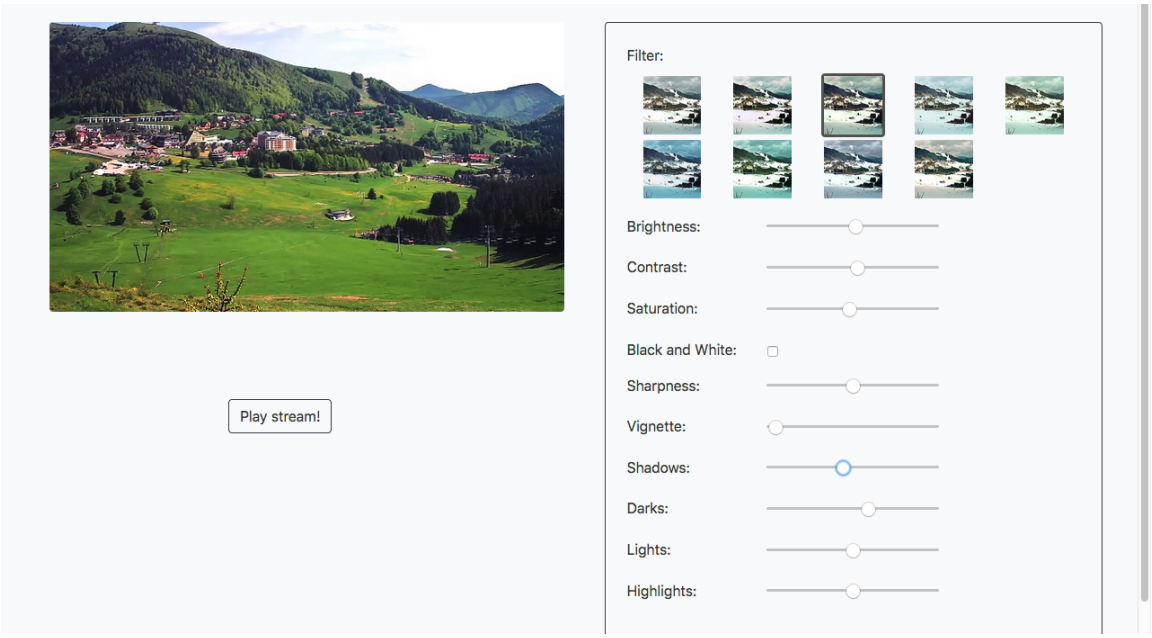

Figure 5.4: The editing mode of the finished application enables user to adjust the settings used to process the video stream with the current settings previewed on a static frame from the input video stream.

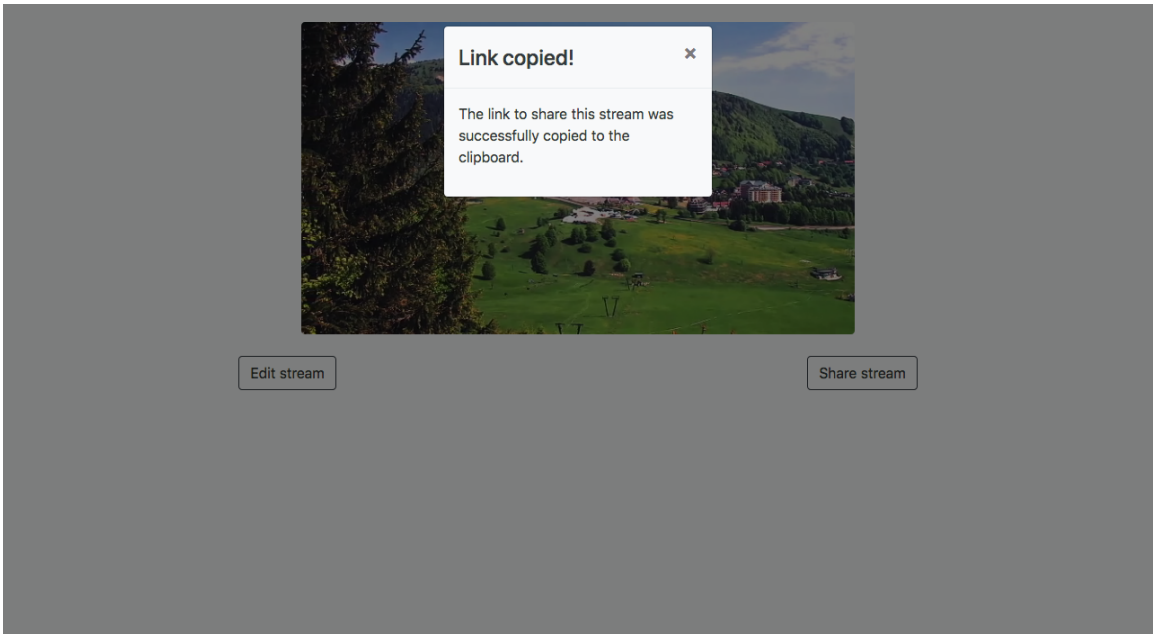

Figure 5.5: Modal window notifying the user of successfully copying the URL for sharing the adjusted stream.

The editing mode displayed in Figure 5.4 shows a preview of currently applied effects on a static frame grabbed from the input video stream, and provides a settings panel for the user to adjust the settings. As the video is not streamed in this mode and only a single image needs to be updated when the user adjusts a setting, the change is displayed almost instantly. When the user is satisfied with the settings, the button labeled "Play stream!" redirects the user back to the playing mode, with the adjusted settings now applied to the stream.

The user can now see the processed live stream and optionally return to the editing mode to tweak the settings some more. The processed stream can be shared through the playing mode using the button on the right. This button simply copies the current URL to the clipboard, as the processing settings as well as the input stream URL are already encoded in it – when the user clicks the "Play stream!" button in the editing mode, the current settings are actually submitted as a HTML form using a HTTP GET request. When the URL is successfully copied to the clipboard, user is notified by a modal as shown in Figure 5.5.

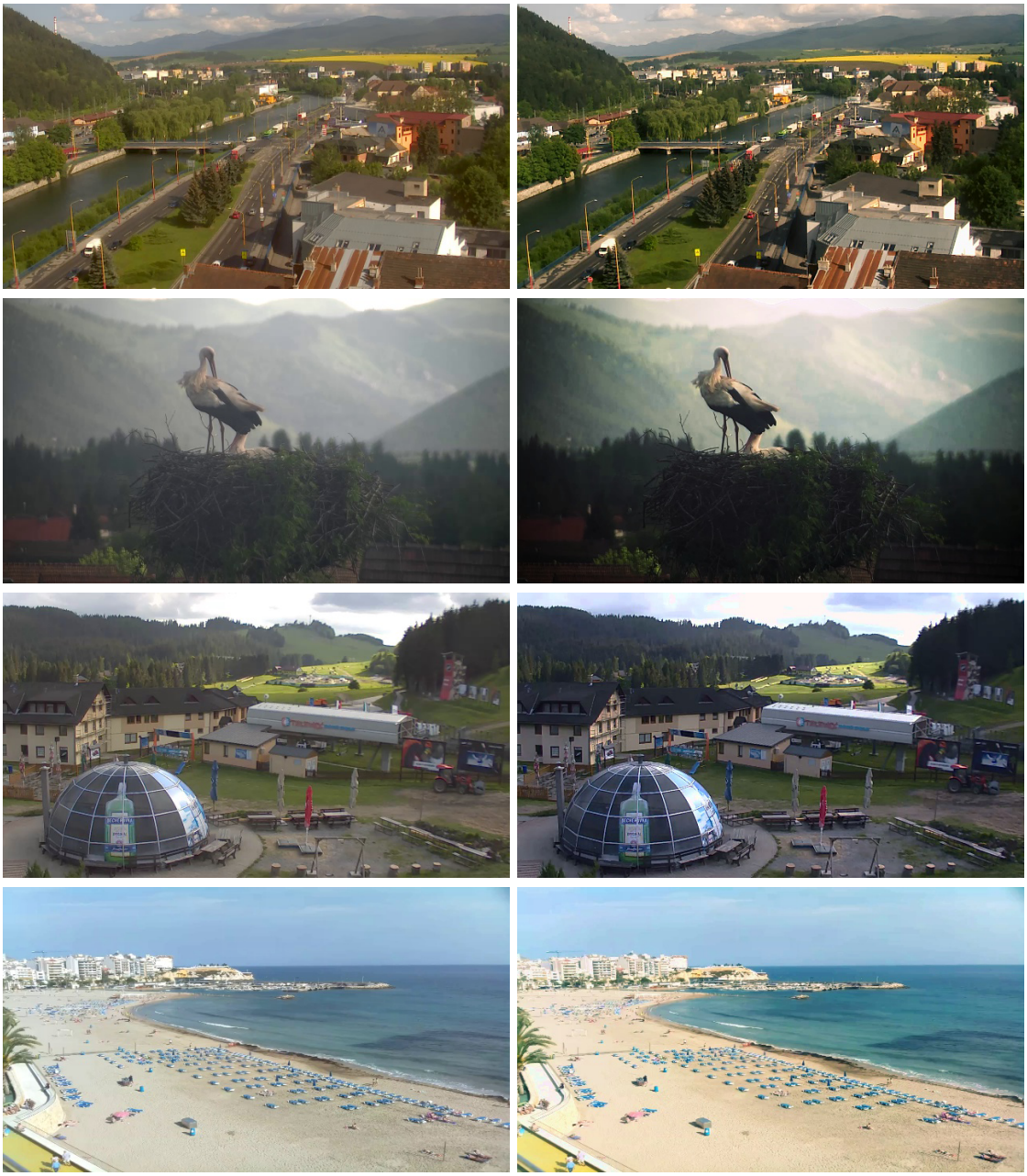

Figure 5.6: Original frames from 4 example live streams and their processed versions outputted by the video processing server.

Chapter 6

Conclusion

This work proposed a server solution that enables to process live streamed video in real time using various filters and effects, which is accessible via a web user interface allowing the user to adjust the effects applied on the video as well as display it. The proposed solution consists of three main parts – *the video processing server*, *the web server* and *the web user interface*, each responsible for a different task – processing and re-streaming the input video stream, handling communication between the other parts and delivering the stream and the web user interface to the user, and enabling the user to interact with the server, respectively. The solution utilizes various image processing methods and communication protocols that are also described in this thesis. It relies on the HLS protocol to deliver the processed stream and RPC and WebSocket protocols to maintain a communication channel between its individual parts.

The proposed solution is then implemented. For this purpose, it utilizes various libraries, frameworks and tools, the use of which is thoroughly discussed in the chapter regarding implementation. For the video processing part, it uses OpenCV, while the streaming is handled using FFmpeg and a web server written in Node.js. The web server uses gRPC to communicate with the video processing server and Socket.IO to maintain a connection to the web user interface. The web user interface is implemented using standard web technologies, with the help of some libraries, like Bootstrap and hls.js. The implementation was quite a challenging task because the compatibility between the existing libraries in video processing and streaming is very poor and the documentation is often not quite sufficient either. However, I managed to create a working application which is currently capable of serving one user at a time although this could be relatively easily scaled. If the application runs on a medium to highly powerful computer, it can process the video stream in real time without any lag, but the video takes around 10 seconds to load due to the use of HLS. As the application uses only a preview on a static frame while adjusting the settings, the feedback is almost instant which is essential for this use case.

Improvements that could be made in the future include scaling the web server to be able to handle multiple users simultaneously, optimizing the video processing to run even smoother using acceleration on the GPU, implementing adaptive resolution, so that the video processing server produces a lag-free video stream with any available CPU and implementing RTSP streaming to shorten the delay when loading the processed video stream. In the current state the solution is much more usable as a prototype for research and educational purposes, rather than a full-fledged application ready for a use in the industry. However, this was not the goal of this work, and the implemented application is fully functional under the specified conditions, which makes it a viable solution for the given task.

This work posed a big challenge for me. However, solving it has broadened my knowledge of this field and improved my understanding of many advanced concepts. Presenting this work at Excel@Fit 2018 Conference [17] has helped me clarify the structure and the content of this thesis, as well as giving me many interesting insights on improving this work in the future.

Bibliography

[1] Apple Inc.: *About HTTP Live Streaming*. October 2014. [Online; visited 27.04.2018]. Retrieved from: https://developer.apple.com/library/content/referencelibrary/GettingStarted/AboutHTTPLiveStreaming/about/about.html

[2] Apple Inc.: Using HTTP Live Streaming. March 2016. [Online; visited 06.05.2018]. Retrieved from: https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/UsingHTTPLiveStreaming/UsingHTTPLiveStreaming.html

[3] Apple Inc.: *HTTP Live Streaming*. 2018. [Online; visited 27.04.2018]. Retrieved from: https://developer.apple.com/streaming/

[4] Barry, P. J.; Goldman, R. N.: *A recursive evaluation algorithm for a class of Catmull-Rom splines. ACM SIGGRAPH Computer Graphics*. vol. 22, no. 4. 1988: pp. 199–204.

[5] Bellard, F.: *About FFmpeg*. 2018. [Online; visited 06.05.2018]. Retrieved from: https://www.ffmpeg.org/about.html

[6] Birrell, A. D.; Nelson, B. J.: *Implementing remote procedure calls. ACM Transactions on Computer Systems (TOCS)*. vol. 2, no. 1. 1984: pp. 39–59.

[7] Bradski, G.; Kaehler, A.: *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media Inc.. 2008.

[8] Canonico, R.; Cotroneo, D.; Russo, S.; et al.: *An Architecture for Streaming Control in Distributed Multimedia Systems*. 2000: page 4.

[9] Catmull, E.; Rom, R.: *A class of local interpolating splines*. In *Computer aided geometric design*. Elsevier. 1974. ISBN 978-0-12-079050-0. pp. 317–326.

[10] Dailymotion: *hls.js*. 2017. [Online; visited 06.05.2018]. Retrieved from: https://video-dev.github.io/hls.js/docs/html/

[11] Deng, G.; Cahill, L.: *An adaptive Gaussian filter for noise reduction and edge detection*. In *Nuclear Science Symposium and Medical Imaging Conference*. IEEE. 1993. pp. 1615–1619.

[12] Fette, I.; Melnikov, A.: *The WebSocket Protocol*. RFC 6455. RFC Editor. December 2011.

[13] FFmpeg Developers: *ffserver program being dropped*. 2016. [Online; visited 06.05.2018].
Retrieved from: https://www.ffmpeg.org/

[14] Google Inc.: *About gRPC*. 2018. [Online; visited 10.05.2018].
Retrieved from: https://grpc.io/about/

[15] Huitema, C.: *Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP)*. RFC 3605. RFC Editor. October 2003.

[16] ITU: *ITU-T Recommendation H.264 (04/2017)*. ITU-T E 41560. International Telecommunication Union. April 2017.

[17] Ivančo, M.: *Attractive Effects for Video Processing*. In *Excel@Fit Conference*, vol. 4. FIT VUT. May 2018.

[18] Jennings, C.; Burnett, D.; Aboba, B.; et al.: *WebRTC 1.0: Real-time Communication Between Browsers*. Candidate recommendation. W3C. November 2017.

[19] Massiot, C.: *VLC media player API Documentation. Mars*. vol. 20005. 2001.

[20] Pantos, R.; May, W.: *HTTP Live Streaming*. RFC 8216. RFC Editor. August 2017.

[21] Parmar, H.; Thornburgh, M.: *Adobe's Real Time Messaging Protocol (RTMP) Specification. Adobe Developer Connection*. vol. 52. December 2012.

[22] Patrikakis, C.; Papaoulakis, N.; Stefanoudaki, C.; et al.: *Streaming content wars: Download and play strikes back*. In *International Conference on User Centric Media*. Springer. 2009. pp. 218–226.

[23] Polesel, A.; Ramponi, G.; Mathews, V. J.: *Image enhancement via adaptive unsharp masking. IEEE transactions on image processing*. vol. 9, no. 3. 2000: pp. 505–510.

[24] Rauch, G.: *Socket.IO*. 2018. [Online; visited 10.05.2018].
Retrieved from: https://socket.io/

[25] Schulzrinne, H.; Casner, S.; Frederick, R.; et al.: *RTP: A Transport Protocol for Real-Time Applications*. STD 64. RFC Editor. July 2003.

[26] Schulzrinne, H.; Rao, A.; Lanphier, R.; et al.: *Real-Time Streaming Protocol Version 2.0*. RFC 7826. RFC Editor. December 2016.

[27] Smith, J.; Colwell, A.; Watson, M.; et al.: *Media Source Extensions*^{TM}. Recommendation. W3C. November 2016.