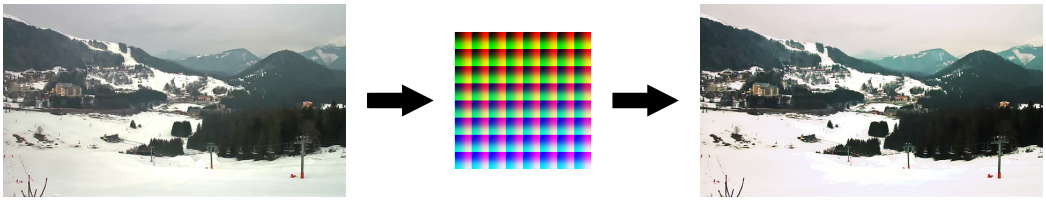
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2018
http://excel.fit.vutbr.cz

Attractive Effects for Video Processing

Martin Ivančo*



Abstract
The goal of this work was to make a server solution for processing video from IP cameras and a web interface. Video processing in this work focuses on making the video more attractive to the viewer by applying various effects to it. Web interface is an essential part of the work which makes the server solution easily accessible. It enables the user to adjust settings and apply colorful filters, and then share the processed stream via a permanent link. The prototype solution is currently limited to single user use, but it can be relatively easily upscaled. As there is little or very incomplete information on this specific problem, this work can be useful in solving similar problems.

Keywords: Real-Time Video Processing — Color Filter Application — Live Video Stream Enhancement

Supplementary Material: Demonstration Video — Downloadable Code

*xivanc03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

IP cameras became a very cheap technology over the years, which lead to a significant increase in the number of locations covered by camera surveillance. Majority of these cameras are used for security purposes, but an increasing amount of cameras serve as a live picture of the location for potential visitors. IP cameras, however, were not designed for this purpose. They were designed for low bandwidth streaming later to be used as evidence of a potential crime. As a result, images served by these cameras often lack contrast or saturation, and are not very visually pleasing overall. This can discourage tourists from visiting a location based on the underwhelming images served by the camera.

This work addresses the problem of visually unpleasant images streamed by many IP cameras. 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 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.

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 and process and display it just locally on the running machine, existing libraries like `OpenCV` would be sufficient to solve this problem. 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.

The solution proposed in this paper uses available

tools to construct a pipeline capable of receiving, processing and further streaming a live IP camera video. This pipeline runs on a server which acts as a communication middleman between the pipeline and the web user interface. 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.

2. Fundamental Live Video Streaming Principles

The demand for video streaming has been here for a long while and throughout that time it has evolved a lot. Classic frame by frame **MJPEG** streaming is no longer as popular as it used to be, because newer, much more efficient protocols have taken its place. The two most popular protocols are currently **RTSP** and **HLS** [1]. In my project I chose the latter. Although RTSP has much lower latency when compared to HLS, it is much harder to implement, either on the side of the server or the client.

HLS stands for HTTP Live Streaming [2] and as the name suggests, its major advantage is that it works on top of HTTP and does not need any special communication protocol. Figure 1 shows a diagram explaining how this protocol works. In my project there is only one *media playlist* reference in the *master playlist*, but this is something that could be improved in the future. The protocol works by segmenting the streamed video into smaller chunks and references to them are continually updated in the media playlist. The client then loads this playlist via a basic **HTTP GET** and therefore gains access to the video chunks which it can now load and play in order.

There is one other rule that needs to be met when working with HLS, that caused me problems while trying to display the stream in the browser, as described in Section 4.2. The master playlist referencing available media playlists should also contain information about them, especially the appropriate bandwidth size, codec, resolution and frame rate of the video. Determining the correct codec can be particularly challenging. The video codec is generally `avc1` but the corresponding flags are not so obvious. For this I needed to look at the standard ITU-T **H.264** Recommendation [4]. I encoded the video according to HLS documentation [5] with a **high profile** at **level 4.0** so that it is compatible with most devices without compromising the quality too much. As I learned from the H.264 Recommendation, the corresponding flags to this settings are `640028`. In it, the `64` is `profile_idc` representing the high profile. The `00` is `profile_iop`

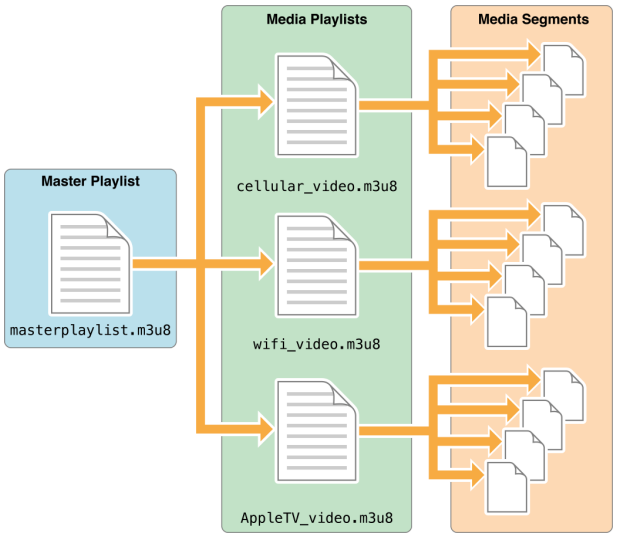


Figure 1. HTTP Live Streaming Diagram [3]. The `masterplaylist.m3u8` contains references to multiple media playlists for various use cases, such as streaming video using cellular, wifi or for high quality streaming for TV. Each of the media playlists then contains references to video chunks with corresponding quality.

which means no constraint flags are set. Finally, the `28` is `level_idc` which represents the level 4.0 according to H.264 Recommendation. All in all, my media playlist reference looked like this:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=4000000,\
 CODECS="avc1.640028",RESOLUTION=1280x720,\
 FRAME-RATE=25
output.m3u8
```

3. Used Image Processing Methods

Being the essential part of the project, image processing needed to have a solid starting ground. For this, `OpenCV` [6] was chosen as the most widely used open source library for image processing and computer vision. Thanks to its capability of loading almost any type of video source, I was quickly able to display frames processed by `OpenCV` locally. With its built in functions I was able to easily adjust brightness and saturation, or convert the frames to black and white. Adjusting contrast, however, is not natively built into `OpenCV`. As I planned to implement *tone curve* for the final version of the project, I decided to base contrast on that.

3.1 Implementation of Tone Curve

The tone curve represents a mapping function for brightness values of pixels. The X coordinate represents the input value and the Y coordinate represents the output

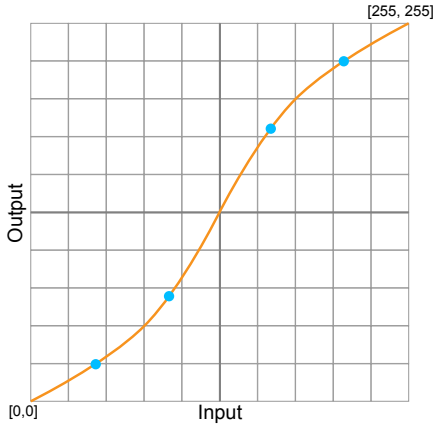


Figure 2. Tone curve example. Enhancing contrast in midtones while reducing it in near black or near white colors is appropriate for most images. Blue dots represent tone curves control points.

value. Figure 2 shows an example of a curve that adds contrast to the image.

Tone curve is defined by 4 control points called *shadows*, *darks*, *lights* and *highlights*. It also has two stable points at $[0,0]$ and $[256,256]$. Using these points, the curve is calculated as a **Catmull-Rom Spline** [7]. For an input value x, the output y is calculated using Equation (1). 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 till Unfortunately, this does not allow to easily calculate output value for a given input (in other words, get y for x). We can solve this problem by dividing the curve segment into several parts to get a reasonable amount of reference points and then using linear interpolation between two of them whose x coordinates match the input value the closest. Figure 3 might explain this solution better.

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

$$\mathbf{C} = \frac{t_2 - t}{t_2 - t_1}\mathbf{B}_1 + \frac{t - t_1}{t_2 - t_1}\mathbf{B}_2 \tag{1}$$

where

$$\begin{aligned}
\mathbf{B}_1 &= \frac{t_2 - t}{t_2 - t_0}\mathbf{A}_1 + \frac{t - t_0}{t_2 - t_0}\mathbf{A}_2 \\
\mathbf{B}_2 &= \frac{t_3 - t}{t_3 - t_1}\mathbf{A}_2 + \frac{t - t_1}{t_3 - t_1}\mathbf{A}_3 \\
\mathbf{A}_1 &= \frac{t_1 - t}{t_1 - t_0}\mathbf{P}_0 + \frac{t - t_0}{t_1 - t_0}\mathbf{P}_1 \\
\mathbf{A}_2 &= \frac{t_2 - t}{t_2 - t_1}\mathbf{P}_1 + \frac{t - t_1}{t_2 - t_1}\mathbf{P}_2 \\
\mathbf{A}_3 &= \frac{t_3 - t}{t_3 - t_2}\mathbf{P}_2 + \frac{t - t_2}{t_3 - t_2}\mathbf{P}_3
\end{aligned} \tag{2}$$

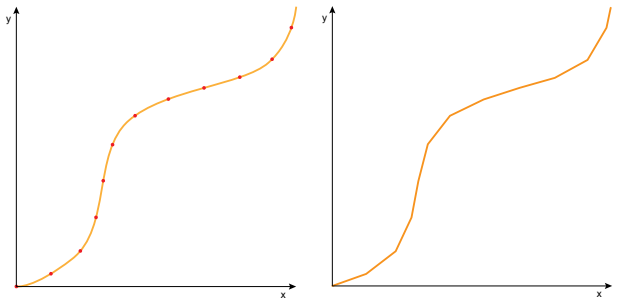


Figure 3. The left part of the picture shows the ideal curve defined by four control points, and highlighted points that have been calculated using Equation (1). The right picture shows how the curve looks when missing points are calculated from the highlighted points using linear interpolation.

and

$$t_{i+1} = \left[\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}\right]^{\alpha} + t_i \tag{3}$$

where $t_0 = 0$ and $\alpha = 0.5$ as a **Centripetal** Catmull-Rom Spline is wanted.

These calculations are not computationally trivial and they introduced a significant slowdown to the program. So far, each pixel had to be recalculated on its own. This was very inefficient so I decided to calculate the output value for every possible input value[1] beforehand and store these values into an array. The time needed to process each frame decreased significantly. This inspired me to think if it would be possible to do something similar for other types of adjustments, not just contrast and brightness, which led me to implement RGB **look up tables**. Thanks to them the program was processing frames *much faster*. Figure 4 shows how it works.

4. Completing the Chain

The processed video now needs to be streamed via a streaming protocol and then received by the browser client that needs to display it. The problematics of these tasks are explained in this section.

4.1 Streaming Processed Video

Streaming the processed video turned out to be a difficult problem to overcome. `OpenCV` has great receiving capabilities, but it is not designed for streaming the processed content. Therefore I needed to pass the processed data to another library or tool. This was the toughest part. I did a lot of research and I tried quite a few approaches to this problem, but none of them

[1] The values range from 0 to 255 as we are dealing with a classic 8-bit RGB color space.

Figure 4. The advantage of using look up tables instead of individual adjustments are obvious.

seemed to work properly. I tried `libVLC`, which is very capable on its own, but I could not find a way to correctly transfer the processed video to it. I also looked into other technologies, such as `WebRTC`, but none of them fit my needs completely.

After a lot of struggle, `FFmpeg` solved this problem, although with some caveats. `FFmpeg` is a widely used command line tool to convert and transform video. Although it is built on several libraries which can be included and used freely in any program, the documentation for them is poor. `FFmpeg`, however, is able to receive data via standard input, which is exactly what I used. This solution is not optimal, but it is a working method. Using the official documentation I was able to run the command below and `pipe` the processed frames into it. In the future, I would like to find an alternative and more efficient solution.

```
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
```

4.2 Displaying Video in Browser

Displaying the streamed video in browser did not propose any major difficulties thanks to `hls.js`. This javascript library enables loading and playing HLS streams using an easy to use API.

5. Communication Protocols

Although we are now able to stream the processed video in the browser, we still need to ensure some sort of communication between the user and the image processing program which can then execute users requests. For this, we will need a *middle man*, a piece of software that will listen to requests sent by client and then process them by calling the image processing program. This is implemented as a part of the web server, which is written in `Node.js`. Thanks to `Node.js` we can take advantage of the variety of different libraries and frameworks available to communicate with either the client or the image processing program.

5.1 RPC Protocols

RPC stands for *Remote Procedure Call* [8] and it does just that. It is a way to call functions from a different program. `GRPC` is a great RPC framework initially developed and used by Google, which was later open-sourced and is now available to the public. It supports a wide variety of programming languages, including `Node.js` (web server) and `C++` (image processing program) which is exactly what we need.

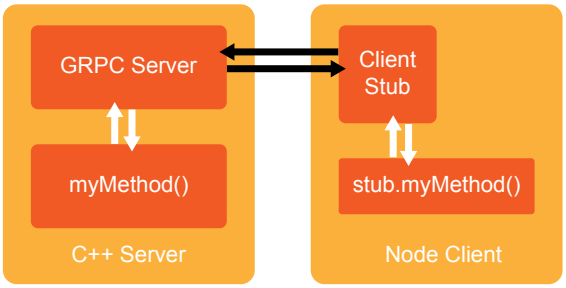


Figure 5. Diagram representing the use of `GRPC` to communicate between the video processing server written in C++ and the Node client.

Figure 5 shows the use of `GRPC` in this project. All the complicated communication is handled by `GRPC`, we simply need to implement the setting methods in the image processing program according to `GRPC`s API and then instantiate a *stub* in the `Node.js` web server (which acts as a client in this scenario). After that it is as simple as calling a method in `Node.js`.

5.2 Web Sockets

The last remaining part of connecting the web server to the clients web user interface is done using *web sockets*. This protocol enables bidirectional communication between the server and the clients browser.

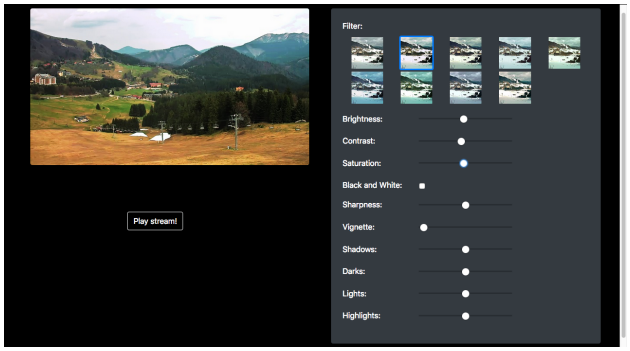


Figure 6. The final user interface capable of adjusting the settings using sliders and buttons on the right side. The adjustments are visible almost instantly.

A great framework implementing this protocol for `Node.js` is `Socket.IO`. Thanks to `Socket.IO`, we only need to include its javascript client library to the web user interface and then send a message via the `socket` every time the user changes a setting. `Node.js` web server can then handle this message properly.

Finally, after connecting all parts of this solution, the user is able to view the processed stream right in the browser, as well as adjust the settings. Figure 6 shows the final look of the user interface for adjusting the settings.

6. Conclusions

This paper explained a video processing server solution implementation in detail. It explained the chosen **HLS** protocol used for streaming the processed video and then proceeded with explanation of implemented image processing methods, such as **tone curves** and **look up tables**. Finally it showed a way to bind all the parts together using communication frameworks such as **GRPC** and **Socket.IO**.

Resulting program is capable of loading a **video stream** submitted by the user via a web user interface, **processing it** and streaming the processed video which shows up in the user interface. The program allows user to set the adjustments made to the video via the web interface, such as tweaking the brightness, contrast, saturation, or changing the entire look of the video by applying colorful filters.

In its current state, the program only supports one user at a time, but this can be changed relatively easily. This is also one of the future improvements that I would like to implement. However, I believe that even now it can serve as something that can help other developers tackling a similar problem. As the pipeline of receiving, processing, streaming and displaying a live stream in the browser has been solved, **other video effects** can easily be implemented too. Video processing in real time is a difficult task, and therefore the solution needs to run on a quite powerful machine to run smoothly. The delay between submitting the video stream url and displaying it in the browser is quite long (usually around 10 to 15 seconds on my computer), but again, the length of the delay depends on how powerful the machine that the program runs on is.

Acknowledgements

I would like to thank my supervisor Adam Herout for his help.

References

[1] Paul Berberian. How video streaming works on the web: An introduction. *Medium.com*, January 2018.

[2] R. Pantos and W. May. Http live streaming. RFC 8216, RFC Editor, August 2017.

[3] Apple Inc. About http live streaming. [online], October 2014. https://developer.apple.com/library/content/referencelibrary/GettingStarted/AboutHTTPLiveStreaming/about/about.html.

[4] ITU. Itu-t recommendation h.264 (04/2017). [online PDF], April 2017. https://www.itu.int/rec/T-REC-H.264-201704-I/en.

[5] Apple Inc. Using http live streaming. [online], March 2016. https://developer.apple.com/library/content/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/UsingHTTPLiveStreaming/UsingHTTPLiveStreaming.html.

[6] OpenCV Team. *The OpenCV Reference Manual*, 3.4.1 edition, February 2018.

[7] Raphael Rom Edwin Catmull. A class of local interpolating splines. In Richard F. Reisenfeld Robert E. Barnhill, editor, *Computer Aided Geometric Design*, pages 317–326. Elsevier, 1974. ISBN 978-0-12-079050-0.

[8] Bradley Mitchell. Rpc - remote procedure call. [online], February 2018. https://www.lifewire.com/remote-procedure-call-816432.