Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Aleksandra Zhuravleva

Progressive Web Camera Application using OpenCV WebAssembly module

Master's Thesis
Espoo, June 11, 2020

Supervisor: Prof. Petri Vuorimaa, Aalto University
Advisor: Rijubrata Bhaumik, Intel Corporation

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Aleksandra Zhuravleva		
Title:	Progressive Web Camera Application using OpenCV WebAssembly module		
Date:	June 11, 2020	**Pages:**	ix + 65
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Prof. Petri Vuorimaa, Aalto University		
Advisor:	Rijubrata Bhaumik, Intel Corporation		

The Web platform is a low friction, linkable and universal platform where users can create powerful applications using JavaScript and expect it to run everywhere. However, most of the powerful camera applications or media capture solutions are implemented using native technologies for that platform.

This thesis presents WebCamera application with computer vision capabilities performed by OpenCV library. A set of use cases include Instagram filters, Card scanning, Emotion recognition and others. The WebCamera is designed as a progressive web application to provide native-like features such as app installation, offline mode and responsive screen size.

OpenCV is built by Emscripten compiler in WebAssembly module to achieve near-native speed on the web thanks to SIMD and threads optimization options. Measured performance statistics demonstrate that these optimizations allow us to reach up to 9x speedup compared to not optimized OpenCV version in a browser.

WebCamera demos are showcased at Chrome Dev Summit 2019 in the talk about WebAssembly. Moreover, V8 engine's web page called "Fast, parallel applications with WebAssembly SIMD" presents WebCamera's use cases implemented in this work.

Keywords:	OpenCV, WebAssembly, progressive web application, computer vision, image processing, face detection, card scanning, document enhancement, emotion recognition, color segmentation
Language:	English

Acknowledgements

I would like to express my deepest gratitude to Prof. Petri Vuorimaa, my supervisor from Aalto University, for carefully reading every draft and providing valuable feedback. His guidance and encouragement helped me while writing this thesis.

Furthermore, this master's studies could not have been completed without support and outstanding ideas of Rijubrata Bhaumik, my friend and advisor from Intel Corporation. I want to thank Riju for giving me the opportunity to work on WebCamera project and for leading me through every single step of this research.

Last but not least, I wish to extend my special thanks to my family, especially to my husband, for keeping me away from stress and helping me to stay focussed and motivated during the challenging time. I appreciate the optimism and love of my family supporting me in my personal development.

Espoo, June 11, 2020

Aleksandra Zhuravleva

Abbreviations and Acronyms

CV	Computer Vision
ML	Machine Learning
DL	Deep Learning
AI	Artificial Intelligence
DNN	Deep Neural Network
AR	Augmented Reality
Wasm	WebAssembly
PWA	Progressive Web Application
CPU	Central Processing Unit
GPU	Graphics Processing Unit
VPU	Vision Processing Unit
SIMD	Single Instruction Multiple Data
pthreads	POSIX threads
API	Application Programming Interface
UI	User Interface
HTML	HyperText Markup Language
JS	JavaScript
CSS	Cascading Style Sheets
FPS	Frames Per Second
HSV	Hue, Saturation, Value
RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
W3C	World Wide Web Consortium
WOFF	Web Open Font Format
SW	Service Worker
Workbox CLI	Workbox Command Line Interface
OCR	Optical Character Recognition

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Background and Motivation

Over the past decade, capturing and sharing of digital media has skyrocketed due to the proliferation of smartphones, digital cameras and the popularity of social networks. This immense trove of data, when labeled like ImageNet dataset[1], and the affordable computing resources, like GPUs (Graphics Processing Units) or advancement in CPUs (Central Processing Units) such as SIMD instructions, have made Computer Vision (CV) applications achieve accuracies that surpasses human abilities.

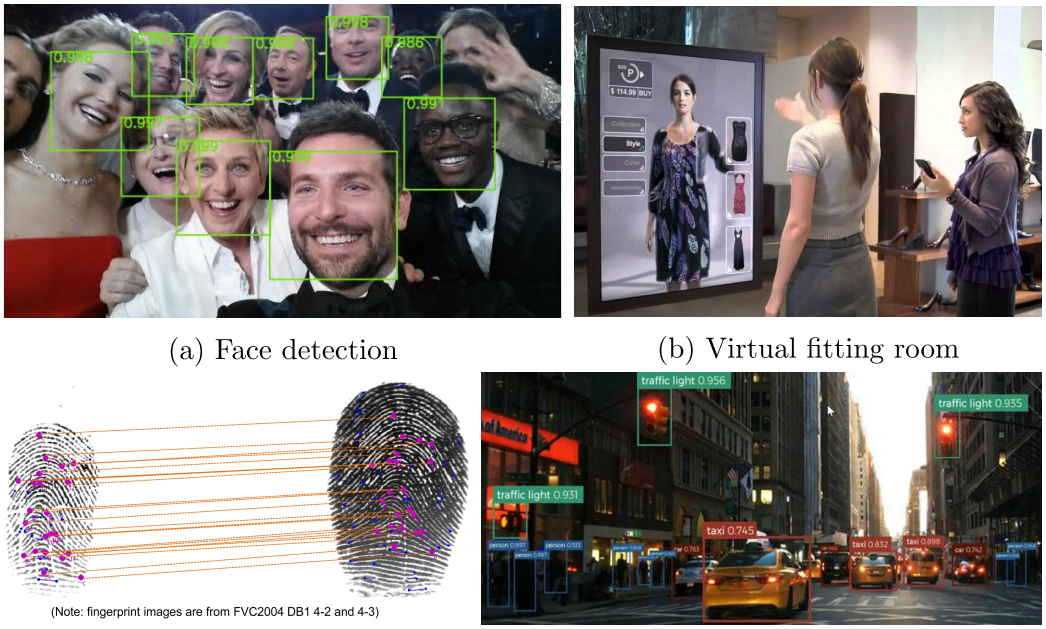
CV is an interdisciplinary research field originating in the 1960s which aims to learn and simulate tasks performed by the human visual system [7]. CV encompasses Machine Learning (ML) techniques and image processing algorithms to derive and analyze meaningful information from digital media [17]. Common usages of CV (Figure 1.1) include face recognition, fingerprint matching, detection of suspicious human behavior, traffic lights analysis in self-driving cars, virtual fitting rooms and many more.

Usually, client devices are equipped with hardware good enough for inferencing, but sometimes heavier workloads are augmented by using edge or cloud services such as Microsoft Azure Cognitive Services, Google Cloud Vision or Amazon Rekognition. As an example, Amazon Rekognition software allows us to identify objects, people, text, scenes, and activities in images as well as in videos or provide highly accurate facial analysis capabilities [23]. This might concern a privacy-aware user preferring data and computation to happen on the client device itself.

Until recently, CV algorithms have been developed only for native appli-

[1]ImageNet is a large visual database designed by Stanford University to use in object recognition research.

cations, that is, for use on a particular platform or device, mostly because the algorithms needed a lot of computation power. Using a native language and platform provided better access to the hardware needed to compute the workload. The universality and low friction aspects make the Web Platform a true platform of choice for scaling to the masses. Web applications can reach anyone, anywhere, on any device with a single codebase.



(a) Face detection

(b) Virtual fitting room

(c) Fingerprint matching

(d) Traffic lights analysis

Figure 1.1: Computer vision practical applications

Although CV is fairly new to the web platform development, plenty of libraries have already been designed to be run in a browser. Some of the most common examples of CV JavaScript libraries are Tensorflow.js, Tracking.js, Jsfeat, WebGazer.js, Headtrackr, Clmtrackr and Three.ar.js [37, 65]. To ensure better user experience, most of the libraries perform complex graphics tasks by using WebGL, which is a JavaScript API (Application Programming Interface) for rendering high-performance interactive 2D and 3D graphics [9, 11].

Until recently, JavaScript was the only language of choice to develop applications for the Web. A new portable binary instruction format called Web Assembly (Wasm) is used alongside JavaScript for high performance and compute-heavy workloads for the Web Platform [6]. Porting native libraries to the Web Platform is accomplished using the Emscripten toolchain. Existing C, C++ or Rust projects ported to Wasm can be run on the web at

near-native speed thanks to SIMD and threads optimizations [22, 33].

Speaking of optimizations, SIMD instructions are a special class of instructions that exploit data parallelism by performing the same operation on multiple data elements simultaneously. Therefore, compute-intensive applications, such as image processing, take advantage of the data level parallelism using the underlying hardware, thus, accelerating the performance. WebAssembly SIMD proposal introduces a new v128 value type and a number of operations that utilize this type. [67]

Furthermore, most native applications implement parallel computing using threads, like POSIX threads (pthreads) in C/C++. WebAssembly threads [31] are the Web platform equivalent of pthreads, a kind of threads that can share the same Wasm memory accomplished by a JavaScript primitive called SharedArrayBuffer [46]. SIMD and Web Assembly threads together have bridged the performance gap between Web Platform and the native environment by a wide margin [8, 18]. OpenCV.js was one of the first frameworks to utilize the power of SIMD and Wasm threads.

OpenCV [50] has been one of the stalwarts among the Image Processing libraries from the advent. Moreover, OpenCV has added a Deep Neural Network (DNN) module to further utilize a variety of ML models trained by TensorFlow, Caffe, Torch and other ML frameworks [56]. Thus, OpenCV library has very powerful functionality for both image processing and object detection, so it is popular in commercial and open source projects, in industry and academic environment [4].

In addition to the mentioned web technologies, Progressive Web Applications (PWA) are built and enhanced with modern web APIs to deliver native-like capabilities, reliability, and installability to web platforms [59]. All major browsers like Chrome, Edge, Safari, Firefox and Opera have shipped the building blocks that collectively constitute Progressive Web Apps [16, 63]. PWA allows us to add a launcher icon on a phone's desktop, exploit an app in offline mode or receive push-notifications [16]. Plenty of well-known mobile applications such as AliExpress, Twitter Lite, Uber, Spotify, Pinterest already have a PWA equivalent [25].

All these developments for web platforms have contributed to the range of web applications that is gradually expanding, including applications that previously were only available as native apps. Nowadays, we can run such popular apps as VLC media player [45] or AutoCAD computer-aided design tool [44] completely in a web browser. However, there is still no popular web camera counterpart for applications that are based on modern CV methods, such as drawing virtual objects for a recognized face, scanning a credit card or a document.

1.2 Thesis aim

The Web platform is a low friction, linkable and universal platform where users can create powerful applications using JavaScript and expect it to run everywhere. However, most of the powerful camera applications or media capture solutions, such as video conferencing tools with AI capabilities, are implemented using native technologies for that platform. For example, the popularity of social networks like Snapchat and Instagram has given rise to "Image Filters". Similarly, Facebook Messenger provides a feature called "Funny Hats". Video conferencing solutions such as Skype or Microsoft Teams offer advanced features like "Face Detection" and "Background Segmentation". These compute-intensive CV and image processing problems are mostly executed on specialized hardware like GPU, VPU (Vision Processing Unit) or some other AI accelerator.

In this context, the main research question of this work is:

- Is it feasible to create a CPU-based CV application for the web platform in a fashion where there is a negligible loss of user-perceived quality?

Moreover, the thesis attempts to answer the following supporting questions:

- What are optimization methods for efficient code execution in a web browser?
- Can a well-known OpenCV library be used to implement popular CV and image processing tasks for the web platform? Does it have required functionality?
- How to achieve a native-like experience for a web application?

Thus, the aim of this thesis is to develop a web camera application performing CV and image processing tasks in a browser and utilizing only CPU resources. In order to achieve this goal, the thesis will design and implement the app in JavaScript using OpenCV library, PWA features, Emscripten tool as well as WebAssembly format with SIMD and thread-based parallelism. The thesis will explore available functionality of OpenCV to develop seven popular media capture use cases in the application: Instagram Filters, Face Detection, Funny Hats (virtual hats and glasses), Card Scanning, Document Enhancement, Emotion Recognition and Invisibility Cloak (color segmentation).

1.3 Thesis structure

The rest of the thesis is organized as follows. Chapter 2 reviews existing CV libraries available for the web platform and CV applications related to web camera experience. Chapter 3 introduces the environment and methods applied in the application. Chapter 4 describes the implementation of each web camera use case. Chapter 5 demonstrates the use cases in a browser and presents performance statistics. Chapter 6 analyses results in terms of performance based on different optimization options, discusses limitations of the designed application and suggests improvements for future development. Finally, Chapter 7 summarizes the work.

Chapter 2

Related work

This chapter reviews CV libraries for the web and existed CV applications.

2.1 Review of CV libraries

The number of CV and image processing frameworks is growing steadily. To create WebCamera application, I need a robust and efficient library with a comprehensive set of CV capabilities ranging from various image processing algorithms to object detection and recognition. In this section, I introduce OpenCV, review other popular CV libraries available for the web platform and compare their functionality with OpenCV. I will not consider cloud-based CV solutions like Microsoft Azure Cognitive Services, Google Cloud Vision or Amazon Rekognition as they perform processing in data centers sending user data over the Internet. Though, they provide JavaScript APIs for CV and image processing tasks. Instead, I will focus on frameworks that utilize the power of client machines demonstrating CV capabilities in a browser. It can be either a Wasm version compiled from other programming languages, framework with WebGL backend or plain Javascript library. Wasm provides near-native execution in a browser thanks to efficient binary instructions as well as SIMD and threads optimizations [6]. Read Section 3.2 to find more details about Wasm format. WebGL represents a JavaScript API for accelerated 3D graphics on the web utilizing GPU resources [9, 11].

OpenCV is an Open Source Computer Vision library started by Intel in 1999. The library functionality has been gradually expanding, and now it contains more than 2500 of CV and image processing algorithms. Application field of OpenCV is quite exhaustive and includes object identification and segmentation, face and gesture recognition, motion tracking, augmented reality, mobile robotics, egomotion estimation and many others. DNN mod-

ule of OpenCV supports models generated by such ML frameworks as Caffe, TensorFlow, Torch, Darknet, ONNX and Intel's Model Optimizer. OpenCV is distributed under a BSD license, thus, it is widely used in academic and commercial projects. [50]

Moreover, OpenCV is a cross-platform library as it provides support for Windows, Linux, Mac OS and Android. The source code is written in C/++ languages, however, it has Python, Java and MATLAB interfaces as well [50]. There are also unofficial wrappers for Rust, Ruby, Perl, Haskell, C# and other languages. In 2017, OpenCV released JavaScript bindings making available CV for the Web platform [51]. Thus, OpenCV.js can be ported to either asm.js or Wasm format using Emscripten compiler.

Similar to OpenCV, there are other recognized frameworks with Wasm backend, such as TensorFlow.js, ONNX.js, Keras.js and WebDNN, able to detect and classify objects in a browser [10, 62]. These state-of-the-art solutions combine powerful functionality to experiment with various ML models. For example, they can effectively perform face detection, emotion recognition, pose estimation or semantic segmentation. In addition to model execution, Tensorflow.js enables model training. While WebDNN is able to run any neural network trained by TensorFlow, Keras, Caffe or Pytorch, Keras.js accepts only Keras models, ONNX.js executes ONNX models, and TensorFlow.js works with both TensorFlow and Keras models [10]. However, the capabilities of these frameworks are limited to ML, Deep Learning (DL) and Artificial Intelligence (AI) areas, so they don't contain image processing algorithms.

In contrast to the mentioned ML frameworks, quite popular CV libraries like Tracking.js and JSFeat combine a wide range of algorithms for image processing and only few methods for object detection. For instance, Tracking.js is capable of real-time color tracking, feature and object detection [43]. Object detection is based on Haar Cascades approach [43]. In addition to Haar detector, JSFeat is able to detect objects using Brightness Binary Feature (BBF) technique. Similar to Tracking.js and JSFeat, other libraries, such as Lena.js [32], CamanJS [42] and MarvinJ [24], perform some image filtering like grayscale, thresholding, blurring or edge detection [70]. However, none of them provides either Wasm or WebGL backend, so they are plain JS libraries.

Other less famous but still impressive CV libraries are WebGazer, Headtrackr, Clmtrackr and Three.ar.js. They are pure JS libraries, however, they use WebGL API, except for Headtrackr. Each library has a highly specialized focus. For example, WebGazer is an eye tracking solution that infers eye-gaze locations of web visitors on a page [12]. It self-calibrates eye model by evaluating how web visitors interact with a web page. While Headtrackr

library offers functionality for a face and head tracking using the position of user's head in relation to the computer screen [53], Clmtrackr provides more precise face control and recognizes facial features including eyes, nose, lips and brows via constrained local model fitted by regularized landmark Mean-Shift [14, 54]. Finally, Three.ar.js combines helper classes for building Augmented Reality (AR) experience, i.e., real-time interaction with an object in 3D environment [36, 40].

Framework	Github stars	Main contributor	Functionality	WebGL	Wasm	Plain JS	Last commit date
OpenCV	44537	Intel	CV and image processing	No	Yes	Yes	15.05.20
TensorFlow.js	13225	Google	Training and executing ML models	Yes	Yes	Yes	16.05.20
ONNX.js	1060	Microsoft	Executing ONNX models	Yes	Yes	Yes	01.06.19
Keras.js	4724	Leon Chen	Executing various ML models	Yes	No	Yes	16.08.18
WebDNN	1714	The University of Tokyo	Executing DNN pre-trained models	Yes	Yes	Yes	17.01.20
tracking.js	8461	Eduardo Lundgren	Image processing, feature and object detection	No	No	Yes	17.05.18
jsfeat	2531	Eugene Zatepyakin	Image processing, feature and object detection	No	No	Yes	03.03.18
Lena.js	348	Davidson Fellipe	Image filtering	No	No	Yes	06.04.20
CamanJS	3289	Ryan LeFevre	Image filtering	No	No	Yes	20.02.20
MarvinJ	127	Gabriel Ambrosio Archanjo	Image filtering	No	No	Yes	24.07.19
WebGazer	2403	Brown University	Eye tracking	No	Yes	Yes	08.08.19
Headtrackr	3612	Audun Mathias Oygard	Face tracking and head tracking	No	No	Yes	18.05.14
Clmtrackr	6062	Audun Mathias Oygard	Precise tracking of facial features	No	Yes	Yes	22.11.18
Three.ar.js	2413	Google	Augmented reality experience	No	Yes	Yes	02.03.18

Table 2.1: Computer vision frameworks for the web

Table 2.1 summarizes reviewed libraries and presents some data including Github stars, main contributors, core functionality, presence of Wasm or WebGL backend and date of the last commit as of May 15th, 2020. In contrast to OpenCV.js, most of the described CV libraries for the web utilize WebGL API to perform compute-intensive tasks on GPU while OpenCV has only Wasm backend, i.e., perform processing on CPU. However, OpenCV encompasses more considerable functionality combining both ML capabilities

and a wide range of image processing algorithms compared to presented CV frameworks. It provides grained control for any kind of CV use case, thus, it is a more suitable option for implementing WebCamera application. WebCamera requires functionality for face detection and emotion recognition as well as a number of image filters to perform such complex tasks as card scanning and document enhancement. Later it may need even more features to extend the app with new use cases. Therefore, OpenCV is an obvious choice for WebCamera development in terms of available CV algorithms. Moreover, it raises an intriguing research question on how efficiently it can perform processing in a browser utilizing only CPU power.

2.2 Existing CV applications

Nowadays, CV applications surround us everywhere. It can be as simple as face detection or as complex as road traffic analysis. A great number of applications related to camera experiences, such as Instagram, Snapchat or CamScanner, are built for mobile platforms like Android and iOS. Sometimes CV features are integrated to apps as a service like "Funny hats" in Facebook Messenger, face detection or background segmentation in Skype and Microsoft Teams applications.

The next step forward in application development is to make these apps cross-platform, e.g., develop them on the web, so that users can access them from whichever device with an internet browser and feel that they still have native-like features and performance. My research on existing solutions of CV apps shows that there is still no web equivalent for a kind of Instagram filters, Funny hats or document scanner that perform processing in a browser but not in a cloud. Even Instagram, Snapchat and Facebook websites have not implemented their CV features in a browser version. Moreover, lists of popular PWAs [38, 41, 57] proves that while there are already plenty of handy web applications for sport, news, traveling, etc., there is still no any kind of native-like web camera application with CV capabilities. A couple of found PWA examples [48, 61] demonstrate only image capture functionality.

However, there are some not PWA but still web demos of image filtering using Lena.js [32], CamanJS [42] and MarvinJ [24] JS libraries. In addition, Tensorflow.js demonstrates ML use cases in a browser such as an image classification, pose estimation, handwritten digit recognition and many others [66].

Since I implement WebCamera application using OpenCV library, I mostly rely on OpenCV tutorials and other existing solutions based on this library including C++ and Python implementations because OpenCV API is similar

for any supported programming language. I want to mention related work that has most influenced my application and even served as the basis for some use cases. The first work is OpenCV tutorial on image processing that gives comprehensive code examples and descriptions for each filter [52]. The other two important solutions are card scanning and document enhancement algorithms developed in Python by Adrian Rosebrock from Pyimagesearch [60]. The next example is Facemoji emotion recognition project [30] written in Python that uses Fisher Faces model (see Section 3.1.3 about Fisher Faces algorithm). The last demo is Invisibility Cloak using color segmentation through HSV color space.

Chapter 3

Environment

Up to this point, tools and methods used in this thesis have only been mentioned in a general context. This chapter introduces environment of the work. The first section describes OpenCV library including building steps and image prcessing algorithms applied in WebCamera app. The second section familiarizes readers with WebAssembly format and Emscripten compiler required to build OpenCV.js file. The third section presents PWA features integrated in the WebCamera.

3.1 OpenCV

OpenCV library contains more than 2500 of CV and image processing algorithms. It is implemented as a collection of modules, where each module is responsible for a subset of functionality [19]. For example, *Core* module contains basic structures and calculations. *Objdetect* module provides object detection algorithms like Haar Cascades. *Imgproc* module deals with image processing like filtering, geometric transformations, color space conversion, etc. *DNN* module is used for DNN inference and supports models generated by Caffe, TensorFlow, Torch, Darknet, ONNX and Intel's Model Optimizer. *Contrib* module is a module for a new contributions and contains the code which does not have stable API or is not well-tested yet. So this module is not a part of official OpenCV repository, however, it can be built with other modules to enable new extra features like Fisher Faces recognition or Face landmark detection.

In this section, I will describe steps to build OpenCV.js library as well as some object detection and image processing algorithms that I use in WebCamera application. Object detection is represented by Haar Cascades and Fisher Faces algorithms. Image processing includes a set of filters like

changing to gray or HSV colorspaces, thresholding, Gaussian/Median/Bilateral smoothing, morphological transformations, image gradients, Canny edge detection and histograms.

3.1.1 Building and using OpenCV

OpenCV.js is built from C++ code into either Asm.js or Wasm format by Emscripten compiler [19]. I use Wasm version of OpenCV.js as it is more compact and much faster than Asm.js. See Section 3.2 for more details about Emscripten compiler and Wasm format.

Listing 3.1 presents a list of commands to download and build OpenCV from source code. On the first line, I clone OpenCV repository from the Github. On the second line, I navigate to the folder of the cloned project. Finally, I execute `build_js.py` python script with arguments, where `build_wasm` is the name of the build forder, `--build_wasm` is a flag meaning that I build Wasm but not Asm.js version of OpenCV, `--threads` and `--simd` are optimization options, `--emscripten_dir` is a path to Emscripten compiler. Threads and SIMD optimization options were released in 2019 and demonstrate impressive performance for some OpenCV functions compared to not optimized build. In section 5.2, I present performance statistics of WebCamera use cases based on Threads and SIMD optimizations.

Listing 3.1: Download and build OpenCV.js

```
1 git clone https://github.com/opencv/opencv.git
2 cd opencv
3 python ./platforms/js/build_js.py build_wasm --build_wasm
    --threads --simd --emscripten_dir="<path to emsdk>/
    emsdk/upstream/emscripten"
```

For Emotion Recognition use case in WebCamera app, I use `FisherFaceRecognizer` class from the *Contrib* module. To make this class available in the build file, I have to provide a path to *Contrib* module and JS bindings for `FisherFaceRecognizer` class in the build scripts. This can be done with the following steps:

- Add `"-DOPENCV_EXTRA_MODULES_PATH=<path to contrib project>/opencv_contrib/modules"` and `"-DBUILD_opencv_face=ON"` flags in `build_js.py` script
- Append "`js`" inside `ocv_define_module` of `opencv_contrib/modules/face/CMakeLists.txt` file

- Provide bindings of `face` module in `embindgen.py` script:

```
face = {
 'face_FaceRecognizer': ['train', 'update', 'predict_label',
     'write', 'read', 'setLabelInfo', 'getLabelInfo', '
    getLabelsByString', 'getThreshold', 'setThreshold'],
 'face_BasicFaceRecognizer': ['getNumComponents', '
    setNumComponents', 'getThreshold', 'setThreshold', '
    getProjections', 'getLabels', 'getEigenValues', '
    getEigenVectors', 'getMean', 'read', 'write'],
 'face_FisherFaceRecognizer': ['create']
}
```

- Add the `face` module to the `makeWhiteList` in `embindgen.py` script
- Add "`using namespace face;`" in `core_bindings.cpp` file

To use OpenCV functions in JavaScript code, I need to load OpenCV.js file and wait until initialization is completed. First, I create a `script` tag for OpenCV.js in HTML file. Inside the tag, I add `async` attribute and provide a path to OpenCV.js file as `src` attribute. Then, I assign `onload` event listener to the script tag. Finally, I start image processing, when OpenCV.js is loaded.

3.1.2 Haar Cascade classifier

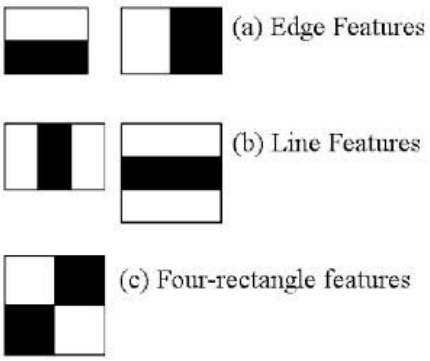
Haar Cascades algorithm is an effective ML based method for object detection proposed by Paul Viola and Michael Jones in the 2001 [20]. This method shows competitive precision rates [5] and even higher performance than some neural networks [68]. Before I chose this approach, I also tested face detection model trained by Caffe framework. The result showed that Haar Cascades are 2-3 times faster than Caffe model. In some sources, Haar Cascade method is also called Viola-Jones approach for object detection. I will focus on face detection and discuss face features required for the algorithm.

Haar Cascade classifier is trained on a set of images labeled as positive (with face) and negative (no face). The core basis of the classification is the Haar-like features that use the change in contrast values between neighboring groups of pixels to recognize the features [21]. The contrast is calculated for a rectangular region of adjacent pixels as a sum so that the group of pixels is represented as white or black rectangle as it is shown in Figure 3.1a. There are three types of features called two-, three- and four-rectangle features. In Figure 3.1b, I present two examples of face features. The first feature relies on fact that the eyes region is darker than the nose and cheeks. The

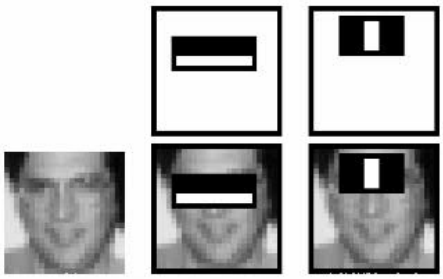
second feature assumes that the eyes are darker than the bridge of nose. By subtracting the sum of white region from the sum of black region, I get the feature value [21]:

$$FeatureValue = \Sigma(pixels_{blackRegion}) - \Sigma(pixels_{blackRegion}) \tag{3.1}$$

Since the most of the image is non-face area, Haar Cascades method applies the features on the image in Cascade manner. In the first cascade or stage, the algorithm checks one feature on the selected region. If the region passes the stage, then, it goes to the next stage. If the region fails, then, the algorithm no longer spends time on this region and moves to the next area, thus, increasing the speed of the detection. The number of features for the first five cascades is 1, 10, 25, 25 and 50, respectively. The total number of features in all stages of the algorithm is 6061. [20]



(a) Feature types



(b) Two face features

Figure 3.1: Haar Cascade features [20]

Listing 3.2: Initialize and load Haar Cascade model

```
1 faceCascade = new cv.CascadeClassifier();
2 faceCascade.load(faceDetectionPath);
3 ...
4 faceCascade.detectMultiScale(gray, faceVec);
```

OpenCV contains some pre-trained classifiers including face and eye detection. I use `haarcascade_frontalface_default.xml` and `haarcascade_eye.xml` models in Face Detection and Funny hats use cases of WebCamera app. Listing 3.2 shows how to initialize Cascade classifier and load the model from the XML file. On the third line of the listing, I run `detectMultiScale(..)`

method of the loaded classifier to obtain a vector of faces from the input gray image.

3.1.3 Fisherfaces recognizer

As it was mentioned in Section 3.1.2, OpenCV provides pre-trained Haar Cascade classifiers for various types of detection like face, eye, body and smile detection. However, it does not have models to recognize different kinds of emotions apart from smile detection. I figured out that emotion recognition is feasible in OpenCV using Fisher Faces model [35].

Fisherfaces approach is a face recognition method based on a concent of Eigenfaces algorithm. Eigenfaces method applies Principal Component Analysis (PCA)[1] to extract image features, or in other terms, Principal Components, maximizing the overall variance in data. PCA is often used to perform dimensionality reduction. However, the disadvantage of this approach is that it does not take into consideration class separability, i.e., it can throw away components with valuable discriminative information. [1]

The Fisherfaces method aims to improve Eigenfaces technique, thus, it uses Linear Discriminant Analysis (LDA) invented by R. A. Fisher in 1936[2]. LDA is optimized for class separability as it finds a linear combination of features that separates best between classes maximizing the ratio of between-classes to within-classes variance, instead of maximizing the total variance [1].

In OpenCV, Fisherfaces algorithm is implemented in Contrib module, so I have to complete some extra build steps mentioned in Section 3.1.1 to use Fisherfaces method. I found Fisherfaces recognition model trained according to the steps described in "Emotion Recognition With Python, OpenCV and a Face Dataset" article [35]. Listing 3.3 demonstrates how to initialize `FisherFaceRecognizer` class, load the model from the file and run the recognition process to get emotion prediction label.

Listing 3.3: Initialize and load Fisher Faces model

```
1 fisherFaceRecognizer = new cv.face_FisherFaceRecognizer();
2 fisherFaceRecognizer.read(emotionModelPath);
3 ...
4 prediction = fisherFaceRecognizer.predict_label(faceGray);
```

[1] PCA is a method used in exploratory data analysis and for making predictive models.

[2] In 1936, R. A. Fisher introduced the Iris flower data set as an example of discriminant analysis in the paper "The Use of Multiple Measurements in Taxonomic Problems".

3.1.4 Changing Colorspaces

Gray and **HSV** filters are based on changing image colorspace. Usually, image color is stored in RGB format (Red Green Blue) or RGBA (Red Green Blue Alpha), where Alpha channel is responsible for image transparency. Overall, OpenCV has more than 150 color-space conversion methods, but I will focus on (RGB → Gray) and (RGB → HSV) conversions. To convert image from one color space to another, I use `cv.cvtColor(...)` function where one of the function parameters specifies the code of conversion. For example, to set (RGB → Gray) conversion, I need `cv.COLOR_RGB2GRAY` conversion code. While RGB consists of three channels, Gray color space has only one channel. So OpenCV multiplies RGB values of a pixel by the following coefficients and sums them into one value:

$$Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B, \tag{3.2}$$

where Y is the final gray shade; R, G and B are Red, Green and Blue components of a pixel [52].

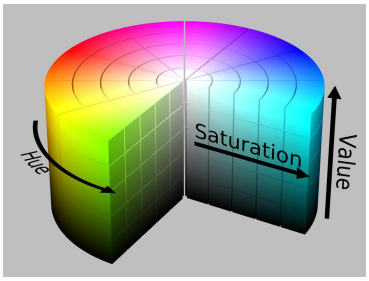


Figure 3.2: HSV color representation

HSV model is an alternative representation of the RGB format, where color channels stand for Hue, Saturation and Value (Figure 3.2). Hue attribute describes the color where each color takes a certain range. For example, in OpenCV.js, ranges of color shades are defined in the following way: 0-30 is blue, 30-60 is cyan, 60-90 is green, 90-120 is yellow, 120-150 is red and 150-180 is magenta. Saturation expresses the amount of gray in a particular color so if this component is around 0 the image has a faded effect, but if the value is max then the color is in purest (truest) version. The Value component refers to the brightness or intensity of the color, where 0 is completely black, and max value reveals the most color. For HSV filter the conversion code is `cv.COLOR_RGB2HSV`. Below are systems of equations for (RGB → HSV) conversion:

$$V' \leftarrow max(R, G, B)$$

$$S' \leftarrow \begin{cases} \frac{V' - min(R,G,B)}{V'} & if(V' \neq 0) \\ 0 & otherwise \end{cases}$$

$$H' \leftarrow \begin{cases} 60(G - B)/(V' - min(R, G, B)) & if(V' = R) \\ 120 + 60(B - R)/(V' - min(R, G, B)) & if(V' = G) \\ 240 + 60(R - G)/(V' - min(R, G, B)) & if(V' = B) \end{cases}$$

$$If\ H' < 0\ then\ H' \leftarrow H' + 360$$

$$On\ output : 0 \leq V' \leq 1, 0 \leq S' \leq 1, 0 \leq H' \leq 360,$$

where R, G and B are Red, Green and Blue components of a pixel; H', S' and V' are intermediate values of Hue, Saturation and Value, which are then converted to the destination data type:

$$V \leftarrow 255 \cdot V',\ S \leftarrow 255 \cdot S',\ H \leftarrow \frac{H'}{2}\ (to\ fit\ 0 - 255\ range), \tag{3.3}$$

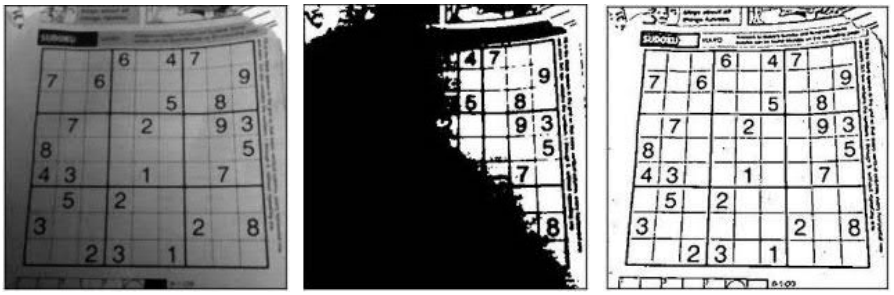
where H, S and V are final values of Hue, Saturation and Value [52].

3.1.5 Image Thresholding

Two filters applied in my application are related to image thresholding. The first one is **Binary Threshold** where destination pixel can get either white or black color depending on whether the color of source pixel is higher than the threshold value or lower. In the code, I call `cv.threshold (...)` function with `cv.THRESH_BINARY` thresholding type. In the UI, the threshold value is configurable through the range slider.

However, in some cases, such as images with varying illumination, this simple filter is not appropriate because the threshold value is set globally, i.e., it is a constant for the entire image. Figure 3.3a shows the image where some areas are lighter than others, and Figure 3.3b demonstrates what happens if I apply Binary Thresholding of 80. Part of the image is eliminated. The threshold method presented below can process this case better.

The second filter that I will consider is **Adaptive Threshold**. This algorithm fixes the problem explained above calculating threshold for small regions of an image so that I have different thresholds for different areas. In the WebCamera, I use `cv.adaptiveThreshold(...)` function with `cv.ADAPTIVE_THRESH_GAUSSIAN_C` option meaning that threshold value is calculated as the



(a) Image with varying illumination (b) Binary threshold applied (c) Adaptive threshold applied

Figure 3.3: Image Thresholding [52]

weighted sum of neighborhood values, where weights are a gaussian kernel. There is also `cv.ADAPTIVE_THRESH_MEAN_C` option in OpenCV where the threshold value is the mean of neighborhood region. In the demo, user can set block size of the filter, size of a pixel neighborhood that is used to calculate a threshold value for the pixel. Figure 3.3c presents adaptive thresholding in action.

3.1.6 Smoothing Images

Smoothing filters, also called blurring filters, belong to the category of low-pass filters, which attenuates high-frequency signals. In general, the blurring can be interpreted as calculating a pixel as the mean value of its neighborhood pixels. I will focus on three image blurring filters provided by OpenCV: **Gaussian Blur**, **Median Blur** and **Bilateral Blur**.

Gaussian filtering is based on Gaussian function for calculating the kernel, which is applied to each pixel of the image:

$$G(x,y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2}+\frac{-(y-\mu_y)^2}{2\sigma_y^2}}, \tag{3.4}$$

where G is Gaussian kernel function, coefficient A is the amplitude, μ_x is the average of x, μ_y is the average of y, σ is the standard deviation, x is the distance from the origin in the horizontal axis, y is the same in the vertical axis [52]. Steps to calculate Gaussian blur for a destination pixel are the following:

- Choose a central pixel and take a region of surrounding pixels
- Apply Gaussian kernel to each neighbouring pixel to get a square array of weights

- Multiply each neighbouring pixel by corresponding weight value
- Add up the values resulting from above multiplications
- Replace the value of central pixel with calculated sum

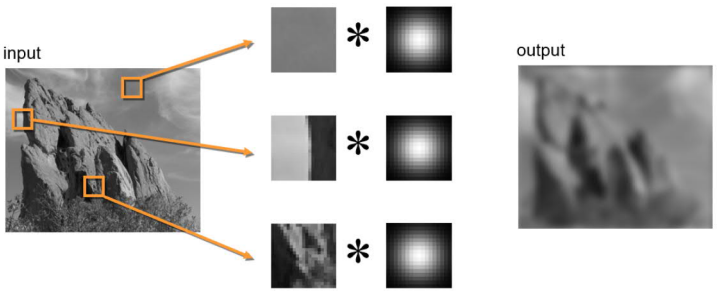
While the Gaussian blur algorithm calculates the average of the surrounding pixels, the Median blur filter just finds the median pixel from a region and replaces the central pixel with the median. However, a drawback of Gaussian and Median filters is that they smooth away the edges of an object. The Bilateral filter can fix this problem, at least at a certain extent. Calculation of Bilateral blurring is similar to Gaussian blurring, but in addition to weights, there is also intensity component, which is measured between the neighboring pixels and the evaluated one. If there is a big difference in the intensity of central pixel and its neighbors, then Bilateral filter does not apply blurring. Thus, this filter blurs only uniform areas and preserves edges of the object (see Figure 3.3a).

In OpenCV, I use `cv.GaussianBlur(...)`, `cv.medianBlur(...)` and `cv.bilateralFilter(...)` to run these filters. In the Instagram Filters demo, user is able to change the kernel size for Gaussian and Median Blur as well as diameter and sigma parameters for Bilateral blur using slider elements.

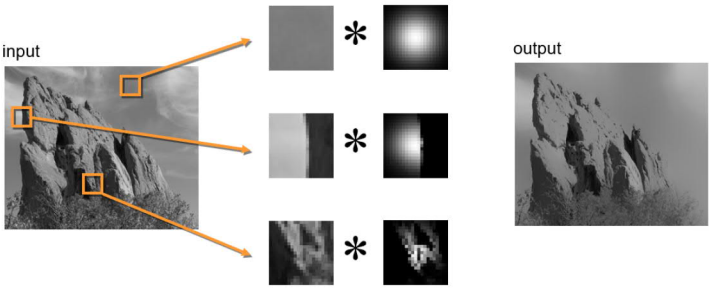
3.1.7 Morphological Transformations

Thresholding filters may bring numerous imperfections to the processed images. In particular, the binary regions of pixels can be omitted or distorted after applied threshold. Morphological operations aims to remove these imperfections according to the form and structure of the image. Basically, they are performed on binary images, i.e., images where each pixel has value of either 0 or 1 (black and white color, respectively). The most fundamental morphological operators are **Erosion** and **Dilation**, however, there are also compound operators like **Opening**, **Closing** and **Top Hat**. Morphological operators probe each pixel of the image using structuring element, which is defined as a matrix of pixels and describes a shape of transformation, i.e., neighbourhood region. Usually, the structuring element takes the shape of circle, square, diamond or cross, however, it can take any other shape as well.

Figure 3.5 depicts the result of some morphological transformations compared to the original image. While background pixels are displayed as white in this image, each pixel in the object is shown as black. Erosion operator is used for reducing the objects contained in the input image. The basic idea of this operator is that if the structuring element on an object pixel touches



(a) Result of applying Gaussian kernel



(b) Result of applying Bilateral kernel

Figure 3.4: Image Blurring [55]

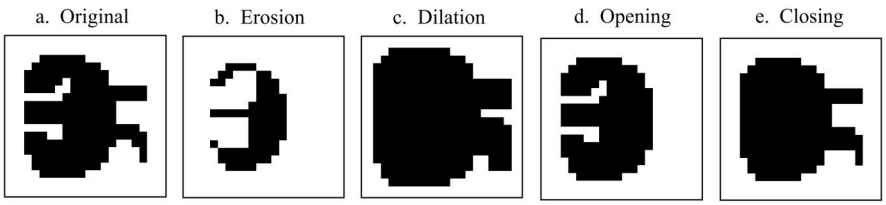


Figure 3.5: Morphological transformations [15]

a background pixel, it is changed into a background pixel. The Dilation is straight opposite process – it expands the shapes contained in the input image. In this transformation, a background pixel is assigned an object value if the structuring element on a background pixel touches an object pixel. Thus, the Erosion makes the object smaller and can break it sometimes into multiple object, while Dilation operator makes the object larger and can merge multiple objects into one. Opening operator is the Erosion followed by the Dilation. It can be used for noise removing. The Closing is reversed operation of the Opening, defined as Dilation followed by Erosion. This operation is suitable for closing small holes inside the object. Top Hat operation is the difference between input image and Opening of the image. [15]

`cv.morphologyEx(...)` function is responsible for morphology transfor-

mation in OpenCV. I can pass an argument with the code like `cv.MORPH_ERODE`, `cv.MORPH_DILATE`, `cv.MORPH_OPEN`, `cv.MORPH_CLOSE` or `cv.MORPH_TOPHAT` to specify the type of opearation.

3.1.8 Image Gradients

Image Gradients are called High-pass filters emphasizing regions of high spatial frequency. These filters are extensively applied in edge detection algorithms, for instance, in Canny Edge detection described in the next section. Image gradient techniques measure the change in intensity of a pixel in horizontal or vertical direction. The idea is to check each pixel and calculate the gradient vector, which points in the direction of largest possible intensity [52]. The length of this vector corresponds to the rate of intensity change in that direction. Thus, if pixels have large gradient, they may represent possible edges.

There are three types of Image Gradients in OpenCV called **Sobel**, **Scharr** and **Laplacian Derivatives**. Corresponding functions are `cv.Sobel(...)`, `cv.Scharr(...)` and `cv.Laplacian(...)`, respectively. Sobel operator combines Gausssian smoothing and differentiation operation. By default, it applies 3×3 convolution kernel either for x or y direction to each pixel in the image:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A\,;\; G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A\,, \tag{3.5}$$

where A is the source image, G_x and G_y are images, which at each pixel contain the vertical and horizontal derivative approximations, respectively and * operation denotes the 2-dimensional signal processing convolution [52]. It is possible to configure kernel size through function argument in OpenCV.

Scharr derivative is alternative representation of Sobel filter, however, it provides better result. Similarly, it uses two kernels, but they have different weghts compared to Sobel kernels.

While Sobel and Scharr operators are first-order derivatives, Laplacian filter calculates second-order x and y derivatives where each derivative is found using Sobel operator [52]:

$$\triangle src = \frac{\partial^2 src}{\partial x^2} + \frac{\partial^2 src}{\partial y^2} \tag{3.6}$$

3.1.9 Canny Edge Detection

Canny Edge algorithm developed by John F. Canny in 1986 [3] is widely used technique for edge detection tasks because of its accuracy and reliability. Canny Edge detector calculates the gradient magnitude of a pixel and classifies it as edge if the gradient is larger than compared to neighboring pixels. A typical implementation of Canny Edge algorithm include the following steps [52]:

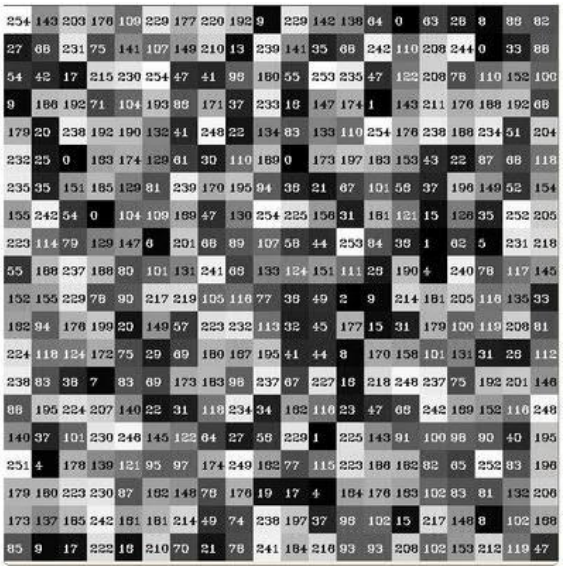
- Reduce the noise using Gaussian filter
- Apply Sobel operator to determine intensity gradient of the image
- Suppress the spurious response to edge detection to get rid of unwanted pixels
- Remove weak edges by applying hysteresis thresholding

Input arguments of `cv.Canny(...)` function in OpenCV allow to control the threshold values and Sobel kernel size. The result of the processing is black image with white edges.

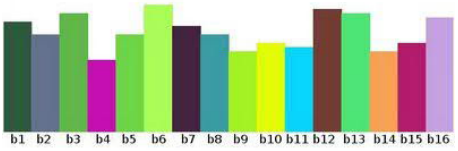
3.1.10 Histograms

In this section, I will focus on **Histogram Calculation** in OpenCV and filters based on this process such as **Histogram Equalization** and **Histogram Backprojection**. An image histogram is a graph or plot usually representing intensity distribution. Histogram Calculation process in OpenCV is based on collecting pixel data and organazing it into a set of bins. Thus, a histogram have predefined bins of pixel values on x-axis and corresponding number of pixels on y-axis [52]. Figure 3.6 shows an example of input image matrix where each pixel is assigned to its intensity value, and calculated histogram where each bin represents a number of pixels belonging to a certain range of intensity values. To calculate a histogram of intensity in OpenCV, I use `cv.calcHist(...)` function. I can define histogram size, i.e., number of bins, and range of intensity values as arguments of this function.

Some images may have background and foreground that are both bright or both dark. Therefore, the values on the histogram are confined to some specific range while the histogram of normal image has values from all regions. Histogram Equalization pursues the goal of improving the image contrast by modifying the image histogram [52]. The idea of this approach is to stretch large peaks in histogram across the whole range of values so that areas of lower local contrast gain a higher contrast (see Figure 3.7).



(a) Image matrix with intensity values



(b) Histogram representing intensity distribution

Figure 3.6: Histogram Calculation [52]

This image enhancement is actively used in medical and satellite images. `cv.equalizeHist(...)` function is called for Histogram Equalization in OpenCV.

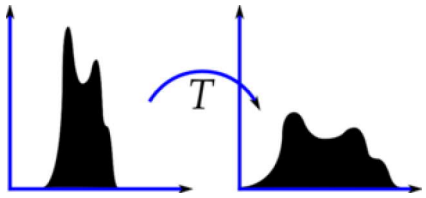


Figure 3.7: Histogram Equalization [52]

Histogram Backprojection method is suitable for object segmentation, i.e., finding objects of interest in the image. The algorithm takes an image histogram as input (color histogram is preferred over grayscale) and provides an image where each pixel corresponds to the probability of that pixel belonging to the object of interest [52]. To apply Backprojection method in OpenCV, I call `cv.calcBackProject(...)` function.

3.2 WebAssembly and Emscripten

The demands of web applications are gradually increasing, so it is necessary to expand the capabilities of the web platform. Until recently, JavaScript was the only language for development on the web. However, it is not well-equipped to meet some requirements of modern web applications in term of security and performance [6]. To make code exectution in browsers more safe and efficient, World Wide Web Consortium (W3C)[3] have implemented a new portable, low-level format called WebAssembly or Wasm [6]. As of 2019, Wasm is supported by all major web browsers such as Chrome, Edge, Safari, Firefox and Opera for both desktop and mobile devices [28].

Wasm standard defines a statically typed binary instruction set for a stack-based virtual machine [69]. It is designed as a compilation target for high-level languages like C, C++, Rust, C#, Go and others [69]. At the moment, it works with 4 data types: 32-bit/64-bit integers and 32-bit/64-bit floating-point numbers [69]. The initial core feature set of Wasm was based on asm.js (Assembly JS) [13]. asm.js is strict subset of JS created by Mozilla allowing browser engines execute it more efficiently by compiling it down to machine code in optimized way [13]. It is generally accepted that Wasm is the next evolutionary step of asm.js, so it is more fast and compact. However, Wasm does not replace JS but runs alongside JS [47]. Wasm modules can be loaded into JS app and executed at near-native speed [47].

As it is laborious to write Wasm code by hand, there is a bunch of tools to produce Wasm files from different programming languages. They are developed at a rapid pace, and many of them are still in early stages. Emscripten is the official, stable and most supported toolchain compiling C/C++ programs to asm.js and Wasm formats [22, 33]. Under the hood, Emscripten uses LLVM (Low-level Virtual Machine) toolchain with Clang frontend for C/C++ compilation and Binaryen compiler for asm2wasm conversion. Emscripten does even more than just compiling as it carries a lot of hidden work. For example, it emulates a local filesystem for `fopen()` calls, wraps OpenGL context with WebGL, provides memory management and Wasm-compatible implementation of C standard library [64]. Projects, that have already been ported to the Web using Emscripten, are ranging from application frameworks like Qt to complex graphical engines such as Unity and Unreal [33].

To port OpenCV source code written in C/C++ to OpenCV.js Wasm format, I install Emscripten compiler. See Listing 3.4 with a list of commands

[3]W3C is an international community where Member organizations, a full-time staff, and the public work together to develop Web standards.

showing how to install and activate Emscripten on Linux. On the first line, I clone Emscripten SDK project from the Github. On the second line, I navigate to the cloned repository. On the third and fourth lines, I install and activate the latest version of Emsdk package. Finally, I run `emsdk_env.sh` script to set Emscripten path to the PATH environment variable. These steps must be completed before executing OpenCV.js build script (see Listing 3.1).

Listing 3.4: Install and activate Emscripten

```
1 git clone https://github.com/emscripten-core/emsdk.git
2 cd emsdk
3 ./emsdk install latest
4 ./emsdk activate latest
5 source ./emsdk_env.sh
```

Recently, WebAssembly group have added SIMD and threads optimizations that significantly improve the performance. Wasm threads are accomplished with a JS primitive called SharedArrayBuffer [46]. It allows sharing an ArrayBuffer's contents concurrently between Web workers similarly as shared memory behaves on native platforms [31]. Thus, the code written in C or C++ that uses pthreads, or any other multi-threading library based on pthreads, can be compiled to Wasm and run in true threaded mode [31]. While threads allow more cores to work on the same data simultaneously, i.e., perform task parallelism, SIMD instructions are a special class of instructions that exploit data parallelism. SIMD performs the same operation on multiple data elements simultaneously. WebAssembly SIMD proposal introduces a new v128 value type and a number of operations that utilize this type [8, 67].

OpenCV community has enabled Wasm build with threads and SIMD options through `--threads` and `--simd` flags as it is shown on the third line of Listing 3.1. In this thesis, I use Chrome browser to present WebCamera application and measure performance of its use cases. To provide Chrome support for OpenCV.js with enabled threads and SIMD optimizations, I turned on `#enable-webassembly-threads` and `#enable-webassembly-simd` experimantal flags in `chrome://flags` (see Figure 3.8).

3.3 PWA

The gap between native and web apps has been bridged by the Google Web Fundamentals group with the help of emerging browser APIs and standards for Progressive Web Applications. PWAs are defined by a set of concepts

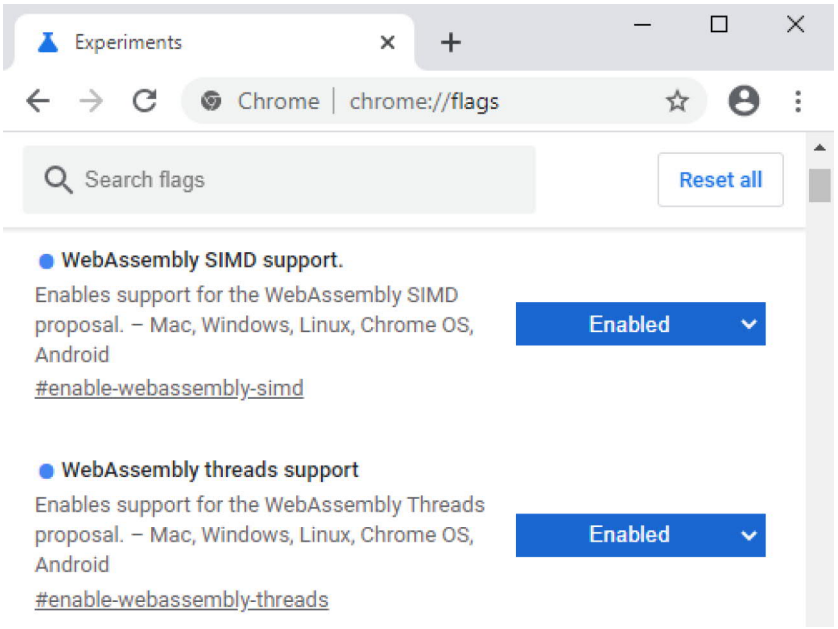


Figure 3.8: Chrome flags for WebAssembly SIMD and threads optimizations

and keywords such as progressive, responsive, connectivity independent, app-like, safe, installable and linkable. PWA term was first coined by A. Russel and F. Berriman in a blog post covering initial design ideas. Introduced PWA features include offline support, background synchronisation and home-screen installation. These PWA contributions unificate the mobile experience bringing native-like features to cross-platform web applications and allowing to work without Internet connection, receive push-notifications, install and distribute web apps without marketplaces. [2]

To migrate my website to the application with native-like capabilities, I integrated some core features of a PWA. The first feature is the app installation allowing user to add WebCamera icon to the home screen and have quick access to it. When a user gets access to the app for the first time through a mobile browser, the website shows a notification on the screen offering to add WebCamera to the home screen. I can also install the app through the three-dots menu of the browser tab. To control how my PWA is added to a user's home screen, I need to create the Manifest file (Listing 3.5) and add it to the main HTML page (Listing 3.6) [58].

The second feature delivers offline experience meaning that app content is cached and available regardless of Internet connection status. Service Worker (SW) is the technology responsible for that feature. It runs on a separate thread from the main JS code thus flexibly managing network requests. SW

loads necessary files after the first access to the app and retrieves data from the local cache for further usage. Complete SW can be generated with Workbox Command Line Interface (Workbox CLI) [58]. Listing 3.7 demonstrates how to install this tool, run it to create `workbox-config.js` configuration file and generate SW. Workbox configuration file, presented in Listing 3.8, describes the caching rules. Finally, I register generated SW in the main HTML page (see Listing 3.9).

Listing 3.5: Create Manifest file for PWA

```
{
  "name": "Web Camera",
  "short_name": "WebCam",
  "display": "minimal-ui",
  "start_url": "samples/",
  "theme_color": "#673ab6",
  "background_color": "#111111",
  "icons": [
    {
      "src": "data/photo_camera_192.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ]
}
```

Listing 3.6: Add Manifest file in the main HTML page

```
<link rel="manifest" href="../manifest.json" />
```

Listing 3.7: Workbox CLI commands to generate SW

```
npm install workbox-cli --global
workbox wizard
workbox generateSW workbox-config.js
```

Listing 3.8: Workbox configuration file

```
module.exports = {
  globDirectory: '.',
  globPatterns: [
    '**/*.{html,js,css,xml,woff2,webp,png}'
  ],
  swDest: 'sw.js',
  maximumFileSizeToCacheInBytes: 100 * 1024 * 1024,
};
```

While the features like installability and offline access are available for both mobile and desktop devices, the third PWA feature in my app applies

only to mobile devices and makes the app content responsive to any screen size. WebCamera adapts to the phone size keeping full-screen camera view and corresponding width of the control bar. To achieve this behavior, I need to add the meta tag inside the head tag in all HTML pages (see Listing 3.10). The meta tag lets take control over the viewport, i.e., the user's visible area of a web page.

Listing 3.9: Register SW in HTML file

```
<script>
    if ('serviceWorker' in navigator) {
      window.addEventListener('load', () => {
        navigator.serviceWorker.register('../sw.js')
          .then(function (registration) {
            console.log('Registration successful');
            console.log('Scope is:', registration.scope);
          })
          .catch(function (error) {
            console.log('SW registration failed, error:', error);
          });
      });
    }
  </script>
```

Listing 3.10: Meta tag for responsive app screen size

```
<meta name="viewport" content="width=device-width, user-scalable=no"/>
```

Chapter 4

Implementation

This chapter describes the implementation of the WebCamera app. The first section introduces application UI (User Interface), a high-level flowchart depicting a general use case and media capture initialization. Subsequent sections discuss the workflow of seven use cases in detail describing algorithms for the processing of one input frame. The use cases I am showcasing here are - Instagram Filters, Face Detection, Funny Hats, Card Scanning, Document Enhancement, Emotion Detection and Invisibility Cloak.

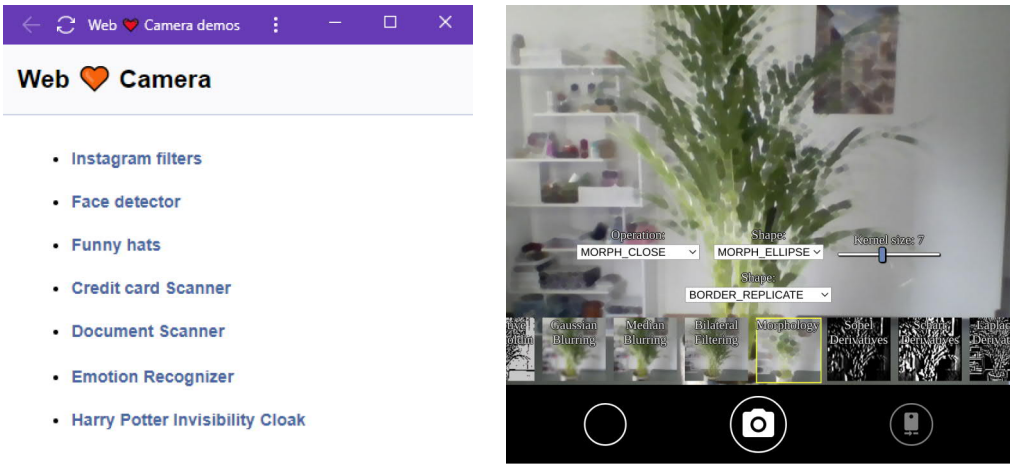
4.1 WebCamera implementation

WebCamera is a browser application containing only client-side code developed using HyperText Markup Language (HTML), JavaScript (JS) and Cascading Style Sheets (CSS). CV and image processing algorithms are computed on a CPU of a client with the help of OpenCV.js, and data, such as a captured face or credit card, does not leave a device. The app is deployed to GitHub Pages automatically from the master branch of WebCamera project [26], thus, a live demo is available through the link:
https://riju.github.io/WebCamera/samples/.

4.1.1 User interface

Since the main purpose of this application is to demonstrate the capabilities of OpenCV.js algorithms, the interface was not given much attention and it was made relatively simple. It uses pure Cascading Style Sheets (CSS) instead of web design frameworks to avoid additional dependencies. The main page of the app provides a list of links to HTML pages with OpenCV use cases (see Figure 4.1a). The interface of each use case affords camera

view and controls, which are described below.



(a) Main page of WebCamera app (b) Filter controls

Figure 4.1: User interface

The use case design is implemented according to the prototype of the native mobile camera applications, so it has such basic elements as a camera view and a camera control bar at the bottom (see Figure 4.1b). The HTML canvas element is used for drawing real-time camera view. On mobile devices, the camera canvas is stretched to the width of the device's screen while on desktops the canvas occupies fixed size: either 640×480 or 320×240 pixels. The control bar has a gallery preview, take-photo button and button for switching camera view between the face and environment mode. The buttons designed using Web Open Font Format (WOFF) of Google Material Icons. Work on the gallery is still in progress. At the moment, I display a thumbnail of the last photo in the place of the gallery preview, but I do not save the image to the device. This drawback is due to the lack of implementation in OpenCV.js allowing us to write an image to the device file system.

Some use cases have additional interface elements directly on top of the camera view. For example, Instagram Filters and Funny hats have a scrolling bar where user can select a filter or hat, respectively (Figure 4.1b and Figure 5.2). In addition, in the Instagram Filters demo, some filters provide sophisticated controls in the form of sliders, checkboxes or select tags. Any elements located on top of the image can be hidden simply by clicking on the free space of the camera canvas. In the same way, user can return these items back. Event listeners are assigned to each element of the interface so that they respond to user requests at any time.

Moreover, there are extra developer options, which are located under

the control bar and can be removed later so as not to clutter up the user interface. These options allow a developer to control the number of threads and enable performance statistics as well as configure other settings that will be described individually for some use cases in the next sections.

4.1.2 Use case workflow

Each use case of WebCamera application has a similar workflow from a high-level point of view. There are three main parts namely initialization, media capture loop and image processing algorithm. Figure 4.2 depicts a high-level flowchart including a sequence of basic steps needed to run a use case.

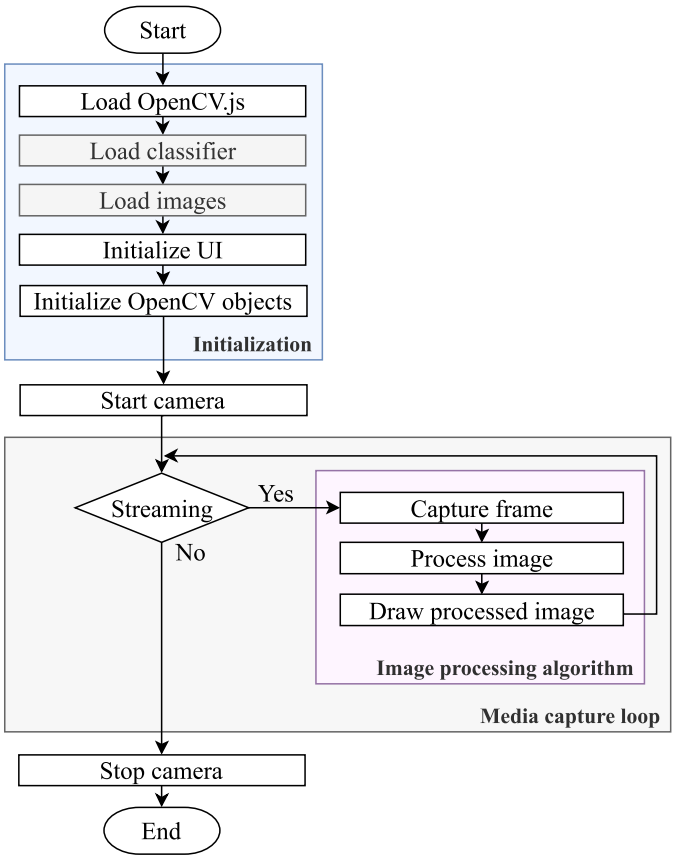


Figure 4.2: High-level flowchart of a use case

Initialization starts immediately after user opens a link from the main menu (Figure 4.1a). The first step of the initialization block is asynchronous loading of OpenCV.js script. Next two steps are necessary only for a few use cases. For example, I load Haar Cascade classifier to detect faces in

Face Detection, Funny Hats and Emotion Recognition demos. In addition to that, I need Haar Cascade model for the eyes detection in Face Detection and Funny Hats. Emotion recognition demo also requires Fisher Face classifier to recognize emotion from the detected face. The third step appears only in Funny Hats use case. It is needed to load images of hats and glasses before starting face processing. The fourth step is related to UI initialization and may vary from demo to demo depending on what interface elements are required. In the fifth step, I initialize OpenCV objects such as source and destination matrices for images, Haar Cascade classifiers and vector of faces.

Before the media processing loop, I set `streaming` variable to `true` and start the camera. Media capture initialization is described in the next subsection in more detail. While streaming, I capture input frames, process them and draw results. These three steps form an image processing algorithm which is different for each WebCamera demo. One iteration of the media processing loop processes one frame. I also calculate statistic numbers to track performance during the iteration of processing. By default, statistics are hidden and available in developer settings. If `streaming` variable is `false`, media processing loop is interrupted and camera stops.

4.1.3 Media capture initialization

To start the camera on a media device, I need to find available media capture hardware. For that purpose, there is `mediaDevices` interface from Web APIs providing access to hardware sources of media data like cameras and microphones. I will consider two important methods from this interface: `enumerateDevices()` and `getUserMedia()`.

`enumerateDevices()` method requests available media input and output devices. From the list of detected devices, I choose `'videoinput'` type of source and check if there are back and front cameras. Usually mobile phones have both cameras and label them as `'facing back'` and `'facing front'`, respectively. In out app, these cameras are needed to switch between "environment" and "user" facing mode. Listing 4.1 provides the code for obtaining back and front video input. If only one video source was found, I disable the facing mode button in the camera control bar.

`getUserMedia()` takes a video constraint defined by a user and checks permission to use requested media source. Then, it returns a `Promise` that provides a video stream. When stream is ready, event listener runs callback function. If matching media is not available or device denies permission, then the `Promise` throws `NotFoundError` or `NotAllowedError`, respectively (see Listing 4.2).

Listing 4.1: Find back and front camera sources

```
navigator.mediaDevices.enumerateDevices()
    .then(function (devices) {
      devices.forEach(device => {
        if (device.kind == 'videoinput') {

          if (device.facingMode == "environment"
            || device.label.indexOf("facing back") >= 0)
            controls.backCamera = device;

          else if (device.facingMode == "user"
            || device.label.indexOf("facing front") >= 0)
            controls.frontCamera = device;
        }
      });
      ...
    });
```

Listing 4.2: Get user media using mediaDevices interface

```
navigator.mediaDevices.
  getUserMedia({ video: videoConstraint, audio: false })
    .then(function (stream) {
       video.srcObject = stream;
       video.play();
       self.onCameraStartedCallback = callback;
       video.addEventListener('canplay', onVideoCanPlay, false);
    })
    .catch(function (err) {
       self.printError(
         'Camera Error: ' + err.name + ' ' + err.message);
    });
```

4.2 Instagram Filters

The demo outlined here emulates Instagram Filters but in a browser using OpenCV image processing algorithms. Changing colorspace, image thresholding, smoothing images, morphological transformations, image gradients, Canny edge detection and histograms are some of the filters that I have demonstrated in my web app. The underlying theory of these algorithms is presented in Section 3.1.

Flowchart of this use case is quite simple, it is shown in Figure 4.3. I apply the selected filter on the input frame and display the filtered image in the canvas. In addition, the demo has a small preview for each filter with a real-time image in the scrolling bar. To implement these previews, I resize

the original image using `cv.resize(...)` function, apply filters one by one and show them in small canvases of the scrolling bar.

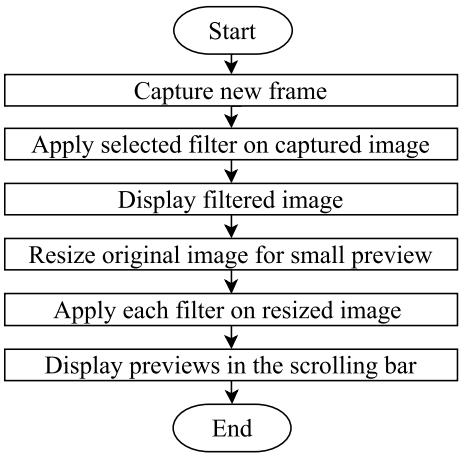


Figure 4.3: Flowchart of Instagram Filters

Mandatory arguments of a filter function are the source and destination matrices for input and output images. Usually filters also require extra parameters such as color conversion code, threshold value or kernel size. Example of calling Gray, Threshold and Median Blur filters is presented in Listing 4.3. `dstC1` and `dstC4` arguments correspond to destination matrix with 1 color channel and 4 color channels, respectively, i.e., to display gray pixel, I need only one value, however, if I want to show image in RGBA representation with transparency channel, then I store 4 values per each pixel.

Listing 4.3: Apply OpenCV filters

```
1 cv.cvtColor(src, dstC1, cv.COLOR_RGBA2GRAY);
2 cv.threshold(src, dstC4, thresholdValue, 200, cv.THRESH_BINARY);
3 cv.medianBlur(src, dstC4, kernelSize);
```

4.3 Face Detection

Face Detection use case demonstrates faces and eyes detection using Haar Cascade classifiers provided by OpenCV. See the idea of Haar algorithm in Section 3.1.2. The processing pipeline of the demo, presented in Figure 4.4, depicts the main steps of one frame processing. First of all, I convert input image to grayscale using `cv.cvtColor(...)` function because it is required in Haar Cascade approach. Then, I downsample image with

`cv.pyrDown(...)` function applying by default the following formula to the destination image size:

$$Size(\frac{src.cols + 1}{2^{dLevel}}, \frac{src.rows + 1}{2^{dLevel}}), \tag{4.1}$$

where *dLevel* is downsampling level (=1, by default), *src* is source image with *cols* as a width and *rows* as a height. I have added configurable developer setting in the Face Detection demo called "Downscale level" with values in the 0-4 range. As it was mentioned in Subsection 4.1.1, on desktop devices, video canvas occupies either 640×480 or 320×240 pixels. So if I consider an input image of 640×480 pixels and downsample it by the level equal to 1, then the result of this step is the image of 320×240 size, i.e., quarter of the original image.

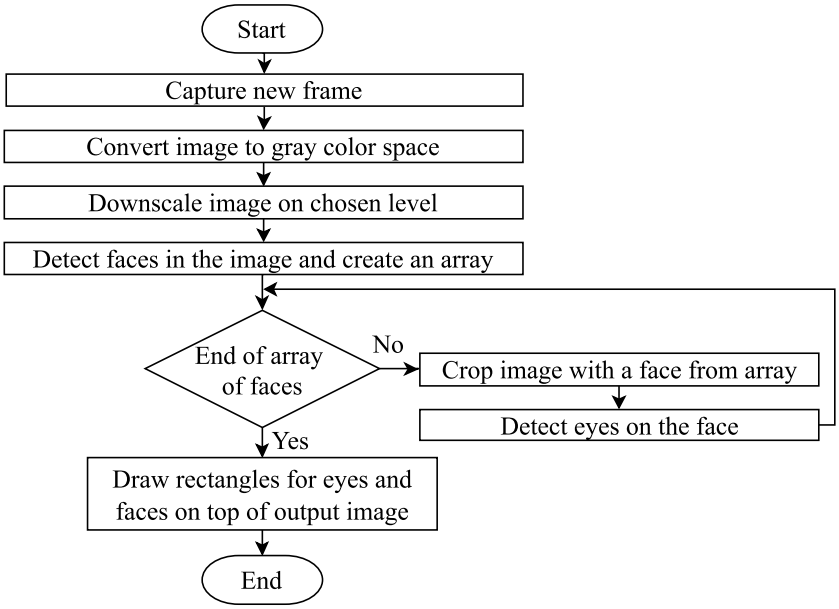


Figure 4.4: Flowchart of Face Detection

The most important part in Face Detection demo is `detectMultiScale(...)` function of `cv.CascadeClassifier()` object. Cascade classifier detects faces in the input image and returns a vector of faces, or in other words, coordinates of face rectangles. Iterating over the vector of faces, I crop a face with given coordinates using `src.roi(face)` function, where `src` is source image and `face` is face rectangle (four points). Then, I detect eyes in the same way as I have done it with faces. When the face vector is processed, I have coordinates for all faces and eyes. Finally, I draw rectangles for calculated coordinates and show the output image.

4.4 Funny Hats

Funny Hats requires face and eyes detection so it is based on the previous demo. As the name of the use case implies, Funny hats draws hats over detected faces. Moreover, since the demo has eyes detection, it draws glasses in addition to hats.

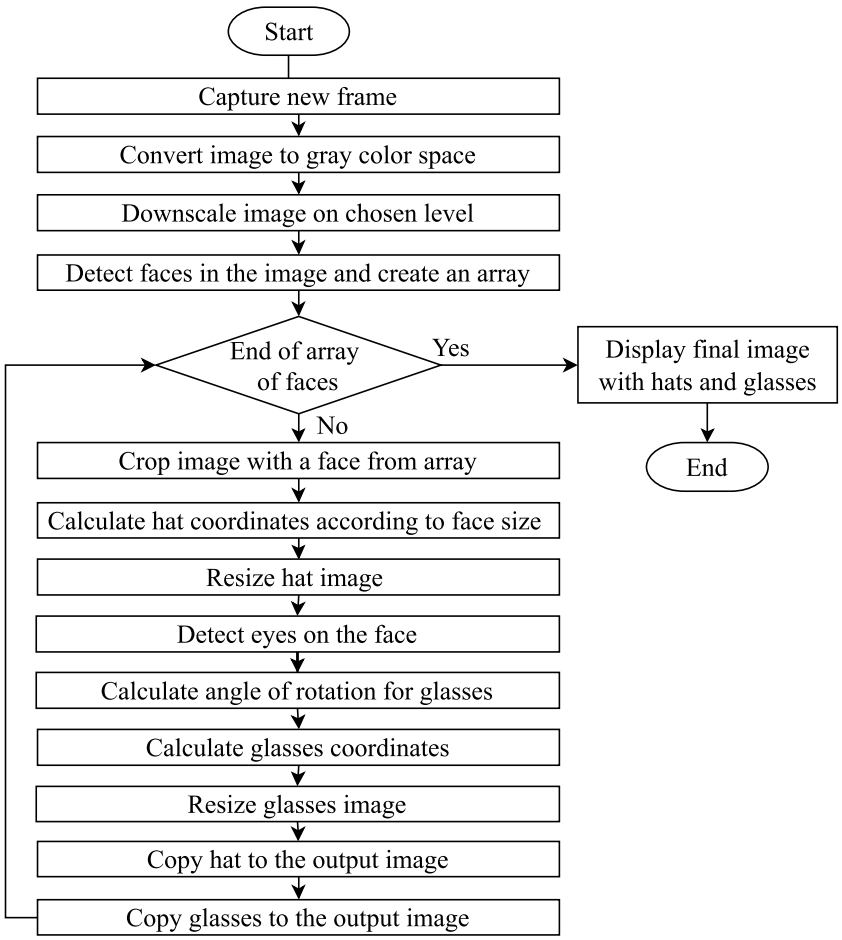


Figure 4.5: Flowchart of Funny Hats

Looking at the flow diagram of the process in Figure 4.5, we can see that sequence of steps is similar to Face Detection demo until I crop detected face from the main image. After that, I start processing template images of a hat and glasses. First, I calculate the hat size according to the face size and find absolute coordinates to overlay the hat later on the output image. Then I resize the image of the hat.

For the glasses, I have a slightly extended approach. First, I use Haar Cascades detects to detect the eyes. Second, I check if there are two eyes because sometimes the classifier can make a wrong prediction. Third, I find right and left eye to calculate angle of rotation using `cv.getRotationMatrix2D(...)` and `cv.warpAffine(...)` functions. Finally, I overlay resized images of hat and glasses on the output image using `src.copyTo(...)`.

To draw hats and glasses with transparent background, I need to create a background mask. In RGBA color representation, the Alpha channel is responsible for image transparency. Thus, I need to split hat image on channels and save the last channel separately to apply it when copy hat and glasses to the output image. Listing 4.4 shows how to read hat image, split it on channels and get the required channel by index.

Listing 4.4: Create transparent hat mask from Alpha channel

```
let rgbaVector = new cv.MatVector();
hatSrc = cv.imread(hatImage);
cv.split(hatSrc, rgbaVector);
object.mask = rgbaVector.get(3);
rgbaVector.delete();
```

4.5 Card Scanning

The WebCamera can also be used for another sort of tasks like credit card and document scanning. In this section, I will describe the implementation of the Card Scanning use case based on Optical Character Recognition (OCR). This demo involves using a template matching algorithm for the OCR-A font, which is commonly used on the front of credit cards. The demo applies to a credit card with a sixteen-digit number divided into four groups as shown in Figure 4.8a. The basic steps of the use case are the following:

- Find the card contour in the image
- Localize 4 groups of digits
- Detect digit contours in each group
- Apply template matching to recognize the digits

A more detailed flow diagram of my approach is presented in Figure 4.6. This diagram describes the processing of one input frame. Before I start processing, I load reference OCR-A digits. The digits are shown in Figure 4.7.

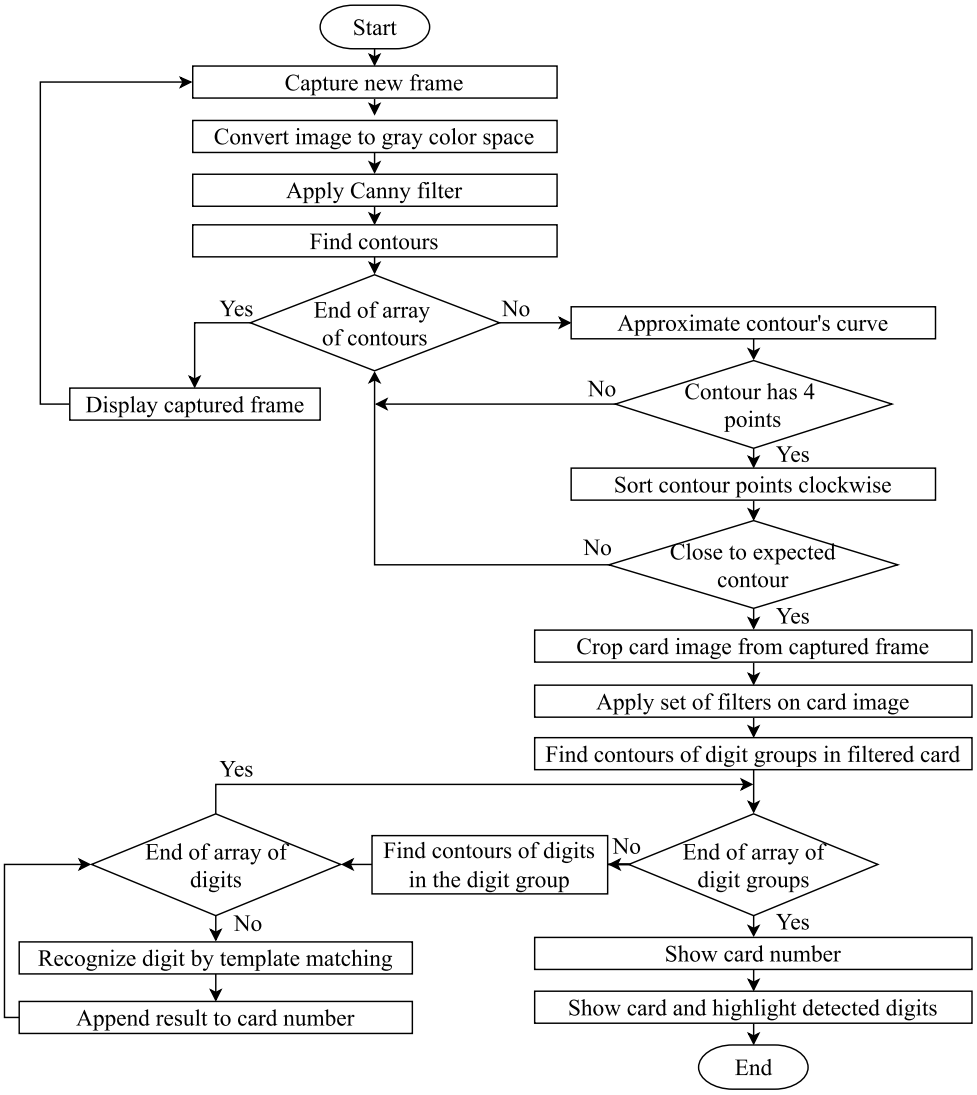


Figure 4.6: Flowchart of Card Scanning

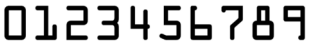
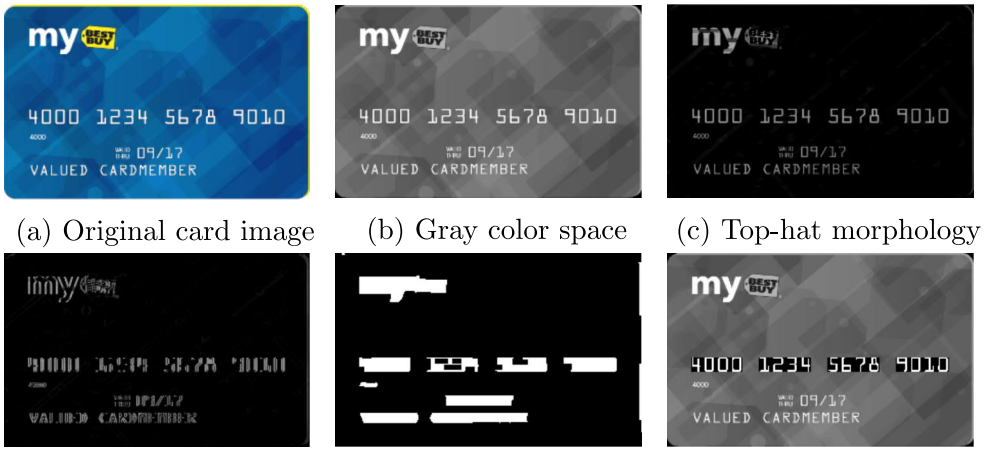


Figure 4.7: Reference OCR-A digits in Card Scanning demo

Later I will use them in template matching. Let me now explain each step of the flowchart.

First, to detect edges of the card, I need to convert image to grayscale, apply the Canny edge filter and find all contours in the edged image. See Section 3.1 to read descriptions of image processing algorithms in OpenCV

applied in this demo. Next, I run a loop iterating over detected contours and approximate number of points. If the approximated curve has four points, I sort these points clockwise and check if they are close to the expected contour. The expected contour is a green rectangle on the screen and user should adjust the card edges as close as possible to the rectangle border. If none of the curves fits the specified area, the loop is completed and the algorithm requests the next input image. If the required contour was found, the part of card edge detection is done.



(a) Original card image (b) Gray color space (c) Top-hat morphology

(d) Sobel gradient (e) Closing operator (f) Number recognized

Figure 4.8: Step by step card filtering

The next part of the Card Scanning approach is detection of 4 digit groups in the card. First, I crop the card from the image and resize it to 300 pixels wide. Next, I apply a set of filters to the card in the following sequence: conversion to gray color space, "Top-hat" morphological transformation, Sobel gradient along the x-direction, "Close" morphological transformation, threshold operation as well as one more "Close" morphological operation. Results of filtering are presented in Figure 4.8. The Top-hat transformation finds light areas against a dark background, Sobel filter shows vertical changes in the gradient, thresholding is used to binarize the image, and "Close" morphological transformation closes small gaps between light regions. Then, in the filtered image, I find contours of 4 digit groups.

The last part of card scanning approach is focusing on the array of digit groups. In each group, I detect contours of four digits and loop over them. Next, I apply correlation-based template matching to the digit and choose the result with the highest score. In this step, I used `cv.matchTemplate(...)` function which compares the digit with the templates and calculates the similarity score corresponding to each reference digit. Then, I append recognized

digit to the card number array and determine the type of card by the first digit of the number where three is American Express, four is Visa, five is MasterCard and six is Discover Card. Finally, I display the card number and its type as well as card image with highlighted digit groups.

4.6 Document Enhancement

This section introduces another scanning approach called Document Enhancement. This use case applies a 2D perspective transform and threshold filter to the document image thus producing an enhanced document. I will discuss the processing pipeline of the demo presented in Figure 4.9.

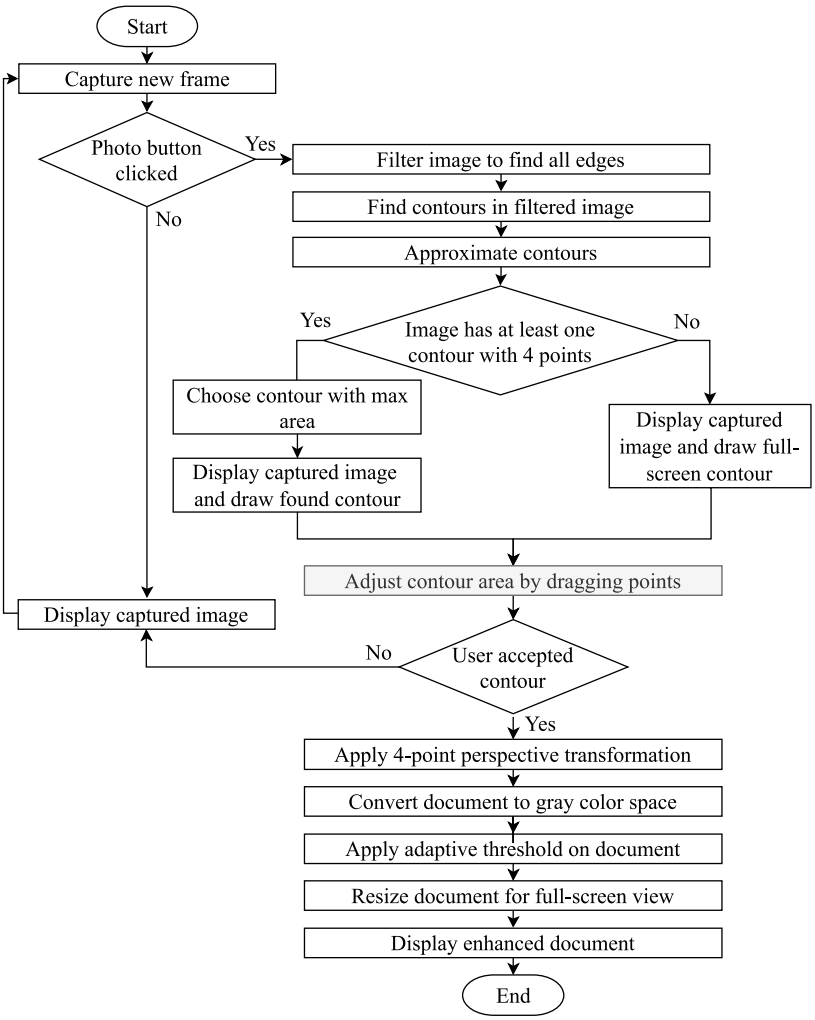


Figure 4.9: Flowchart of Document Enhancement

The camera view just shows the video stream until the user clicks on the button to take a photo. When the button was clicked, I apply the same approach as in Section 4.5 to detect edges of the document: filter image, find contours and approximate them. If the image has at least one contour with four points, I choose a contour with the maximum area and draw this contour as a green rectangle surrounding the detected document. Otherwise, I display full-screen green rectangle. If user is not satisfied with the selected area, he can adjust the area by dragging the points. This step is optional.

When user accepted the contour by clicking the button, I move to the main processing part. First, I apply four-point perspective transformation using `cv.getPerspectiveTransform` and `cv.warpPerspective` functions. It allows obtaining a top-down view of the document. Second, I convert the document to gray color space and apply the adaptive threshold. After this step, I have the scanned document with a black-and-white paper effect. Finally, I resize the document to full-screen size and display it.

4.7 Emotion Recognition

OpenCV has a few face recognition classes that can perform emotion prediction task. The one of these classes is `FisherFaceRecognizer`, which is described in more details in Section 3.1.3. To perform emotion recognition, first of all, I have built OpenCV with Contrib module (see Section 3.1.1). Then, I found a Fisher Face model that was trained on the following set of emotions: ['neutral', 'anger', 'disgust', 'fear', 'happiness', 'sadness', 'surprise'] [30, 35]. In the end, I draw an emoticon image instead of the face with a corresponding emotion. Let's discuss the flow chart of my Emotion Recognition use case (see Figure 4.10).

First, I convert an input frame to gray color space and detect faces in this image using Haar Cascade model and `detectMultiScale(...)` function (see Section 3.1.2). The result of this step is an array of detected faces. Then, I start looping over these faces. In the loop, I crop current face from the input image and resize it to 350×350 pixels as the Fisher Face recognizer is trained on images of this size. Then, I run the recognizer to make predictions and choose emotion with the highest score. The next step is to resize the corresponding emoticon image according to the face size and copy it to the input frame. In addition, I copy a small original frame to the left top corner of the camera view. Finally, I draw the output frame where user sees the emoticon image instead of his face and small original frame.

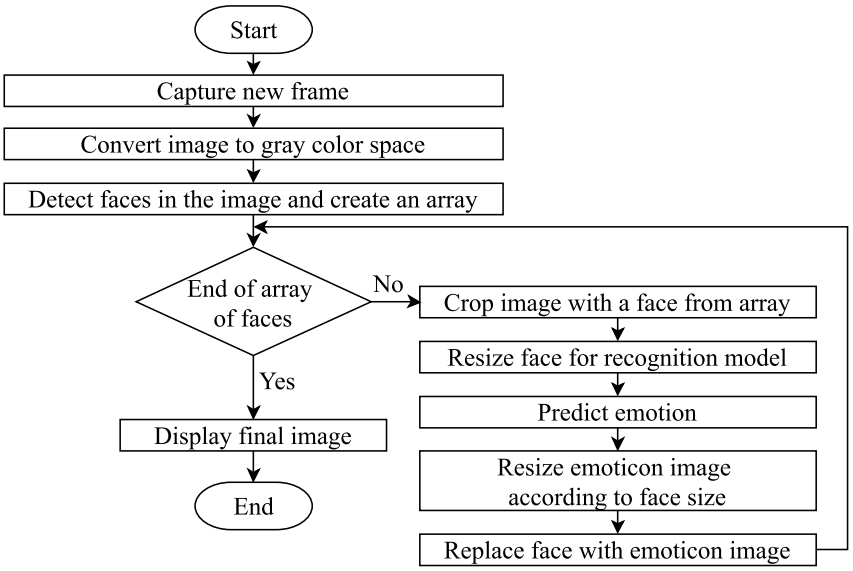


Figure 4.10: Flowchart of Emotion Recognition

4.8 Invisibility Cloak

In this section, I discuss how to create such a magical effect as Harry Potter Invisibility Cloak using simple CV techniques in OpenCV. In technical terms, this magical experience is called color detection and segmentation. The basic idea is to show the background frame instead of things with the specified color. By default, I remove blue color, but this parameter is configurable.

The algorithm starts with capturing the background frame and storing it for future use. To do this, I run a short loop to wait until the camera settings stabilize otherwise background image can have dark shade compared to the next input frames. Then, I start the main loop and process each input frame according to steps in Figure 4.11. The first step is to convert the image from RGB to HSV color space because in HSV representation it is easier to control color shades. HSV is described in Section 3.1.4. Hue component of HSV format defines the color (see Figure 3.2). So in OpenCV, if the object is blue, Hue value will be in the range of 0-30. Saturation and brightness (intensity) can be adjusted by Saturation and Value components of HSV format.

The second step is to apply `cv.inRange(...)` filter where I specify lower and upper boundaries of the color that I want to extract from the image. Since I need to detect the blue color and its shades, I define the Hue range as 0-30, Saturation as 100-255 and Value (intensity) as 7-255 by default. How-

ever, user is able to set own HSV parameters in the demo (see Figure 4.12).

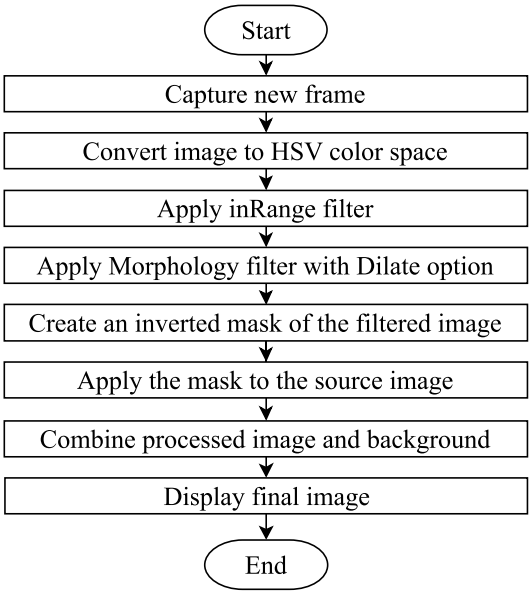


Figure 4.11: Flowchart of Invisibility Cloak

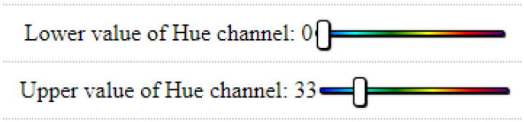


Figure 4.12: Controls for color segmentation

The third step is morphology transformation `cv.morphologyEx(...)` with Dilate option `cv.MORPH_DILATE`. This option increases the area of the object and joins broken parts (see Section 3.1.7 about morphology operators). Next, I create an inverted mask of the filtered object and apply it to the source image. It provides the original image, but now the blue color is segmented out. Finally, I need to add the background in the empty area so I use `cv.addWeighted(...)` function to combine source and background to the destination image.

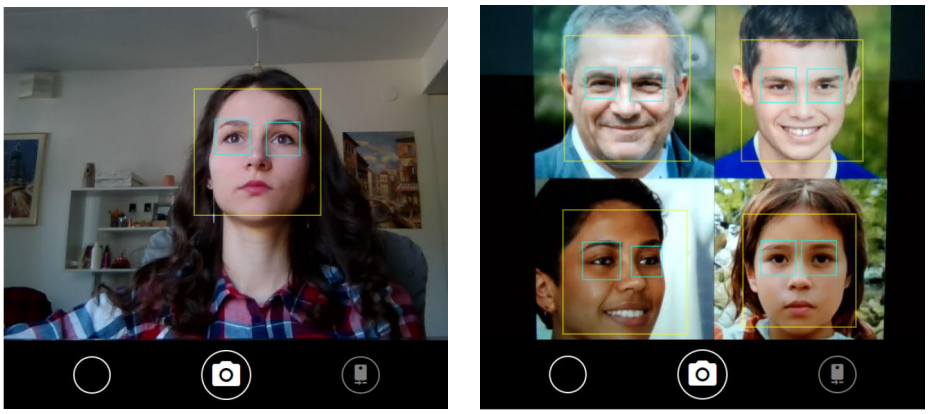
Chapter 5

Results

This chapter showcases WebCamera use cases in a browser and presents performance statistics.

5.1 Demos in a browser

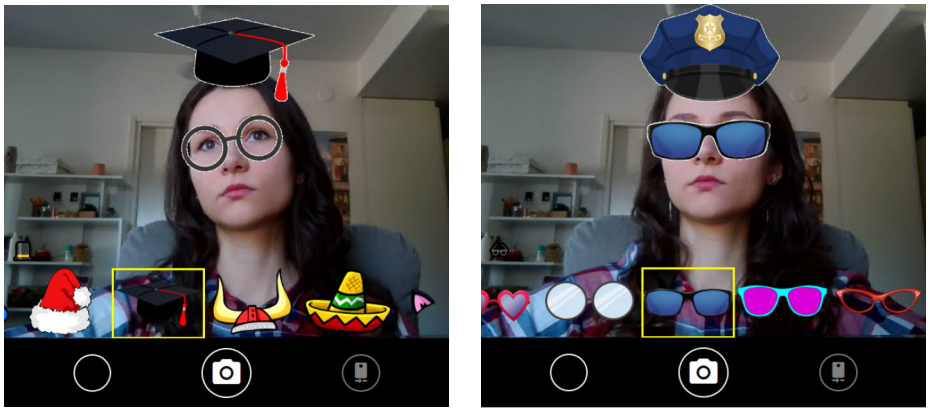
Face Detection demo is shown in Figure 5.1. It demonstrates the ability to detect one or multiple faces in one image. In addition to face detection, eyes detection is enabled.



(a) One face (b) Multiple faces

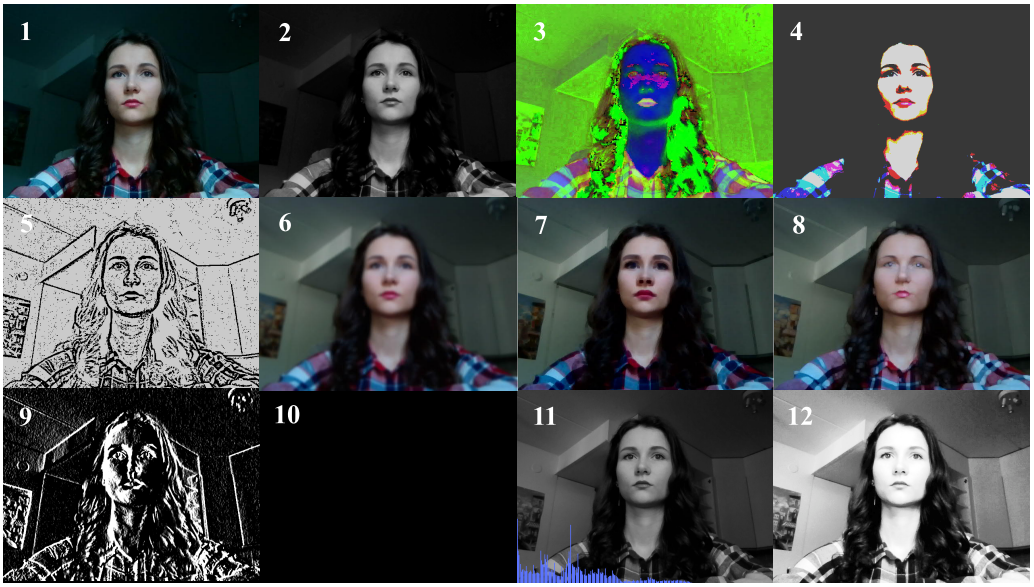
Figure 5.1: Face Detection in a browser

Two examples of Funny Hats demo are showcased in Figure 5.2. User can select templates for hats and glasses from the scrolling bars. There is also a small button on the left side to switch between hats and glasses bars.



(a) Choose hat (b) Funny Hats

Figure 5.2: Funny Hats in a browser



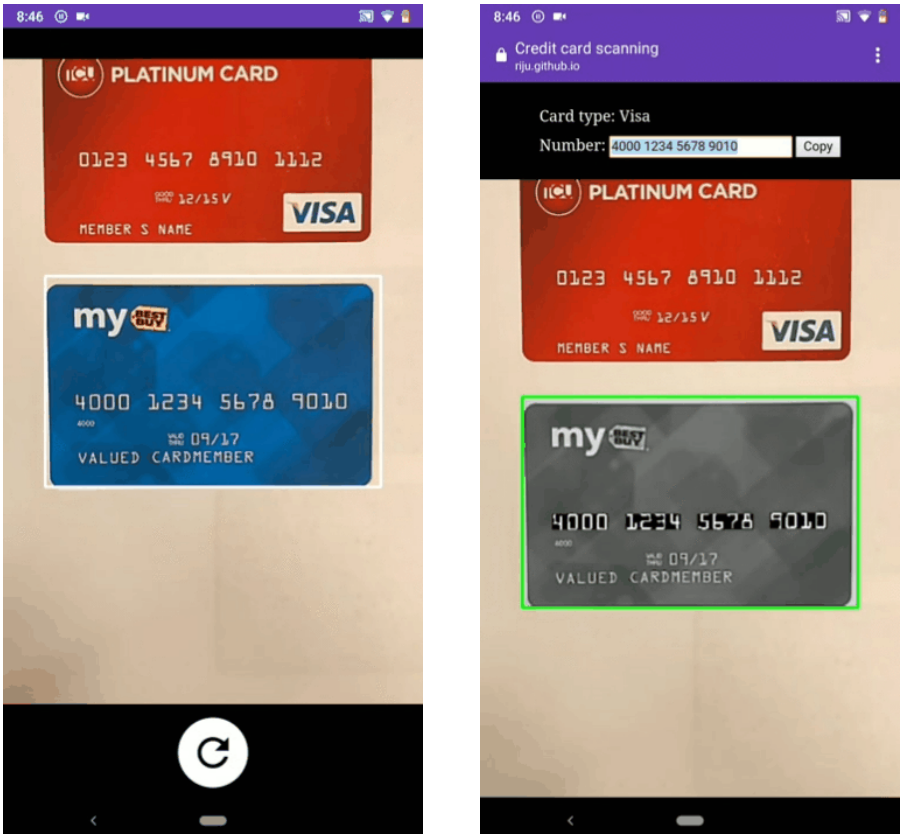
1 - No filter, 2 - Gray, 3 - HSV, 4 - Threshold, 5 - Adaptive threshold, 6 - Gaussian blur, 7 - Morphology (erode operation), 8 - Morphology (dilate operation), 9 - Sobel derivative, 10 - Canny edge, 11 - Histogram calculation, 12 - Histogram equalization.

Figure 5.3: Instagram Filters in a browser

Figure 5.3 presents a collection of screenshots from Instagram filters use case. The figure contains some examples of applied filters while the full set of filters contains 15 options: Gray, HSV, Threshold, Adaptive threshold, Gaussian Blurring, Median Blurring, Bilateral Blurring, Morphology, Sobel

Derivatives, Scharr Derivatives, Laplacian Derivatives, Canny Edge Detection, Calculation, Equalization and Backprojection. The UI has a scrolling bar with previews of filters where user can click on a preview to apply the selected filter on the main image (see Figure 4.1b).

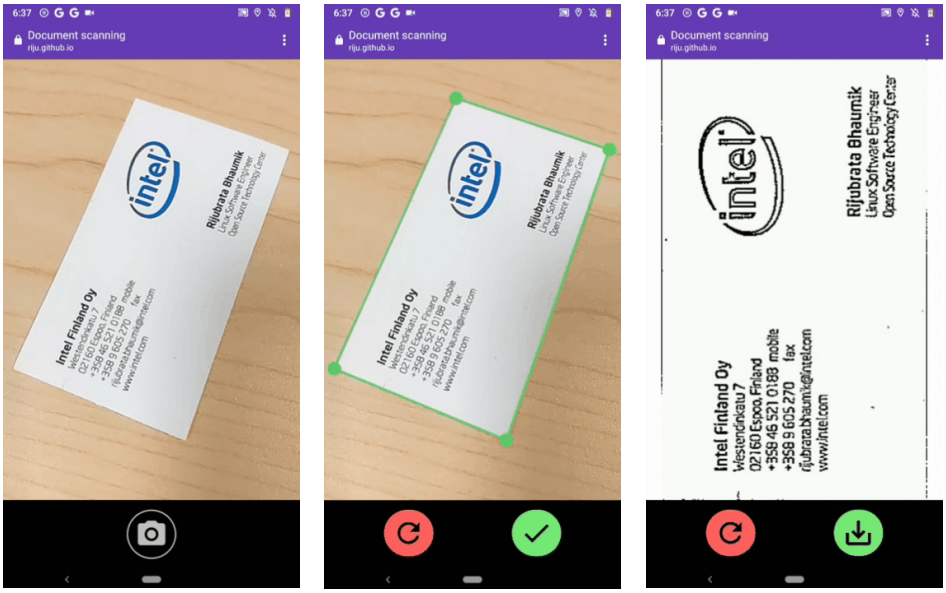
Card Scanning process is shown in Figure 5.4. In the first image, we can see a white rectangle where user has to put a credit card. When card edges were detected, the demo starts processing card number. The second image shows the result of scanning. User gets the card type, card number and card image with highlighted digit groups. User can copy the card number by clicking the Copy button on top of the screen.



(a) Detect card edges (b) Card number recognized

Figure 5.4: Card Scanning in a browser

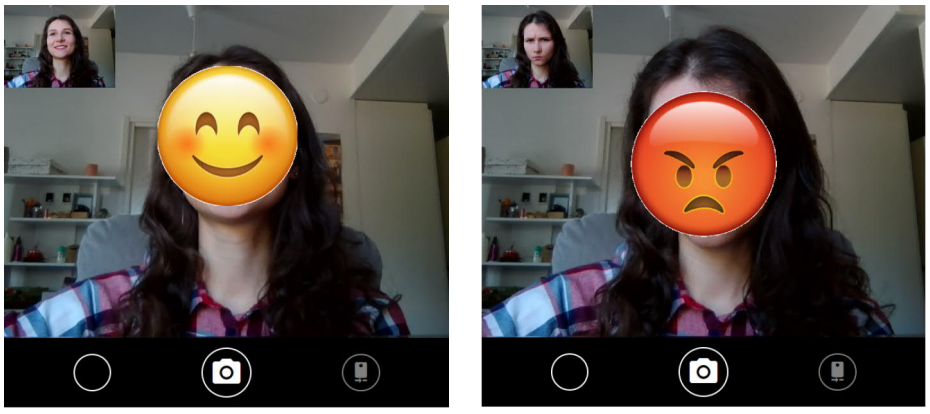
Document Enhancement approach is presented Figure 5.5. The first step is to click the "Take photo" button. The second step is to accept or adjust the detected document. Finally, we see the enhanced document with applied perspective transformation and threshold filter.



(a) Start scanning (b) Edges detected (c) Processing done

Figure 5.5: Document Enhancement in a browser

Figure 5.6 demonstrates two examples of Emotion Recognition - Happiness and Anger. Other available emotions are Neutral, Disgust, Fear, Sadness and Surprise.



(a) Happiness (b) Anger

Figure 5.6: Emotion Recognition in a browser

Finally, Harry Potter Invisibility Cloak is shown in Figure 5.7. In this demo, I segment out the blue color by default, however, user can choose any color he needs.

(a) Without Invisibility Cloak (b) Invisibility Cloak in action

Figure 5.7: Invisibility Cloak in a browser

5.2 Performance statistics

I tested WebCamera app in the Chrome browser on Windows laptop and Android phone. Mobile phone model is Motorolla G8 with Octa-core CPU that has eight logical cores: 4×2.0 GHz Kryo 260 Gold and 4×1.8 GHz Kryo 260 Silver. The laptop has the following CPU characteristics:

CPU characteristics	
Model	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
Base speed	2,21 GHz
Sockets	1
Cores	6
Logical processors	12
Virtualization	enabled
L1 cache	384 KB
L2 caches	1,5 MB
L3 cache	9,0 MB

I performed two experiments for measuring performance. The first experiment is to compare OpenCV.js built with various optimization options on both the laptop and mobile phone. To do this, I have built a Wasm file of OpenCV in four different ways: not optimized, with SIMD optimization, with Threads optimization and with Threads+SIMD optimizations. In this experiment, I set a maximum number of threads when Threads optimized Wasm was used. The browser limits performance to 60 FPS as a maximum.

Table 5.1 shows laptop's FPS (Frames Per Seconds) numbers for Face Detection, Funny Hats and some filters from Instagram Filters use case.

Wasm type	Not optimized	SIMD	Threads (=12)	Threads (=12) + SIMD
Face detection	13	13	37	37
Funny hats demo	14	14	49	49
Gray filter	60	60	60	60
HSV filter	60	60	60	60
Threshold filter	60	60	60	60
Adaptive threshold	25	54	28	56
Gaussian blur	5	22	16	44
Median blur	12	22	13	22
Bilateral blur	14	15	36	42
Morphological transformation	29	60	60	60
Sobel derivative	49	60	60	60
Scharr derivative	56	60	60	60
Laplacian derivative	39	54	47	56
Canny edge	49	49	60	60

Table 5.1: Performance of OpenCV.js with different optimization options on the laptop, FPS

Table 5.2 presents the same FPS numbers measured on the mobile phone. There is no SIMD measurements in this table as there was a problem with OpenCV.js initialization on the mobile phone. The error in Chrome browser indicates "`expected type i32, found s128.load128`" meaning that SIMD is not supported yet in Chrome for Android. However, it will be available soon as it is promised on Chrome status page [29]. While in Face Detection and Funny Hats measurements for the laptop I used first Downscale level, for the mobile phone I applied second Downscale level as it still detects eyes and faces but shows higher performance. See Section 4.3 and Section 6.1 to find more about Downscale level control. Other configurations for filters like threshold values and kernel sizes are kept the same on both the laptop and mobile phone.

In the second approach, I focus on Threads version of OpenCV.js and test performance of Face Detection, Funny Hats and Emotion Recognition demos running with a different number of threads on the laptop (see Figure 5.8).

Wasm type	Not optimized	Threads (=8)
Face detection	6	11
Funny hats demo	4	6
Gray filter	11	15
HSV filter	9	14
Threshold filter	10	15
Adaptive threshold	2	2
Gaussian blur	0	2
Median blur	1	5
Bilateral blur	1	5
Morphological transformation	4	5
Sobel derivative	6	9
Scharr derivative	9	11
Laplacian derivative	4	7
Canny edge	6	11

Table 5.2: Performance of OpenCV.js with different optimization options on the mobile phone, FPS

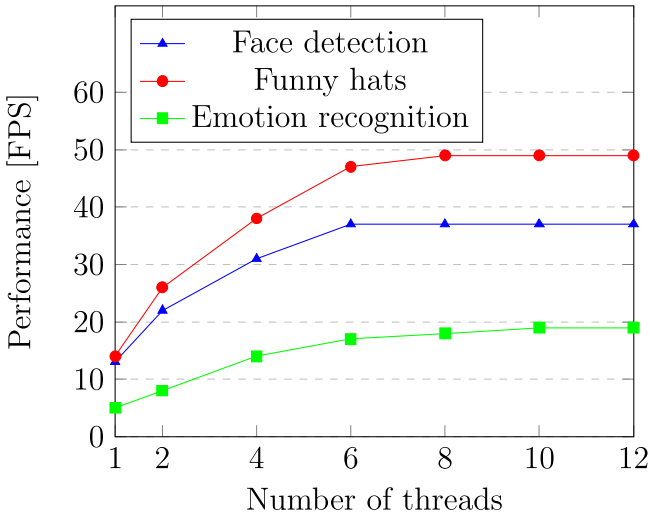


Figure 5.8: Performance of OpenCV.js on the laptop running with different number of threads

Chapter 6

Discussion

As this thesis aims to evaluate the user-perceived quality of the developed application, the first section is dedicated to the analysis of the performance statistics. There, I compare FPS numbers and discuss the effectiveness of optimization options such as SIMD and threads flags for Emscripten compiler. The second section reveals supporting research question about the diversity of OpenCV algorithms to implement various CV and image processing tasks. The third section considers the PWA experience and its native-like features. The fourth section discusses some limitations occured in the app usage. The last section suggests future work and possible improvements for the WebCamera including a short report about an attempt to port Intel's Model Optimizer and Inference Engine to the Web platform.

6.1 Analysis of performance statistics

Table 5.1 and Table 5.2 present performance statistics for the laptop and mobile phone, respectively, based on applied optimizations in OpenCV.js. I tested four versions of OpenCV.js, such as not optimized, SIMD, threads and threads+SIMD, for some use cases of WebCamera application. It is obvious that not optimized OpenCV.js should be slower than other versions and threads+SIMD configuration should show the highest numbers. Indeed, performance results confirm this. For example, for Gaussian filter on the laptop, the threads+SIMD version is almost 9x faster than not optimized OpenCV.js. On average, it shows 3x greater performance for all use cases presented in the first table. Some filters may have even higher FPS values, but Chrome browser limits performance to 60 FPS.

Despite the fact that threads+SIMD configuration provides an excellent increase in speed, and in most cases, FPS numbers for desktop and laptop

users can be as high as possible or near the maximum value, mobile performance is still not acceptable to perform face detection or apply filters in real-time video. From my experience, an FPS value less than 15 is unpleasant for visual perception. In my demos, even with threads optimized version, I get FPS numbers equal to or less than 15. Other mobile processors may compute a little faster, but it is still unsatisfactory quality to be the perfect replacement for native mobile applications.

Moving from general analysis to a more detailed one, I will discuss also SIMD and threads flags applied separately and compare them with the not optimized version. In the first table, SIMD optimization does not affect some use cases. For instance, FPS numbers remain the same in Face detection, Funny hats and Canny edge filter. It means that these algorithms do not utilize v128 intrinsics and Emscripten compiler has not found any place for optimization when porting OpenCV to Wasm. However, this can be fixed by OpenCV developers later. Since SIMD optimizations for Wasm were integrated into OpenCV in 2019, this functionality is still in progress. To get back to laptop performance statistics, SIMD provides around 2x speedup for the Adaptive threshold, Median blur and Morphological transformation. Gaussian blur works even 4.5x faster. Edge detection filters like Sobel and Sharr derivatives are improved up to the maximum FPS value.

In contrast to SIMD, threads optimization affects all use cases. In the first table, it demonstrates 3x speedup for Face detection, 3.5x – for Funny hats, 3x – for Gaussian filter and 2.5x – for Bilateral filter. Similar to SIMD, threads increase performance up to the maximum FPS value for Sobel and Sharr derivatives. In addition, performance of the Canny edge is also improved up to 60 FPS. In other cases of the first table, a slight increase in speed is observed. Specifically, Laplacian derivative has been improved from 39 to 54 FPS, Adaptive threshold – from 25 to 28 FPS and Median blur – from 12 to 13 FPS. At the same time, experiments on the mobile phone presented in the second table show the average increase in performance by 1.5-2 times except the Adaptive threshold where no improvement is observed. From both tables, it is noticeable that blur filters, in particular, Gaussian, Median and Bilateral smoothing, are the most compute-intensive as their performance is worse than in other filters.

Figure 5.8 depicts another kind of statistics showing the dependence of performance on the number of running threads in Emotion recognition, Face detection and Funny hats demos. Single-threaded Emotion recognition takes 5 FPS, Face detection – 13 FPS and Funny hats – 14 FPS. The maximum achieved performance in this demos running with twelve threads is 19, 37 and 49 FPS, respectively. I would say that six threads is optimal as it allows to reach almost maximum possible FPS value. Emotion recognition demo

is twice as slow as the other two demos because it uses two models: Haar Cascades to detect face and Fisher faces to recognize an emotion that is quite a compute-intensive task.

One thing I want to notice in the Table 5.1 and Figure 5.8 is that Funny hats demo is faster than the Face detection. While the maximum rate for Face detection is 37 FPS, Funny hats processes even 49 FPS. The obvious question is how is it possible if Funny hats use case includes face detection and additionally hat processing, while in the Face detection I just draw a face rectangle over the input image. The point is that I redraw hat image only when it changes its position by more than 3 pixels. Otherwise, the hat image stays unchanged. It was done to remove the hat jitter effect. The reason of jitter comes from Haar Cascade model. This model gives a face with a new size every frame, and when I resize a hat according to the new face it looks like the hat is shaking. It turned out that in addition to jitter fix, it also improves performance. The parameter to control the room for jitter is configurable. Thus, I can set it more or less than 3 pixels, however, in my opinion, 3 pixels is optimal. With the jitter limit of one pixel, Funny hats takes the same performance as Face detection.

Some demos like Card scanning, Document enhancement and Invisibility cloak are not presented in the performance statistics. These use cases do not perform heavy workload, and thus run at maximum FPS on the laptop. For example, the Document enhancement shows just input image in real-time until user clicks "Take photo" button. Card scanning demo basically computes Canny edge filter until card edges are detected. Finally, the Invisibility cloak combines simple color space conversion and Morphology transformation. However, on the mobile phone, these demos still run in the FPS range presented in the Table 5.2, i.e., near 20 FPS in the Document enhancement and around 10 FPS in the Card scanning and Invisibility cloak.

It is worth to mention that discussed performance numbers are relative. Even, if the demos are processed on the same CPU, performance results can vary slightly depending on the amount of light in the room, the number of objects and faces in front of the screen, how far the face from the screen, etc. However, measured performance statistics still show an approximate picture of the processing in my application.

6.2 WebCamera limitations

When it comes to the topic of limitations, I would like to mention some drawbacks related to browser support. Since I developed the WebCamera primarily for Chrome browser on Linux OS, Chrome OS, Windows and An-

droid, I did not monitor problems that may occur in other browsers or on Mac OS and iOS. When the project was done for Chrome, I also launched WebCamera app in Edge and Firefox browsers on Windows. I figured out that the latest version of Firefox, specifically 76.0.1 (64-bit), does not support Atomics object [27], which is required for well-defined execution order in OpenCV.js with threads optimization. However, not optimized OpenCV works well in Firefox. Edge browser supports Atomics since version 79 so it accepts both not optimized and multithreaded OpenCV. Feedback from Mac OS and iOS users shows that WebCamera is not available for these platforms, and this issue requires further investigation.

Other limitations are related to implementation of use cases. I will start with Card scanning demo. First, it can detect only OCR-A font. To recognize other fonts, I need to create new digit templates to perform template matching. Second, the card number must be presented in the form of four groups of 4 digits giving 20 digits in total. For a different form of card number, another algorithm must be implemented. Third, sometimes it is not possible to recognize digits when the card contains an image. It is better to have uniform color instead of image on the card. Moreover, the digits should be clearly visible without merging with the card background. Last but not least, the background, on which the credit card is captured, should also be uniform and solid. Otherwise, the edges of the card may not be detected and the number recognition process will not start. In addition, user should not hold the card in hand as fingers will overlap the card contour and interfere with edge detection. Similar to Card scanning limitations, Document enhancement demo requires uniform and solid image background to correctly detect document edges. However, if no edges were detected, user can still process the current image and adjust the edges manually.

6.3 Future work

To make the app even more native and progressive, I would also add support for landscape mode in addition to portrait orientation. Furthermore, it can be useful to provide cache expiration period to be able to update the app sources without refreshing cache manually.

As it was described in Section 2.1, OpenCV supports DNN models generated by Intel's Model Optimizer. These models then are loaded by Intel's Inference Engine to perform efficient DNN inferensing. Both Model Optimizer and Inference Engine form OpenVINO Toolkit, which is a Deep Learning Deployment Toolkit [39]. This project is currently available only for C/C++ usage, and an interesting idea is to port this Engine to the Web platform to

run optimized models in a browser. Though, there are some difficulties that I have encountered while trying to compile this project by Emscripten. The major problem is that this project actively utilizes machine-dependent code, specifically, JIT (Just-In-Time) assembler for 64-bit architecture. However, "Emscripten cannot compile inline assembly code because it is CPU specific, and Emscripten is not a CPU emulator" [34]. So this work requires further research to make OpenVINO Toolkit platform-independent.

Chapter 7

Conclusion

The focus of the work presented in this thesis is the development of CPU-based WebCamera application with CV capabilities performed by the Wasm version of OpenCV library. The research aims to check the possibility to achieve a negligible loss of user-perceived quality in terms of performance compared to native applications. Implemented CV use cases, such as Instagram filters, Face Detection, Funny Hats, Card Scanning, Document Enhancement, Emotion Recognition and Invisibility Cloak, were tested in Chrome browser on a laptop and mobile device to measure performance in FPS. From the one hand, results show that the app is able to reach maximum possible FPS value (60 FPS) in the browser almost in all use cases performing on the laptop except blurring filters and Emotion recognition demo as they are the most compute-intensive algorithms. From the other hand, the mobile phone presents not impressive results meaning that the app still requires improvements to replace native mobile applications in the future. However, the mobile version can be implemented as a processing of captured images instead of real-time video processing. In this case, it is an emerging web alternative for similar existing native solutions.

The thesis also experiments with optimization options, such as SIMD and threads, applied by Emscripten compiler to build OpenCV.js Wasm file. These optimizations are used to improve the performance of running demos in a browser and estimate efficiency compared to not optimized OpenCV version. Measured statistics outline that SIMD option can increase the speed of processing by 2-4.5 times. However, it does not affect all use cases. In contrast, threads optimization provides performance improvement in all presented demos. The maximum speedup reaches a 3.5x rate. Applied together, SIMD and threads allow to execute CV algorithms on the web even 3-9 times faster.

Furthermore, the study found that OpenCV provides a vast range of algo-

rithms in areas of CV and ML, and OpenCV.js contains almost all modules that are implemented in C/C++ language. The diversity of OpenCV.js capabilities outperform the functionality of existing popular CV libraries for the web. Thus, it was feasible to implement all WebCamera demos with required functionality using only OpenCV.js and pre-trained models for face, eye and emotion detection. However, the application field of OpenCV is quite exhaustive and in addition to presented image filtering, face detection, emotion recognition and color segmentation includes motion tracking, augmented reality, mobile robotics, egomotion estimation and many others.

In order to address the demands on native-like experience, the WebCamera has been converted to PWA with a set of features such as app installation, offline mode and responsive screen size on mobile phones. It is very handy to have the app icon on a desktop and launch it in one click instead of opening a browser and looking for a bookmark or typing the website URL (Uniform Resource Locator) manually. Moreover, users do not face extra hassle related to app stores like authorization and approval processes because PWA apps are distributed just through URL. Speaking about the offline feature, it is a truly remarkable possibility as it allows to run any WebCamera use case without the internet like any native application. Screen responsiveness adjusts the camera view for any mobile device so that users can enjoy full-screen WebCamera format on mobile phones. All these progressive enhancement principles entail a smooth user experience associated with native apps.

Promising direction for future research can be porting Intel's Model Optimizer and Inference Engine from C/C++ to Wasm, similar to OpenCV.js. It would allow us to run optimized DNN models in a browser more efficiently to present new use cases in WebCamera app like facial landmarks detection, background removal or human pose estimation.

Overall, this study can help other developers to create their own web camera applications or camera-based services for the web using OpenCV functionality. Implemented Instagram filters can combine certain image processing algorithms and parameters to emulate image processing from the real Instagram app. Funny hats and Emotion recognition can be a complement to web prototype of Snapchat app. Card scanning use case can be integrated into a payment tool on the web, and the advantage of this service is that data does not leave a device during processing. Other WebCamera examples also demonstrate the ability to perform CV tasks in a browser with competitive performance at least on general-purpose processors designed for laptops, desktops and workstations.

From a contribution perspective, the application has had an impact on the web communities of such companies as Intel and Google. WebCamera demos were introduced at Chrome Dev Summit 2019 as examples of opti-

mized Wasm module running in Google Chrome browser [49]. Moreover, V8 Javascript engine developed by the Chromium project also showcased the app on the web page called "Fast, parallel applications with WebAssembly SIMD" [67].

Bibliography

[1] Belhumeur, P. N., Hespanha, J. P., and Kriegman, D. J. Eigenfaces vs. fisherfaces: recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*, 7 (1997), 711–720.

[2] Biorn-Hansen, A., Majchrzak, T. A., and Gronli, T.-M. Progressive web apps: The possible web-native unifier for mobile development. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies* (January 2017), pp. 344–351.

[3] Canny, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-8*, 6 (1986), 679–698.

[4] Culjak, I., Abram, D., Pribanic, T., Dzapo, H., and Cifrek, M. A brief introduction to OpenCV. In *Proceedings of the 35th International Convention MIPRO* (Opatija, Croatia, October 2012), IEEE, pp. 1725–1730.

[5] Dang, K., and Sharma, S. Review and comparison of face detection algorithms. In *2017 7th International Conference on Cloud Computing, Data Science and Engineering - Confluence* (2017), IEEE, pp. 629–633.

[6] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, June 2017), Association for Computing Machinery, pp. 185–200.

[7] Huang, T. S. Computer Vision: Evolution and Promise. *CERN School of computing* (September 1996), 21–26.

[8] Jensen, P., Jibaja, I., Hu, N., Gohman, D., and McCutchan, J. SIMD in JavaScript via C + + and Emscripten. In *Workshop on Programming Models for SIMD/Vector Processing* (2015).

[9] Leung, C., and Salga, A. Enabling webgl. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, 2010), Association for Computing Machinery, pp. 1369–1370.

[10] Ma, Y., Xiang, D., Zheng, S., Tian, D., and Liu, X. Moving deep learning into web browser: How far can we go? In *The World Wide Web Conference* (New York, NY, 2019), Association for Computing Machinery, pp. 1234–1244.

[11] Mobeen, M. M., and Feng, L. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *IEEE 14th International Conference on High Performance Computing and Communication* (Liverpool, UK, June 2012), IEEE, pp. 381–388. DOI: 10.1109/HPCC.2012.58.

[12] Papoutsaki, A., Sangkloy, P., Laskey, J., Daskalova, N., Huang, J., and Hays, J. WebGazer: Scalable Webcam Eye Tracking Using User Interactions. In *25th International Joint Conference on Artificial Intelligence* (New York, July 2016).

[13] Rourke, M. *Learn WebAssembly.* Packt Publishing Ltd, 2018, ch. What is WebAssembly?, pp. 11–12.

[14] Saragih, J. M., Lucey, S., and Cohn, J. F. Deformable Model Fitting by Regularized Landmark Mean-Shift. In *International Journal of Computer Vision* (September 2010), vol. 91, pp. 200–215. DOI: 10.1007/s11263-010-0380-4.

[15] Smith, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, San Diego, CA, 1999, ch. Morphological Image Processing, pp. 436–438.

[16] Steiner, T. What is in a Web View? An Analysis of Progressive Web App Features When the Means of Web Access is not a Web Browser. In *Companion of the The Web Conference 2018* (Lyon, France, April 2018), pp. 789–796. DOI: 10.1145/3184558.3188742.

[17] Szeliski, R. *Computer Vision: Algorithms and Applications.* Springer, Berlin, 2010, ch. What is computer vision?, pp. 3–10.

[18] Taheri, S. *Towards Engineering Computer Vision Systems: From the Web to FPGAs*. PhD thesis, Computer Science, University of California, Irvine, US, 2019. https://escholarship.org/uc/item/78b6q2wv/.

[19] Taheri, S., Vedienbaum, A., Nicolau, A., Hu, N., and Haghighat, M. R. Opencv.js: Computer vision processing for the open web platform. In *Proceedings of the 9th ACM Multimedia Systems Conference* (New York, NY, USA, 2018), Association for Computing Machinery, pp. 478–483.

[20] Viola, P., and Jones, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Kauai, HI, USA, 2001), vol. 1, IEEE, pp. 511–518.

[21] Wilson, P., and Fernandez, D. Facial feature detection using haar classifiers. *Journal of Computing Sciences in Colleges 21* (April 2006), 127–133.

[22] Zakai, A. Emscripten: an LLVM-to-JavaScript compiler. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Portland, OR, USA, October 2011), pp. 301–312. DOI: 10.1145/2048147.2048224.

[23] Amazon Web Services. Amazon Rekognition. https://aws.amazon.com/rekognition/, 2020. Accessed 25.3.2020.

[24] Archanjo, G. A. MarvinJ. http://marvinj.org/en/releases/marvinj_1.0.html, July 2019. Accessed 18.5.2020.

[25] Bhaumik, R. Bringing High-quality Imaging to the Web Platform. https://medium.com/@rijubratabhaumik/bringing-high-quality-imaging-to-the-web-platform-8b2e2eb67b56, January 2019. Accessed 25.3.2020.

[26] Bhaumik, R. WebCamera. https://github.com/riju/WebCamera/, 2019. Accessed 15.4.2020.

[27] Can I use. Atomics. https://caniuse.com/#search=atomi, 2020. Accessed 20.5.2020.

[28] Can I use. WebAssembly. https://caniuse.com/#search=wasm, 2020. Accessed 10.5.2020.

[29] Chrome Platform Status. WebAssembly SIMD. https://www.chromestatus.com/feature/6533147810332672, 2020. Accessed 20.5.2020.

[30] Dabrowski, P. Facemoji. https://github.com/PiotrDabrowskey/facemoji, July 2019. Accessed 18.5.2020.

[31] Danilo, A. WebAssembly Threads ready to try in Chrome 70. https://developers.google.com/web/updates/2018/10/wasm-threads/, October 2018. Accessed 25.3.2020.

[32] Davidson Fellipe. Lena.js. https://fellipe.com/demos/lena-js/, April 2020. Accessed 18.5.2020.

[33] Emscripten. https://emscripten.org/, 2015. Accessed 25.3.2020.

[34] Emscripten. FAQ. https://emscripten.org/docs/getting_started/FAQ.html, 2015. Accessed 20.5.2020.

[35] Gent, P. Emotion recognition with python, opencv and a face dataset. http://www.paulvangent.com/2016/04/01/emotion-recognition-with-python-opencv-and-a-face-dataset/, April 2016. Accessed 7.5.2020.

[36] Google AR. Three.ar.js. https://github.com/google-ar/three.ar.js/, 2018. Accessed 25.3.2020.

[37] Huang, S. Computer Vision .js frameworks you need to know. https://www.freecodecamp.org/news/computer-vision-js-frameworks-you-need-to-know-b233996103ce/, March 2019. Accessed 25.3.2020.

[38] Husar, A. 20 Progressive Web Apps Examples That Will Inspire You to Build Your Own. https://onilab.com/blog/20-progressive-web-apps-examples/, February 2020. Accessed 18.5.2020.

[39] Intel. OpenVINO Toolkit. https://github.com/openvinotoolkit/openvino/commits/2019, 2020. Accessed 20.5.2020.

[40] Juneja, P. Building Markerless AR For Web using Three.ar.js (Part 1). https://medium.com/@pulkit.16296/building-markerless-ar-for-web-using-three-ar-js-part-1-5eca95d545ec/, July 2018. Accessed 25.3.2020.

[41] Lee, A. 40 Examples of Progressive Web Apps (PWAs) in 2020. https://www.tigren.com/examples-progressive-web-apps-pwa/, January 2020. Accessed 18.5.2020.

[42] LeFevre, R. CamanJS. http://camanjs.com/examples/, February 2020. Accessed 18.5.2020.

[43] Lundgren, E. Tracking.js. https://trackingjs.com/, 2018. Accessed 25.3.2020.

[44] Made with WebAssembly. AutoCAD Web App. https://madewithwebassembly.com/showcase/autocad/, November 2019. Accessed 25.3.2020.

[45] Made with WebAssembly. Vlc.js. https://madewithwebassembly.com/showcase/vlc/, February 2020. Accessed 25.3.2020.

[46] MDN Web Docs. SharedArrayBuffer. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer/, January 2020. Accessed 25.3.2020.

[47] MDN web docs. WebAssembly. https://developer.mozilla.org/en-US/docs/WebAssembly, February 2020. Accessed 11.5.2020.

[48] Morony, J. Using the Camera API in a PWA with Capacitor. https://www.joshmorony.com/using-the-camera-api-in-a-pwa-with-capacitor/, September 2019. Accessed 18.5.2020.

[49] Nattestad, T. and Stepanyan, I. Oh the things you'll compile: Modern WebAssembly. https://www.youtube.com/watch?v=kZrl91SPSpc&feature=youtu.be&t=649, November 2019. Accessed 25.5.2020.

[50] OpenCV. https://opencv.org/about/, 2020. Accessed 25.3.2020.

[51] OpenCV Change Log. https://github.com/opencv/opencv/wiki/ChangeLog, 2020. Accessed 4.5.2020.

[52] OpenCV tutorials. Image Processing. https://docs.opencv.org/master/d2/df0/tutorial_js_table_of_contents_imgproc.html, 2020. Accessed 3.5.2020.

[53] Oygard, A. M. Headtrackr. https://github.com/auduno/headtrackr/, 2014. Accessed 25.3.2020.

[54] Oygard, A. M. Clmtrackr. https://github.com/auduno/clmtrackr/, 2018. Accessed 25.3.2020.

[55] Paris, S. Fixing the gaussian blur: the bilateral filter. https://people.csail.mit.edu/sparis/bf_course/slides/03_definition_bf.pdf, June 2007. Lectire notes. Accessed 30.4.2020.

[56] Pisarevsky, V., and Kurtaev, D. Deep Learning in OpenCV. https://github.com/opencv/opencv/wiki/Deep-Learning-in-OpenCV/, December 2018. Accessed 25.3.2020.

[57] Progressive Web App Room. Great examples of progressive web apps in one room. http://progressivewebapproom.com/index.html. Accessed 18.5.2020.

[58] PWA Workshop. Adding a Web App Manifest. https://pwa-workshop.js.org/1-manifest/#manifest-fields, 2020. Accessed 4.5.2020.

[59] Richard, S., and LePage, P. What are Progressive Web Apps? https://web.dev/what-are-pwas/, January 2020. Accessed 25.3.2020.

[60] Rosebrock, A. PyImageSearch. https://www.pyimagesearch.com/, 2020. Accessed 18.5.2020.

[61] Said, P. Vue-pwa-camera. https://github.com/pierresaid, April 2020. Accessed 18.5.2020.

[62] Sajjad, Z. Ai in browsers: Comparing tensorflow, onnx, and webdnn for image classification. https://blog.logrocket.com/ai-in-browsers-comparing-tensorflow-onnx-and-webdnn-for-image-classification/, December 2019. Accessed 16.5.2020.

[63] Santoni, M. Progressive Web Apps browser support and compatibility. https://www.goodbarber.com/blog/progressive-web-apps-browser-support-compatibility-a883/, January 2018. Accessed 25.3.2020.

[64] Surma. Emscripting a C library to Wasm. https://developers.google.com/web/updates/2018/03/emscripting-a-c-library, March 2018. Accessed 12.5.2020.

[65] TangibleJS. Code Libraries: Computer Vision. `https://tangiblejs.com/libraries/computer-vision/`, 2020. Accessed 25.3.2020.

[66] TensorFlow. TensorFlow.js is a library for machine learning in JavaScript. `https://www.tensorflow.org/js/`. Accessed 25.3.2020.

[67] V8 project. Fast, parallel applications with WebAssembly SIMD. `https://v8.dev/features/simd/`, January 2020. Accessed 25.3.2020.

[68] Wang, C. What's the difference between haar-feature classifiers and convolutional neural networks? `https://towardsdatascience.com/whats-the-difference-between-haar-feature-classifiers-and-convolutional-neural-networks-ce6828343aeb`, August 2018. Accessed 7.5.2020.

[69] WebAssembly. `https://webassembly.org/`, 2020. Accessed 10.5.2020.

[70] Zatepyakin, E. Jsfeat. `https://github.com/inspirit/jsfeat/`, 2018. Accessed 25.3.2020.