Wrocław University of Science and Technology
Faculty of Electronics, Photonics and Microsystems

Field of study: **Electronics**
Speciality: **Advanced Applied Electronics (AAE)**

MASTER THESIS

Enhancing the capabilities of digital cameras by using machine learning methods

Zwiększanie możliwości aparatów cyfrowych poprzez wykorzystanie metod uczenia maszynowego

Dominik Giełbaga

Supervisor
Ph.D. Tomasz Kubik

Keywords: digital cameras, machine learning, firmware addon

WROCŁAW, 2023

Streszczenie

Niniejsza praca magisterska została poświęcona zbadaniu zasadności zastosowania metod uczenia maszynowego w celu zwiększenia możliwości aparatów cyfrowych. Tradycyjne urządzenia fotograficzne napotykają ograniczenia w zakresie jakości, zdolności rozpoznawania scen oraz przetwarzania obrazów, które można przezwyciężyć poprzez inteligentne integrowanie algorytmów wymienionego wyżej uczenia.

Badania rozpoczęto od przeglądu stosowanych obecnie technik w zakresie wizji komputerowej i systemów uczących się, podkreślając przydatność innowacyjnych rozwiązań. Następnie skupiono się na architekturach układów wbudowanych badanych urządzeń i ich obecnych oprogramowaniach. Na podstawie zdobytych informacji zaproponowano autorskie rozwiązania poszerzające funkcje aparatów cyfrowych. W kolejnych częściach pracy przedstawiono sposoby wdrożenia nowych metod, a także omówiono problemy napotkane podczas realizacji tego zadania. Uzyskane wyniki porównano między sobą lub ze wzorcami oraz skomentowano pod względem przydatności ich użycia w praktyce. W przypadku poszerzenia istniejących już funkcji udało się zwiększyć ilość pomocniczych obrazów nakładanych na widok z obiektywu, które ułatwiają kadrowanie. Dokonano tego poprzez wprowadzenie zmian do znajdującego się w oprogramowaniu kodu źródłowego, a następnie kompilacji skrośnej kodu i wczytaniu nowej wersji programu do pamięci urządzenia. Ponadto przeprowadzono testy prototypów funkcji odpowiadających za automatyczny balans bieli oraz wykrywanie rys na folii ochronnej telefonów komórkowych, które bazują na wykorzystaniu konwolucyjnych sieci neuronowych.

Słowa kluczowe: aparaty cyfrowe, uczenie maszynowe, nakładka na firmowe oprogramowanie

Abstract

In the thesis I examined the feasilibity of applying machine learning methods to enhance the capabilities of digital cameras. Traditional photographic devices encounter limitations in image quality, scene recognition, and image processing. These restrictions can be overcome through the intelligent integration of the prior mentioned learning algorithms.

I began the research with a comprehensive review of currently employed techniques in field of computer vision and machine learning highlighting the importance for innovative solutions. Subsequently, I studied the architectures of the examined devices' embedded systems and their applied firmwares. Based on the gathered information, I proposed original solutions expanding the functionality. Afterwards, I presented ways of implementing new methods and discussed the problems encountered during the tasks' execution. Furthermore, I compared the obtained results among themselves or with reference standards and commented on their practicability. In the case of expanding existing functions, I managed to increase the number of auxiliary images overlaid on the lens view, which help with framing. I achieved this by introducing changes to the source code of software, followed by cross-compiling the program and loading its new version into the device's memory. Moreover, I conducted tests on prototypes of functions performing automatic white balance and detecting cracks on mobile phone screen protective films. These methods based on use of convolutional neural networks.

Keywords: digital cameras, machine learning, firmware addon

Contents

List of Figures

List of Tables

List of Listings

Acronyms and abbreviations

4D (*Four Dimensional*)
A/D (*Analog to Digital*)
ABI (*Application Binary Interface*)
ADAM (*Adaptive Moment Estimation Optimiser*)
ADC (*Analog to Digital Converter*)
AF (*Autofocus*)
AIF (*All In Focus*)
API (*Application Programming Interface*)
ARM (*Advanced RISC Machine*)
ASIC (*Application-Specific Integrated Circuit*)
ASIP (*Application-Specific Instruction Set Processor*)
ATA (*Advanced Technology Attachment*)
BMP (*Bitmap*)
BN (*Batch Normalisation*)
CCD (*Charge Coupled Device*)
CFA (*Colour Filter Array*)
CISC (*Complex Instruction Set Computing*)
CLAHE (*Contrast Limited Adaptive Histogram Equalization*)
CMOS (*Complementary Metal-Oxide-Semiconductor*)
CNN (*Convolutional Neural Network*)
CPU (*Central Processing Unit*)
CUDA (*Compute Unified Device Architecture*)
DDR (*Double Data Rate*)
DMA (*Direct Memory Access*)
DoF (*Depth of Field*)
DRAM (*Dynamic Random Access Memory*)
DSLR (*Digital Single-Lens Reflex*)
DSP (*Digital Signal Processor*)
EABI (*Embedded Application Binary Interface*)
EEPROM (*Electrically Erasable Programmable Read Only Memory*)
EOS (*Electro-Optical System*)
FPGA (*Field Programmable Gate Arrays*)
FV (*Focus Value*)
GPU (*Graphics Processing Unit*)
GUI (*Graphical User Interface*)
HD (*High Definition*)
HOG (*Histogram of Oriented Gradients*)
ISP (*Image Signal Processor*)
JPEG (*Joint Photographic Experts Group*)
JSON (*JavaScript Object Notation*)

JTAG (*Joint Test Action Group*)
KVM (*Kernel-based Virtual Machine*)
LBP (*Local Binary Patterns*)
LCD (*Liquid-Crystal Display*)
MinGW (*Minimalist GNU for Windows*)
MLH (*Maximum Likelihood*)
MPSDD (*Mobilephone Panel Surface Defects Detection*)
NB (*Naive Bayes*)
NTSC (*National Television System Committee*)
OS (*Operating System*)
PC (*Personal Computer*)
PCMCIA (*Personal Computer Memory Card International Association*)
RGB (*Red Green Blue*)
RAM (*Random Access Memory*)
RF (*Random Forest*)
RISC (*Reduced Instruction Set Computing*)
ROM (*Read Only Memory*)
RTL (*Register Transfer Language*)
RTOS (*Real Time Operating System*)
SCSI (*Small Computer System Interface*)
SD (*Secure Digital*)
SDRAM (*Synchronous Dynamic Random Access Memory*)
SoC (*System-on-a-Chip*)
sRGB (*standard Red Green Blue*)
SVM (*Support Vector Machine*)
TCG (*Tiny Code Generator*)
TIFF (*Tagged Image File Format*)
UART (*Universal Asynchronous Receiver-Transmitter*)
USB (*Universal Serial Bus*)
VRAM (*Video Random Access Memory*)
WB (*White Balance*)

Chapter 1

Introduction

1.1. Foreword

In recent years, the rapid advancements of machine learning techniques revolutionised many fields of studies. They bring new approaches to solving problems, offering different perspectives to overcome current challenges. Besides traditional machine learning methods, lately there is a wide growth in the deep learning branch incorporating neural networks. Because of their self-adjusting nature of hidden layers, they showcase huge potential for new solutions and overcoming present difficulties. The following thesis was created in order to explore the state-of-art machine learning methods with intention to use them alongside classic digital camera functions.

Digital cameras have become an integral part of people's lives. With the increasing presence of smartphones and dedicated devices, capturing high-quality images has become more accessible to a wider range of users. The continual advancements in technology have significantly improved the standard and convenience of image shooting, but difficulties still exist when it comes to achieving optimal results under various conditions. Most of the current photographic devices have a firmware that manages all of their functions, from exposure parameter selection and focus control to pre-processing and post-processing of captured images. Despite existing hardware capabilities, the scope of these functions is often incomplete or limited. To address this issue, a new software containing extensions can be loaded into the camera's memory card and run as add-on to original firmware. The emerging field of machine learning offers exciting opportunities to overcome these challenges and enhance the capabilities of digital cameras.

1.2. The aim and scope of the work

The purpose of the study is to analyze the feasibility of using machine learning methods to enhance the range of features offered by digital cameras. A structured exploration of the current state-of-the-art machine learning algorithms in computer vision and their applicability to embedded systems' environment is conducted. Through an extensive literature review, relevant techniques and methodologies are identified that can be adapted to enhance the functions of these devices. Based on that, ideas and implementations of solutions are described, such as deep neural networks for automatic white balance and phone screen cracks detection. The proposed algorithms are then experimentally tested and evaluated. Furthermore, the hardware of existing digital still cameras is studied. The main focus in this part is aspect of programming the embedded devices in order to implement developed functions. The working case on which the research will be based is Canon's family of digital cameras.

1.3. Structure of the work

The thesis is divided into seven main sections. In the introduction an outline and motives of the thesis are explained. In the second section, a literature review on machine learning in digital photography is conducted. In this chapter functions regarding different approaches on autofocus, white balance and edge detection are analysed. In the third section, a case study of programming digital cameras is performed. Firstly, the hardware elements of such embedded system are described in order to analyse its architecture and possibilities. The main part of this section is focused on exploring capabilities of firmware, and furthermore its customisation and extension opportunities. Shortly after, a closely related topic of cross-compilation and its usage for the needs of this thesis is explained. Lastly, a few completed systems are shown. In the fourth chapter, the existing features of Magic Lantern custom firmware are discussed, as well as programming opportunities for further extensions and developing new functions with provided application programming interfaces. Also, setting up the environment for emulating the original firmware and the add-on extension is addressed. The fifth section is about introducing the ideas for new solutions and providing the implementations. The functions of custom cropmarks, automatic white balance and cracks detection on phone screen are subsequently described. The sixth chapter contains the results of analysed algorithms. The achieved outcomes are discussed and compared. In the last chapter, summary and conclusion about the experience gathered throughout the whole work is stated.

Chapter 2

Application of machine learning in digital photography

2.1. Autofocus

Focusing is a crucial aspect of photography as it helps to capture desired sharp sections in the picture. An image is said to be in focus when the light rays from the subject being photographed converge to a focal plane, where the camera's sensor is. Right focusing can be achieved either manually or automatically, and while many professionals use manual focus, autofocus offers several advantages. It is easy to use and offers opportunities such as following a moving object without significant losses of sharpness. A wide variety of algorithms were proposed, which can be divided into passive and active category.

The two most frequently used methods of autofocus in digital cameras are phase detection (an active approach) and contrast detection (a passive approach), see figure 2.1. The first one uses a dedicated autofocus sensor to measure phase difference between the light rays passing through prisms and microlenses within the camera. If there is no phase difference, then the subject is in focus. On the contrary, the lens is adjusted to properly focus on a subject. This method is faster than contrast detection, making it better suited for capturing fast-moving subjects and tracking their movements. On the other hand, contrast detection AF (Autofocus) analyses the contrast of pixels on the camera's sensor and moves the lens back and forth until it finds the right focusing point. When the contrast is at its highest, the subject is in focus. This method is slower than phase detection, but overall it performs better in low light situations and capturing static subjects. Moreover, in some modern cameras a hybrid method is implemented, which combines both approaches to obtain better performance in various shooting situations [7].

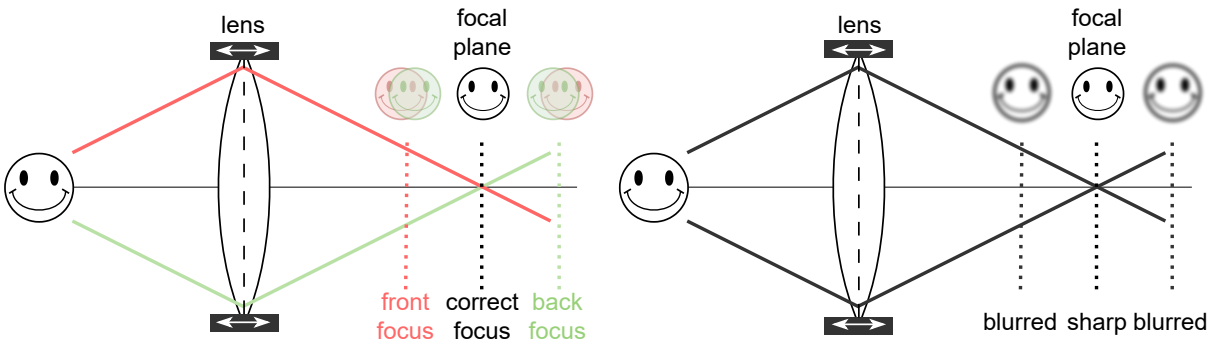

Figure 2.1: An overview of focusing the subject on the camera sensor

2.1.1. Colour information autofocus

The task of improving already existing autofocus algorithms has attracted many researchers. One of the proposed approaches taken addressed the use of colour information [7] in the implementation of the Absolute Threshold Gradient-Based Contrast Algorithm. Algorithms of this kind are commonly implemented in modern DSLR cameras due to their quick response and straightforward approach. The main function calculates the sum of absolute values of luminance (brightness) differences between pixels separated by a certain number of pixels both horizontally and vertically, resulting in an average luminance difference, also known as the FV (Focus Value). This value is low when the picture is out-of-focus, and significantly increases as sharpness improves, reaching its peak at the best focus point.

The authors of previously mentioned paper believed that previous passive autofocus approaches did not include all the available information on three channels of RGB, as well as were not optimised in terms of choosing the pixel separation parameter. Therefore, they proposed an autofocus algorithm that takes into account all three colour components and multiple pixel gap values to generate more efficient results. Furthermore, a pre-edge filtering for sharper FV and easier maximum value detection is proposed as shown in the figure 2.2.

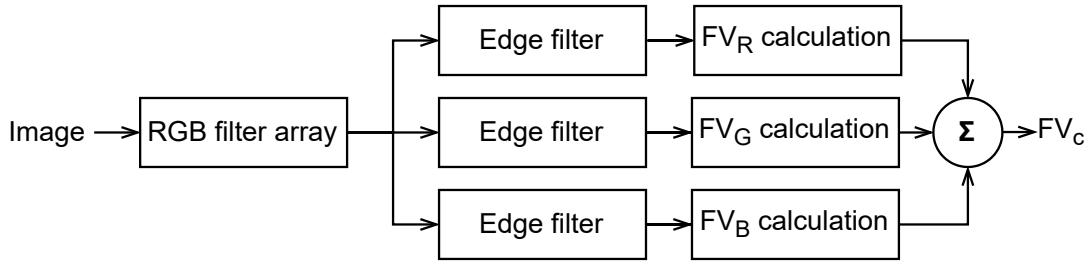

Figure 2.2: A schematic of Focus Value evaluation function, based on [7]

Authors stated that using all three color information matrices in the FV calculation leads to a steeper slope of the FV curve around the optimal in-focus point compared to the current algorithm that uses only a single color for the calculation. The addition of edge filtering also positively affect the slope of FV.

2.1.2. Image refocusing

To prevent autofocus errors a new through-focus algorithm was proposed in [1]. Blurred images can be corrected after capture using a deconvolution process. However, such approach can be computationally demanding. Researchers then thought of shooting a series of photos, with different subjects in focus. Such approach is called AIF (All In Focus), and it aims to create a final image that combines multiple low-noise image captures taken with a wide aperture that does not suffer from the noise associated with a single image shot with smaller aperture. However, that is not exactly what the authors propose. Based on AIF image, a bokeh can be produced. It is an effect, in which the subject is in sharp focus while the background and other parts of the image are blurred in a pleasing way. Bokeh is often characterized by its smooth, creamy, and sometimes circular appearance, and it can add a sense of depth and dimensionality to an image.

The idea behind presented algorithm, which is shown in figure 2.3, is to produce AIF image from a through-focus shooting series. Moreover, from the same series a range segmentation is performed which allows to derive the range of interest for bokeh effect in refocused image. The authors managed to achieve a simple solution designed for constrained computational resources, as are available in a digital camera. The algorithm managed to highlight the subject of an image meanwhile keeping the background desirably blurred.

Figure 2.3: Image refocusing algorithm schematic, based on [7]

A similar approach is considered in [12], where authors want to apply the subject sharpness and background blur also in mobile phone cameras. Due to the size limitation, such thing is harder to achieve, but there is an alternative way. Having the depth information of the scene, defocusing effect can be applied to overcome hardware limitations. However, as most of phones have only one camera, it is another obstacle, because it is challenging to get depth information basing on a single view. The authors state that other algorithms such as those using saliency were used in order to obtain the depth map. When the foreground and background of an image have significant differences, such functions are capable of generating pleasing results. However, in the opposite case, much poorer results were obtained.

A new algorithm proposed for synthesizing defocused images from a single input image incorporates a saliency detection based on face prior, as well as a novel gradient domain guided image filter. When adopting the idea to portrait images, the faces are regarded as the most attention important areas. Nowadays, most mobile phones have already built in face-detection functions. Therefore, it could be used for detecting saliency region without significantly increasing the further computational costs. Finally, the algorithm is utilized to detect the foreground portrait and is then enhanced using the new gradient domain guided image filter. The results of this approach are images which look as if they have been taken with wider aperture than they truly were.

2.1.3. Machine learning possibilities

Machine learning can be utilized in autofocus systems to improve their accuracy and efficiency. The algorithms can be trained to recognize and track objects or features in an image by analyzing images with known depths and corresponding focus settings. Based on that, the appropriate focus positions for different situations can be predicted. The use of this approach might be especially useful in complex situations where traditional autofocus techniques may struggle, such as low-light conditions or scenes with complex foregrounds and backgrounds.

Supervised learning in autofocus

In [34], the authors address identifying global and significant local optimums when focusing an image. Focus stacking, as already mentioned, is a crucial task where a series of images are captured and merged in post-processing to create a final image that is all in focus. The research highlights the potential of using machine learning to semi-automate the development of heuristics for addressing these challenges. Specifically, a supervised learning approach is used, where the model learns from labelled training examples that provide the correct solutions.

The effectiveness of such method largely depends on the quality of the recorded features that can differentiate between various scenarios. In the research, the authors considered sixty features that were promising. These attributes are all based on the current and previous values of the focus measure and include both novel characteristics and generalizations of previously researched and proposed features. New ones were generated by applying straightforward operations or their combinations of fundamental features. Basic functions include comparing two attributes, computing their ratio, or calculating the logarithm of their subtraction.

A very important factor was to assure that the obtained results represent reality properly. Initially, a remote control application was developed that enables the camera to be linked to a computer via a USB cable and managed through software. The basic operations of the camera, including adjusting the aperture, displaying the live preview stream, and regulating the focus position of the lens were mimicked. Such preparation allowed to collect 25 sets of images with various range of scenarios, as well as lighting conditions. Each set of pictures consists of either 167 (if 50 mm lens were used) or 231 (if 200 mm lens were used) JPEG images, one for each focus position of the lens. The experiments were conducted using a Canon EOS 550D camera.

Next, focus measurements and appropriate labels for each location were acquired. Each instance in the data is a vector of feature values and the correct classification or tag for that instance. The labelling was done in a semi-automated way where initially it was automatically generated and then manually adjusted to eliminate any inaccuracies caused by noise. Lastly, an actual machine learning data was generated. Starting from the near focus and continuing until the farthest, the algorithm computed the values of all sixty features using the current and previous values of focus, then generated the class label that corresponds to the current lens position, and finally advanced the distance as specified by the class tag.

The subsequent step involved feature selection. Identifying and saving only the most crucial attributes for creating effective heuristics. The selected ones were then retained in the data, and the others, unnecessary or irrelevant were eliminated before passing them to the learning algorithm. The two key reasons for such pre-processing are enhancing the efficiency of the learning process and improving the quality of the learned heuristic. The used attribute evaluator constructed a decision tree classifier. Based on the features, it evaluated accuracy of the tree applying 10-fold cross-validation. This accuracy was later used as a figure of merit.

During the experiments, authors observed that the machine learning-based algorithm showed a significant increase in speed, reducing the average number of iterations required for focusing by 37.9% in common photography settings and 22.9% in a more challenging focus stacking scenario, while still achieving the same level of accuracy.

Unsupervised learning in autofocus

In [48] approach, an unsupervised machine learning is presented with the idea to annotate data. This task involves detecting and localizing a single object of interest in an image. It incorporate finding a tightly fitting bounding box around the object and generating a binary mask that highlights the regions of interest. This type of annotation is simpler to validate by humans with a straightforward "Yes/No" response, and it can be directly utilized for training an object detector. Unlike other annotation methods that require identifying numerous boxes covering all items in an image, this approach only focuses on a single object of interest.

The authors distinguished two categories of objects present in the images: 'things and stuff' and 'edges and connections'. The first includes individualities such as persons or a cars, as well as shapeless entities like roads or grass referred as stuff, which can be identified as recurring unpredictable patterns. The second is meant to detect edges and junctions playing a vital role in identifying foreground objects. The reason for the saliency measure's bias towards edges and junctions can be understood by examining an image containing a single object with a uniform texture and a smooth boundary, placed against a homogenous background. The combination of these attributes results in a saliency map of objects that has highly desirable properties.

To assess the effectiveness of the object location proposals, the precision recall curve was applied as a performance metric. However, the recall rate vs. object proposals is more suitable for comparing the recall rate at a high number of proposed locations, but not for evaluating the precision when only one object location is proposed. To provide a comprehensive evaluation,

both the PRC and the recall rate vs. object proposals are reported. Moreover, the precision and recall are presented as a function of the number of object proposals per image.

The authors claim, that based on the average precision, the proposed object locations significantly outperform those of other approaches. The first proposal per image by this method correctly detects an object in 42% of the pictures, which is approximately 10% better than the closest of the four different considered approaches. However, according to recall criterium, as the number of proposed windows increases, other methods can locate objects more effectively despite the outstanding performance of the authors' first box result. Nevertheless, the authors still state that their approach are well suited for weakly labelled data annotation. This is because in such application the first proposed object location precision is crucial. The simplicity and effectiveness makes this method an appealing option for integration with supervised methods and for addressing a broad range of problems.

Deep learning in autofocus

Contrast and salient object detection

Recently, deep CNNs (Convolutional Neural Networks) have been applied to achieve more robust features for salient object detection, resulting in significantly better outcomes than the prior state-of-the-art methods. CNNs contain more high-level semantic information as they are commonly pre-trained on datasets for visual recognition tasks. However, in all of these techniques, these networks are applied at the patch level (small divisions of the image) rather than the pixel level. Furthermore, all image patches are considered as independent data samples for classification or regression, even if they overlap. Because of that, these methods need to run a CNN at least thousands of times (once for every patch) to obtain a complete saliency map. This results in significant redundancy in computation and storage, making both training and testing a very time and space-consuming task.

Guanbin Li and Yizhou Yu [27] propose an end-to-end deep contrast network to improve current possibilities. In this case, end-to-end means that the deep network is designed to only require a single run on the input image to produce a complete saliency map with the same pixel resolution as the input image. The network comprises a fully convolutional stream at the pixel level and a spatial pooling stream at the segment level that work together to make a saliency map. To conceive such architecture, they had the following considerations. Firstly, the network should be deep enough to produce multi-level features for detecting salient objects at different scales. Secondly, the network should be able to discover subtle visual contrast across multiple maps holding deep features. Lastly, fine-tuning an existing model is desired since there are not enough images to train such a deep network from scratch.

The authors observed that salient objects frequently have irregular shapes, which can make the saliency map look uneven around the edges. To overcome this, their multiscale fully convolutional network operates at a subsampled pixel level. To improve modelling of visual contrast between regions and saliency along region boundaries, the authors designed a segment-wise spatial pooling stream in their network.

Regarding the training, firstly the second stream of the network is trained using the VGG16 [47] architecture pre-trained on the ImageNet dataset [26]. The initial weights were obtained by extracting segment features from the superpixels of the training images. After initialization, the two streams of the network were fine-tuned alternately. During each epoch, they fixed the parameters in one stream and train the other stream using stochastic gradient descent. The weights were updated for fusing the saliency maps from the two streams and the parameters in the multiscale fully convolutional network. The complete algorithm consisted of superpixel segmentation and spatial coherence. In the first case, the input image was decomposed into non-overlapping parts. Secondly, as the deep contrast network assigns saliency scores to individual

pixels or segments, it does not consider the consistency of saliency scores among neighboring ones. To address this issue, a pixelwise saliency refinement model was proposed that uses a fully connected conditional random fields. This model helps to improve the spatial coherence of saliency scores.

Analysing the results, authors claim that their implementation outperforms significantly all five other salient object detection algorithms taken into consideration in this work. Therefore, these experimental results indicate that the proposed deep model shows a considerable improvement over the existing state-of-the-art techniques.

Automatic focus position neural network

Active illumination and phase detection are capable of determining the correct focus in a single time step, which makes them suitable for capturing dynamic scenes or moving objects. However, these methods require specialized hardware, which can be expensive and may reduce image quality. In addition, these techniques miss the fundamental goal of autofocus in the era of artificial intelligence, which is to optimize the captured image by analyzing scene content and motion. According to [53], the assumption that a global "best" focus position exists is no longer applicable, and even with specialized hardware, autofocus should be based on the image itself.

Traditionally, contrast maximization, requires numerous iterations to produce satisfactory results and does not need any specialized hardware beyond the basic camera. However, recent studies have demonstrated that deep learning control can achieve the advantages of scene-based contrast optimization in real-time single-frame rates similar to phase detection. Hand-crafted evaluation metrics measure the focal degree of an image, and search strategies aim to maximize them by sweeping the focus range. However, these methods have clear drawbacks: the evaluation metric provides no direct information about the focal state, and it requires many comparisons and time steps. In contrast, humans can instantly determine if a minor or significant focus adjustment is necessary from a single image. Such behaviour also motivates the use of machine learning techniques in focus control. These methods take a different approach by exploring the connection between the image and focus position, without relying on explicit evaluation metrics. However, relying on hand-crafted features extracted from the images might still be crucial.

The authors of [53] introduced a novel approach to address the autofocus problem through deep learning. Rather than predicting the optimal focal position from a defocused image, a CNN is used to estimate the absolute distance from the optimal focus and a discriminator to evaluate if an image is in-focus. This two-component AF system effectively resolves the uncertainty problem and enhances focus speed and image quality compared to conventional methods.

To address the problem of cameras' limited depth of field, authors propose a pipeline for all in focus imaging. The environment can be estimated dynamically from captured frames, and the region of interest for focus analysis can be determined by an agent. It has been demonstrated that the minimum number of captured frames for generating static all in focus images can be achieved by the proposed pipeline, surpassing traditional methods in this area.

Calibration The models should be firstly calibrated for a given camera. Calibration in this case means finding proper parameter values, which allow numerical conversion of focused image to defocused. It can be achieved by involving a method, which finds and maps variables such as defocus blur filter and scaling factor, based on distance between the two images on optical axis. Looking at figure 2.1, this distance could be represented as length between the front or back focus plane and focal plane. The process involves applying multiple various blur filter and scaling factor values on original images, computing its similarity with the defocused picture, and updating the mapping. Further repeating these actions for distinct planes of focus results in finding a numerical correlation, which is useful in next operations.

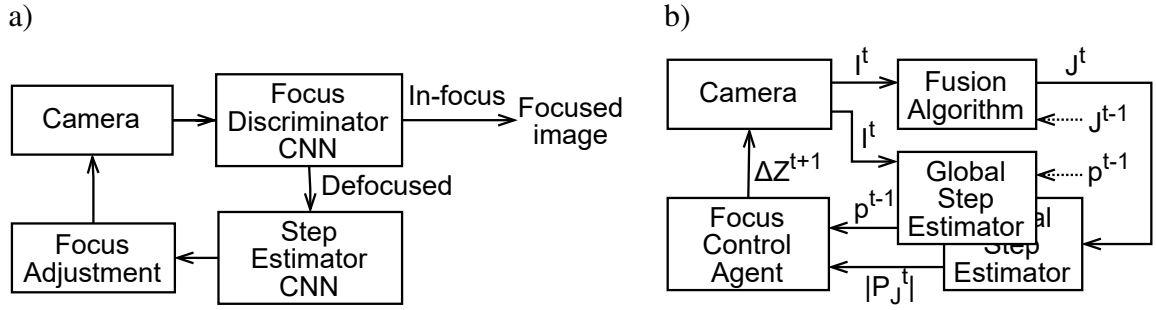

Figure 2.4: Structure of processing pipelines in: a) Deep Learning AF, b) AF positioning with the use of Agent; based on [53]

Deep Learning AF Pipeline Architecture of proposed system structure consists of a CNN, called the "step estimator", implemented to estimate the distance from the optimal focus position using defocused images. This network, shown in figure 2.4a, takes small image (512 x 512 pixels) as input and outputs an estimated distance from the in-focus position.

Traditional autofocus algorithms move the camera's lens through the entire focus range or until a certain tolerance is reached. This approach is inefficient because it cannot evaluate if an image is already sharp during the search process and often requires restarting the system. The proposed solution to improve this process is a focus discriminator that can directly determine if an image is sharp enough and end the algorithm when the optimal focus is achieved. The discriminator, just like the estimator, takes the same size image of (512 x 512 pixels) as input and outputs a predicted label of either "in-focus" or "defocus." It is also shown in figure 2.4a.

The proposed autofocus process begins by selecting a random focal point, and at each step, the focus discriminator examines the image and stops the process once a clear image is obtained. If the image is still blurry, the distance predictor determines how far to move. The focal points is then updated with position in direction, based on their estimated distances from the ideal focal point.

All In Focus imaging The all in focus imaging approach adapted by authors introduce a smart approach that gives priority to a position with the ability to enhance the image quality of upcoming merged frames. To develop such strategy, it is necessary to analyse the focus state of the entire image. To achieve this, the step estimator is converted to a fully convolutional network. Its considered structures are described in table 2.1. Output from the new estimator contains distances for all the patches, which are arranged in a matrix called a step map. Outcome values from step estimator are always positive. However, if the difference between position at time t and $t-1$ is higher than 0, it means that the distance is being adjusted in the wrong direction. In such case, a minus sign is assigned for step map at time t.

Table 2.1: Local and global step estimators structures, based on [53]

Local step estimator	Global step estimator
Conv(4,8,8)	Conv(4,8,1)
Conv(8,4,4)	Conv(8,4,1, dilation = 8)
Conv(8,4,4)	Conv(8,4,1, dilation = 32)
Flatten + fully connect (1024)	Conv(1024,4,1, dilation = 32)
fully connect (512)	Conv(512,1,1)
fully connect (10)	Conv(10,1,1)
fully connect (1)	Conv(1,1,1)

A new pipeline is proposed to estimate the focus position of a frame in a way that optimizes the quality of the fused all in focus images. Since the capture process is recurrent, the idea is to estimate focus position based on camera and environment states estimations up to that point. The proposed pipeline uses an adaptive approach to determine the travel distance. It consists of a global step estimator, a fusion algorithm and a rule-based or reinforcement focus control agent (see figure 2.4b). The process starts with camera capturing and image I^t with a specific focus position Z^t. It is then forwarded to global step estimator and fusion algorithm. The latter one outputs estimated all-in-focus frame J^t, based on its previous time step instance J^{t-1} and the image from camera. The global step estimator evaluates two step maps – a signed one p^t using its prior step sample p^{t-1} and I^t, and unsigned one $|P_J^t|$ from J^t. Both are forwarded to the agent, which based on them computes focus position for the next time step ΔZ^{t+1}.

The rule-based agent improves the image quality of the next merged frame by making greedy decisions - based on immediate benefits. On the other hand, the reinforcement agent works to enhance the image quality of all future merged frames. The process of achieving in-focus images is similar for both static and dynamic scenes, with a distinction in termination conditions for static cases. The capture process stops when no more enhancements can be made to the fused image quality. This typically occurs when either the remaining regions are outside the focus range or when the next predicted focal position overlaps with the depth of field of previous ones.

Performance evaluation To assess the performance of the step estimator and focus discriminator, experiments on 60 samples were conducted. Since the training data only included specific ranges of values, the focus discriminator had a 36% failure rate for certain values in testing. More training data in this range may improve performance.

In order to evaluate the efficiency and reliability of the suggested autofocus method, various objects were focused on under different lighting conditions and at different distances. The experiment began from four initial positions. Even though the defocus model was calibrated using fluorescent lamps, the autofocus method accurately predicted the focal point for all types of light sources. The experiment revealed that when using incandescent bulb or natural light, the noise level was higher in comparison to fluorescent one. This observation suggests that the method is capable of effectively managing and mitigating noise.

Compared to traditional methods that require over 10 steps, the new method achieves optimal focus in just a few, or even one step. It is versatile, efficient, and does not require a specific starting position. The proposed method's computation time remains comparable to traditional methods, even without GPU (Graphics Processing Unit) acceleration. Despite the perception of deep learning being computationally demanding, this approach leverages its data analysis capabilities while maintaining computational efficiency.

Regarding static scenes, this method is different from existing approaches because it does not need to scan the entire range or go through refinement steps. Instead, it focused on capturing important frames and visits each position once, which means fewer frames and faster processing time. Additionally, convolutional layers parameters can be adjusted to reduce the number of patches, lowering time-consuming computations. Dynamic scenes, where camera makes axial or transverse movements, also benefit from using this approach. As for agents, the reinforcement one should be considered in real-time systems, as it required 10-fold less time to process a frame than his rule-based counterpart, while both used the same processor.

The authors conclude that their proposed solution is efficient and achieves quicker focus than traditional contrast maximisation methods. Moreover, the step estimator not only obtains better results than more conventional focus stacking, but also is able to produce all-in-focus videos.

2.2. White balance

White balance is a fundamental process applied to digital images, primarily in photography and computer vision, to adjust the colours in the image to look as natural as possible for human eye. Its primary purpose is to ensure that objects in a scene appear with the same colour even when captured under varying illumination conditions. The camera's onboard integrated signal processor performs white balance as one of the initial steps in manipulating the raw-RGB image captured by the sensor. The process involves adjusting the relative levels of the RGB colour channels to eliminate any colour cast and make achromatic objects appear neutral, typically grey. However, the processor's colour rendering process may depend on aesthetic considerations based on photographic preferences. Such thing can mislead the image look from the white light assumption. These preferences can differ based on factors such as cultural preference and scene content [3, 17].

2.2.1. Haze removal

The following research [5] considers utilising white balancing method for haze removal in the images. Bad weather such as haze, fog, and mist can make it harder to see things outside. Haze comes from various sources like farming, traffic, and industry, as well as natural things like volcanic ash, plants, and wildfires. The particles of which these things are made are bigger than air molecules but smaller than fog droplets. This can make it tough to take good pictures of faraway things because the light gets scattered, and the colours look dull. The complete method contains white balancing, saliency map, morphological opening and contrast enhancing. The schematic overview of chosen method is shown in figure 2.5.

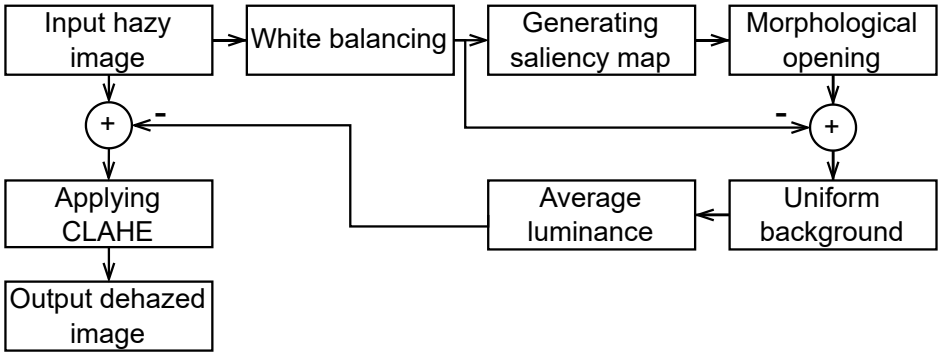

Figure 2.5: An overview of dehazing method schematic, based on [5]

The goal here is to produce natural-looking images by removing any chromatic casts created by atmospheric color. The authors state that numerous white balancing algorithms have been developed in recent years, but for a computationally efficient dehazing approach, but they have chosen the shades of gray color constancy technique proposed by [18]. The primary objective of the algorithm is to determine the color of the illuminant, which is affecting the image. The idea behind saliency map has been already explained in the autofocus subsection. In the analysed problem, a saliency map was generated through estimation to identify the most prominent areas of an image. This map highlights the regions of the image that are particularly attention-grabbing. Morphological opening is a two-step process that involves the application of erosion followed by dilation using a structuring element. The first one removes small features or details, while the second function enlarges the remaining structures. By using a structuring element of a specific size and shape, morphological opening can be used to remove small specularities and texture fluctuations in an image.

When taking into account the optical model, the light intensity for each pixel that reaches the observer is influenced by two primary factors: direct attenuation and veiling light. Since haze typically dominates hazy images, it greatly affects the overall image average value. The luminance of affected regions is assumed to increase proportionally with distance due to the linear increase of air light. Using this information, the average luminance value can be subtracted from the input image to improve low-contrast areas. Lastly, the CLAHE (Contrast Limited Adaptive Histogram Equalization) method is applied on luminance to increase the diversity of the image at a local level. This algorithm partitions the image into several blocks and improves the contrast of each one. The adjacent parts are subsequently combined by utilizing bilinear interpolation to eliminate artificial boundaries. The enhancement of contrast is only applied to uniform regions to prevent amplification of noise that could be present in the image. Moreover, the number of blocks can be fine-tuned, as well as the number of histogram bins and the clip limit.

Compared to other dehazing techniques the analysed method produces better results. The authors highlight that some of the other approaches oversaturate colors, introduce artifacts in regions with infinite depth, or have limitations with dense haze due to the statistical interpretation that requires variance to estimate the depth map. The measures to evaluate the results, which were used include indicators for edges, mean ratio of gradients at visible edges, and percentage of pixels that become black or white after restoration. The authors claim that their approach yields minimum values for the indicator representing mean ratio of gradients at visible edges when compared to other techniques. Moreover, it is effective for homogeneous hazy images, but has limitations with non-homogeneous haze. In such situation, the technique cannot completely remove the unwanted phenomenon. However it can moderately restore colors and visibility in some regions. Lastly, it is important to highlight that the simplicity of the proposed method allows for easy implementation and reduces the time requirements, as it does not require complex calculations like a depth map.

2.2.2. Deep learning in white balance

White balance editing neural network

To understand why white balance editing in sRGB images is difficult, it is helpful to know how cameras perform this function. Firstly, the colour of the lighting in the scene is estimated and that information is recorded as a raw-RGB vector. Next, each colour channel is divided in the image by a number based on that vector. After this procedure, a scaling operation is performed on each colour channel independently to normalize the illumination. This is done using a diagonal matrix with a size of 3x3. The result is a white-balanced raw-RGB image that is then further processed by the camera's ISP through several nonlinear operations, which makes it challenging to correct images with strong colour casts caused by camera white balance errors using traditional diagonal correction methods. The final output of the ISP is an image in the sRGB colour space.

To achieve precise post-capture white balance editing, it is necessary to correctly convert the displayed sRGB values back to their original raw-RGB values, and then re-display them. In the paper [3], a new deep learning framework has been developed for realistic post-capture white balance editing of sRGB images. It contains a single encoder network and three decoders targeting different WB settings, allowing for adjustment of incorrectly white-balanced sRGB images. Authors state that the method has been tested extensively, demonstrating its generalization capability to images outside of the training data and its state-of-the-art performance in both tasks. Such approach could also be applied to live view of digital camera to automatically correct the current settings of white balance.

Analysing the structure of multi-decoder, a U-Net architecture [46] with multi-scale skip connections between the encoder and decoders is utilized. The framework is composed of two

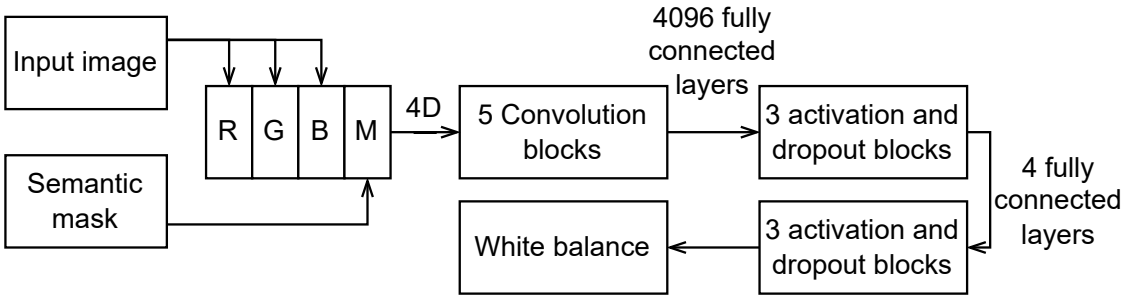

Figure 2.6: An overview of the modified semantic WB CNN architecture, based on [2]

primary units: a 4-level encoder unit responsible for extracting a multi-scale latent representation of the input image and a second unit with three 4-level decoders, each having a different bottleneck and transposed convolutional layers. The convolutional layers in the encoder and each decoder start with 24 channels at the first level and double the number of channels at each subsequent level. Therefore, the fourth level of each convolutional layer has 192 channels. The model was trained with dataset containing approximately 65 thousand of images. Moreover, another thousand was added in augmented pictures taken from the same dataset, but with random colour temperatures. Several images from the Internet were used to test the generalization capability of the approach. Results demonstrate the ability to produce compelling outputs compared to other methods and commercial software packages for photo editing, even when the input images have strong colour casts.

Semantic white balance using CNN

In another approach [2], the authors investigate the impact of utilizing a semantic mask in conjunction with the input image to train a convolutional neural network. This mask provides pixel-level semantic information to enhance the CNN's understanding of the image content.The images are represented as a 4D (Four Dimensional) volume instead of using the conventional three RGB channels. The first three dimensions of the volume indeed contain information about RGB colour channels, but the last dimension is the aforementioned semantic mask, which is assumed to be given. To accommodate the 4D volume, the modified AlexNet architecture [26] is used, which is shown in figure 2.6. The work demonstrates that making use of such semantic information improves accuracy of the illuminant colour estimation process resulting in better color balance outcomes.

Regarding used dataset containing around 22 thousand images, it was assumed that the white balance of all images is correct, despite being in the sRGB space, which is not ideal for white balancing. The decision to use this space was due to the lack of raw image datasets with semantic masks. In terms of data augmentation, 769 synthetic images were generated for each image in original dataset. These synthetic images were created by applying random incorrect white balance and gamma correction, as well as spatial data augmentation. In order to compare the effectiveness of the proposed method, the authors conducted a comparison with a regular AlexNet model [26] that did not utilize semantic information. To ensure a fair comparison, the same training setup was used. The results of this comparison revealed that the inclusion of semantic information led to a reduction of over 40% in the average root mean square error. Also the impact of semantic information on improving the results was investigated by feeding the trained model with a correct and a fake mask. The use of an incorrect one led to totally poorer outcome.

2.3. Edge detection

Edge detection is a fundamental image processing technique used to identify and extract the boundaries of objects or regions within an image. It analyses the pixel gradients to locate areas of rapid intensity transitions, which typically correspond to edges. Edge detection can be used for image segmentation and data extraction in areas such as image processing and computer vision.

2.3.1. Traditional machine learning approaches

Recognizing images of reinforced concrete damage

Natural processes can harm reinforced concrete structures. They can be damaged physically, by changes in pressure, stress and temperature, or chemically, by corrosive substances.This leads to cracks, spalling, and other issues that weaken the structure of concrete over time. To avoid any significant risks, detecting damage early is important and can prevent further problems. Manual inspections are subjective and time-consuming, so computer-based methods are being developed for more efficient and reliable detection. In the research paper [16] in order to detects concrete cracks, authors propose three machine learning approaches, such as MLH (Maximum Likelihood), SVM (Support Vector Machine) and RF (Random Forest).

Maximum Likelihood classification is a popular method used in remote sensing and image classification to assign class labels to pixels or image regions. It is a supervised technique that makes use of statistical probability distributions. At first, a training dataset is created, consisting of labelled samples where each one is associated with a known class. For each class, the statistical distribution of the pixel values or feature vectors are estimated using learning samples. The sample data is assumed to have normal distribution. During classification, the likelihood of a pixel or region belonging to each class is calculated based on its observed values or features. The likelihood is obtained by evaluating the probability density function of the corresponding class's distribution. The class label for the pixel or region is then assigned based on the class with the maximum likelihood. This method assumes that the statistical distributions estimated from the training samples accurately represent the probability distributions in the entire image or scene, and that the pixels are independently and identically allocated.

Support Vector Machine works by finding an optimal hyperplane that separates different classes or data points in a feature space. The goal is to maximize the margin, which is the distance between the hyperplane and the nearest data points from each class. These data points, known as support vectors, play a crucial role in defining the decision boundary. An ideal hyperplane position is in the middle of maximum margin between two support vectors of different classes, as shown in figure 2.7. SVM can be used for both linear and nonlinear classification models.

Random Forest is an ensemble machine learning algorithm that combines the predictions of multiple decision trees to produce a final outcome. It starts by randomly selecting subsets of the original training data, which is known as bootstrapping. These so called training samples can appear multiple times or not at all in a particular subset. Such technique is called sampling with replacement. For each subset, a decision tree is created independently by recursively partitioning the data based on features to make binary decisions. At each node of the decision tree, the algorithm looks for the best split. The criteria aim is to minimize the impurity or increase the homogeneity of the target variable within each resulting subset. The tree grows until a stopping condition is met, such as for instance reaching a maximum depth or minimum number of samples in a leaf node. Once the Random Forest is built, predictions are made based on the predictions of all the individual decision trees. For classification tasks, the most common prediction strategy is to use majority voting, where the class with the most votes from the trees is selected as the final prediction. In case of regression, the average or median of the individual tree predictions is

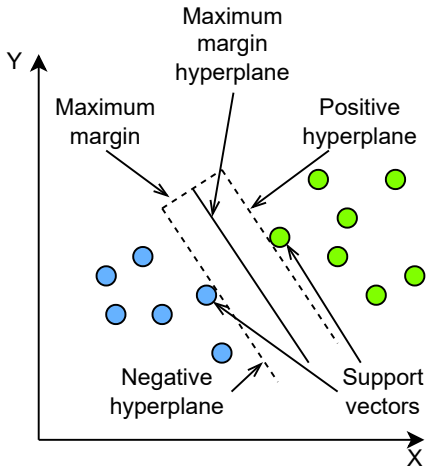

Figure 2.7: Concept of SVM

typically used. Combining predictions from different, independent trees introduces randomness and helps to reduce overfitting, which leads to more robust and accurate results.

In the analysed study, a dataset containing pictures of heavily damaged building was chosen to feed the considered models and assess performance and results. Three categories of cracks were introduced based on the cause of damage, such as spalling, exposed rebar and efflorescence. The images were divided into five types, depending on conditions in which they were taken. The factors influencing the division were: position of optical axis of the camera regarding the crack surface, lighting conditions and blur. For evaluation, 500 random samples were fed to each model. Outcome was compared using confusion matrix. The metrics were $Accuracy$, $Precision$, $Recall$, and $F1 score$, defined as follows:

- $Accuracy = (TP + TN)/(TP + FP + FN + TN)$
- $Precision = TP/(TP + FP)$
- $Recall = TP/(TP + FN)$
- $F1\ score = 2 * (Precision * Recall)/(Precision + Recall)$

where TP, TN, FP, TN denote, respectively, true positive, true negative, false positive, true negative outcomes.

The classification process was divided into two parts – training and testing (see figure 2.8). During the training, images from all types from previous division were labelled with the category of damage to obtain ground truth for assessment. Furthermore, original damaged images were divided into several regions based on type of the damage. Regions which were homogeneous

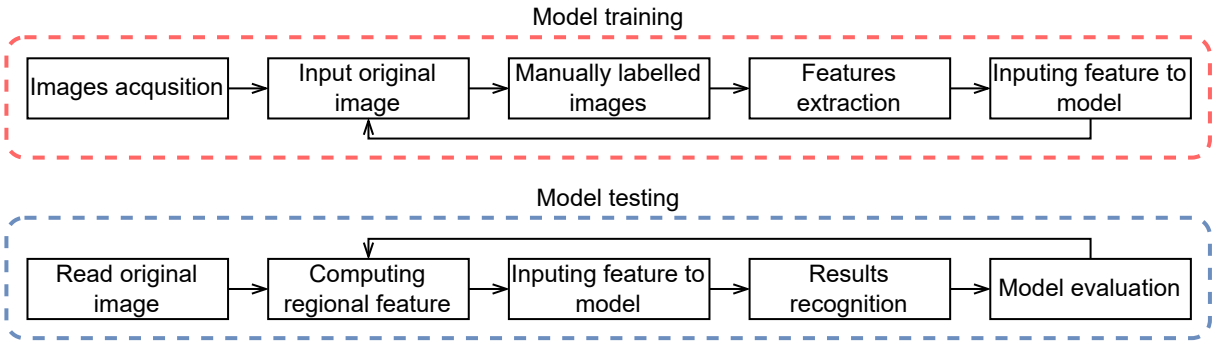

Figure 2.8: Process of supervised reinforced concrete damage classification

were selected and features from them were used as training pixels for the machine learning models. In the testing section, specific values based on the unique characteristics of each damage type were calculated from input pictures. From that, features of the damage could be extracted, which could be then compared to the features of respective cracks' types.

The results for all three classifiers were great for the first and second types, which were the ones with the best image acquisition conditions. However, for consecutive types, the outcomes dropped significantly. Regarding the four considered metrics, all the best results for individual types were scored by either MLH or SVM, but RF was not extremely worse. To compare numbers, F1 score is the metric taken into consideration for purpose of this work, because it combines precision and recall. In terms of percentage scores, the results of MLH and SVM dropped from in-between 92%-97% for first and second types to around 52%-57% for types four and five.

To conclude, MLH and SVM were comparable in performance with slightly different performance based on the image type and evaluation metric. Therefore, the RF algorithm gave the worst predictions in this application.

Surface defects detection for mobile phone panel workpieces

The problem described in [24] revolves around examining mobile phone panels during the manufacturing process. To reduce the cost of manually detecting imperfections in each workpiece and improve performance of already known approaches, the authors propose a Mobilephone Panel Surface Defects Detection (MPSDD) framework. The tasks involved in this method are image cutting and standardisation, features extraction, blocks separation, defects classification, and detection results

In the first section, the backgrounds of images were removed, which in this case were homogeneously green. Then, the pictures were resized to fixed dimensions. Lastly, gray scaled images were filtered by median and mean filters.

Regarding the approaches proposed by authors, detection of surface defects could be done in two ways. The first one was to pass the whole image as an input to the classifier, which is faster but less accurate. The second option giving better performance locally is dividing the image into blocks and inspecting each of them separately. To obtain feature vectors following the latter approach, extractions were done using LBP (Local Binary Pattern) and HOG (Histogram of Oriented Gradients). This generated data is essential to be provided for machine learning models. LBP compares the values of central pixel with the neighbouring ones. It assigns a binary value to each neighbour based on whether their intensity is greater or lesser than the central pixel. These binary values are then stored in a vector, which can be also easily represented by a decimal value. By analysing these numbers across an image and constructing a histogram, LBP captures local texture information. On the other hand, HOG quantifies local gradient directions and magnitudes to capture the shape and structure information of objects. After calculating the gradients of image pixels with techniques such as Sobel operator, the picture is divided into small cells. Within each of them, a histogram of gradient orientations is computed, having the angle of edge on X axis and value on Y axis depending on magnitude for that direction. To improve robustness against variations in illumination and contrast, neighbouring cells are grouped together into blocks. The histograms within each block are concatenated and normalized. The normalized histograms from all blocks are concatenated to form the final feature vector, which represents the distribution of gradient orientations across the image.

To resolve the problem, authors proposed two classifiers: the SVM and NB (Naive Bayes). The latter one is based on the Bayes' theorem and assumes that the features are conditionally independent of each other given the class label. During the training phase, the algorithm learns the probability distribution of the features given each class label. It calculates the prior probability of each class label and the likelihood probabilities of the features. During the classification phase,

it then calculates the posterior probability given the observed features using Bayes' theorem. It selects the class label with the highest posterior probability as the predicted class.

For the experiment, the data was labelled either as qualified workpieces or the ones with defects. Four classifiers were created, connecting the two initial SVM and Naive Bayes each with both LBP and HOG. Lastly, the prediction was run for whole images, as well as for blocks mode. In case of whole images, all four combinations achieved similar, satisfactory results of around 92% to 98% of right predictions. There were tiny gradual drops with increase of test data percentage. However, for the smaller blocks' mode, the performance of LBP features especially combined with Naive Bayes significantly dropped to around 65%-70% regardless of test data size compared to 90% and higher for other classifiers. It is important to note that regarding the speed of detection, LPB with NB achieved the fastest time of training and detection. However, for the time metric there was significantly bigger difference between NB and SVM than the one between LBP and HOG.

To conclude, for whole images all the presented approaches accomplished good results. The best prediction rate can be obtained using HOG with SVM combination. The drawback of this method is long training and prediction times, but the latter one should still be enough in this particular application. For problems where time of detection is crucial and should be as short as possible, LBP with NB or HOG with NB approaches should be considered.

2.3.2. Crack detection and recognition based on CNN

In the paper [11] authors again proceeded to solve problem of crack detection and recognition. However, in this approach they did not focus on so called traditional machine learning methods, but instead they tried to implement a CNN and improve currently achievable results.

The proposed neural network consists of convolutional, pooling, activation, batch normalisation and dropout layers (see figure 2.9).

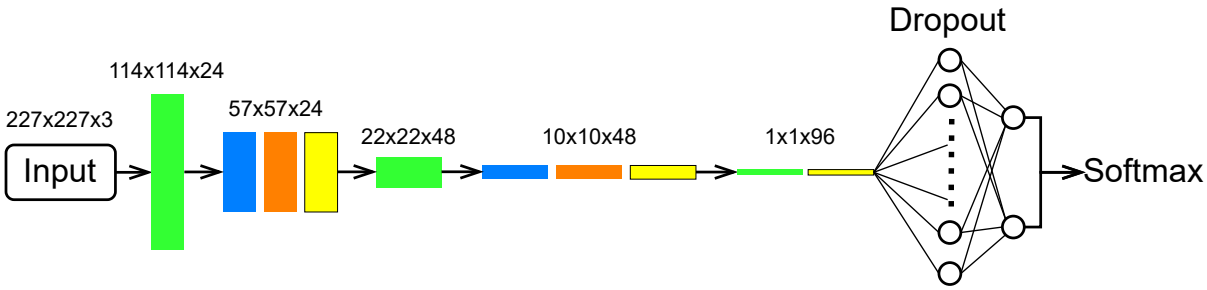

Figure 2.9: Architecture of crack detection and recognition DNN

Outputs from different layers are represented by colours: green for convolutional, blue for pooling, orange for BN (Batch Normalisation), and lastly yellow for ReLu. In this work, as a criterion for pooling layers, the maximum value pooling was used. Authors state that due to high contrast between cracks and background, convolutional kernel and pooling size can be large and still achieve satisfactory results. Therefore, for both convolutional and pooling layers, stride of 2 was applied. After each pooling, batch normalisation was implemented. It normalises the output from activation function as shown in equation:

$$\hat{x}_i = \frac{x_i - \mu}{\sigma} \tag{2.1}$$

where μ is the mean and σ is standard deviation.

After that, it multiplies this output by an arbitrary parameter and additionally adds another arbitrary parameter to the product of that multiplication (see equation below:).

$$y_i = \gamma \hat{x}_i + \beta \tag{2.2}$$

These calculations set new standard deviation and mean for the data. The four parameters: mean, standard deviation, and the two arbitrary parameters are all trainable, meaning that they can be optimized during the training process. Because of that, weights within the networks are not imbalanced with extremely high or low values. Additionally, it increases the training speed and stability of the model and may reduce overfitting to some extent. Regarding optimiser, adaptive moment estimation (ADAM) was used. After being passed through all defined layers, some nodes are dropped randomly and proportionally in order to avoid overfitting. Then the data is subjected to the softmax function, which transforms the values into a probability distribution, deciding on the final outcome.

The dataset used for the experiment was split into two groups, first one containing cracked images, and the second one with the crack-free. The total number of pictures was 40000. During the training process, the model was evaluated every 1000 iterations using verification set. The model achieved accuracy of 99.6% quite quickly – after ten thousand of iterations. The training stopped automatically after 110 thousand of samples with the best accuracy of 99.71% throughout the experiment. The authors highlight that in case of analysing relatively straightforward tasks, using large convolution and pooling methodology with lesser number of layers might lead to better results.

Chapter 3

Programming digital cameras - case study

3.1. Digital camera as an embedded system

Many information systems for data acquisition, monitoring, quality control, etc. use so-called embedded systems, designed specifically for capturing, processing and recording digital images. A special case of an advanced embedded system is a digital camera. Apart of optical and mechanical parts it consists of an image sensor, image processor, memory, firmware and other components working together consistently to provide users with a high-quality digital imaging experience. Moreover, such system is designed to handle a wide range of input and output operations, for instance receiving user inputs, adjusting camera settings, controlling the lens, and displaying the captured images on a screen. Embedded systems' design metrics, which can be cost, size, performance, or power, are usually tightly constrained. The reason for this is to obtain cheap and small enough device with performance needed for specific application [52].

Nowadays, there is a wide variety of digital cameras. However, to obtain its basic functionality, there is a set of common tasks that should be performed. The input for embedded system of digital camera usually is light entering through lens, aperture and shutter. Then, the light signal should be converted into image representative digital signal and saved into a memory. The storing space should be easily accessible in order to transform saved data for usage in the system. The exact way of implementation and other functionalities may differ based on a camera's model. F. Vahid and T. Givargis [52] propose image compression and auxiliary pixel coprocessor to aid rapid display of images. Additionally, they suggest a memory chip controller and a DMA (Direct Memory Access) module to enable the access without requiring the use of a microcontroller. For external communication, there is also an UART (Universal Asynchronous Receiver-Transmitter) for transmission of files to a PC (Personal Computer). Moreover, an LCD (Liquid-Crystal Display) is provided for displaying the images. The central component of this system is a microcontroller that functions as a processor responsible for managing the functions of all other circuits. While individual devices are designed to carry out specific tasks, the microcontroller is a more versatile processor engineered for general purpose.

Regarding the type of digital camera, commonly used are DSLR (Digital Single-Lens Reflex) ones containing a mirror reflecting the light signal through pentaprism into pathfinder, where photographer can see the image. When capturing an image, the mirror is moved of the way, which results in the image sensor being directly exposed to the light signal. Recently gaining popularity and being under considerable development are mirrorless cameras. In this approach, the disposal of mirror allows reducing size and weight of cameras [8]. A generic system architecture is presented in figure 3.1. The light signal is represented by dashed-line, meanwhile

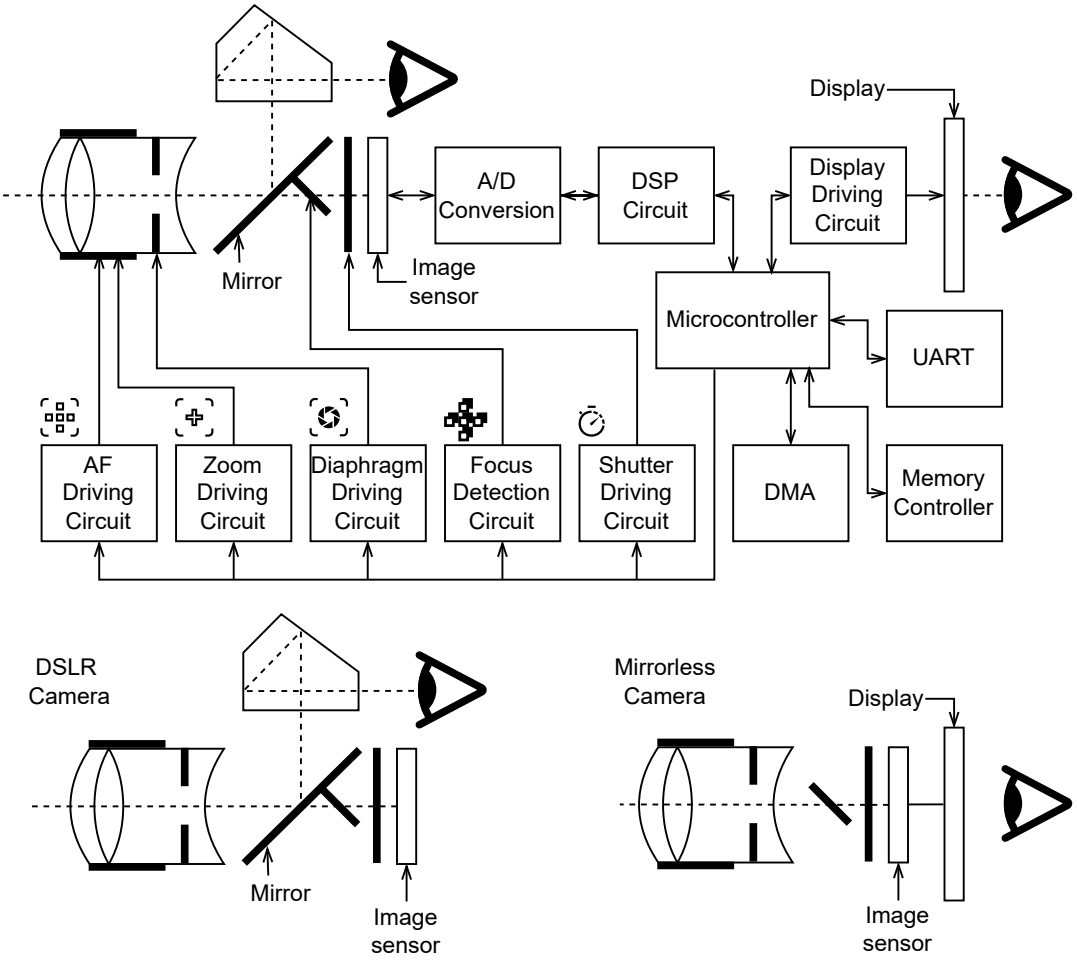

Figure 3.1: An overview of embedded digital camera system architecture

continuous line corresponds to digital signal. In the bottom section of the figure, the difference between DSLR and mirrorless construction type is explained. It is important to notice that video and audio signals are not represented in this schematic.

The construction of digital cameras, as well as their architecture, are usually protected by the manufacturers and are subject to patent law. One of the Canon patents published in last few years [21] is related to the use and storage of multiple parallax images for computational photography. A Quad Pixel Autofocus technology is defined and the challenges connected to this technology of combining four composite images to obtain final result.

Firmware installed in main controlling unit typically allows controlling effectors such as the lens - for zooming and focus adjustment, aperture – to provide the required amount of light which affects depth of field, or shutter – to expose an image sensor to the light for a set time. Such control allows taking artistic photos with full control over composition and exposure. Usually, creative options such as portrait, landscape, sports, etc. are based on the camera's best parameter settings stored in memory for given shooting conditions. The firmware automatically selects the settings. Sometimes, as is the case with Canon's open-source user developed firmware add-on Magic Lantern, the functions responsible for settings are included in the API (Application Programming Interface) provided to the programmer. This opens the way to expand the capabilities of digital cameras.

3.2. Image sensors

When light enters the camera lens, it passes through a series of optical elements, such as lens, aperture and shutter, that focus it onto the image sensor. The image sensor then converts the light into an electrical signal by detecting photons that strike its surface. Each pixel on the sensor generates an electrical charge proportional to the intensity of light. In other words, the basic function of image sensor is to convert light into electricity by exploiting the photoelectric effect. These imagers are usually composed of CCD (Charge-Coupled Device) or CMOS (Complementary Metal-Oxide-Semiconductor). They use a capacitor to measure the accumulated signal charge, which is then converted to voltage. In the case of CCDs, the voltage is then amplified by nodes before being sent for further processing. These devices work by sequentially moving charge packets across their surface in order to transport all the signal to the output node. On the other hand, CMOS imagers convert the signal from charge to voltage within each pixel, and output voltage signals individually, when selected by row and column busses [29]. The brief idea behind structure of CCD and CMOS is presented in figure 3.2.

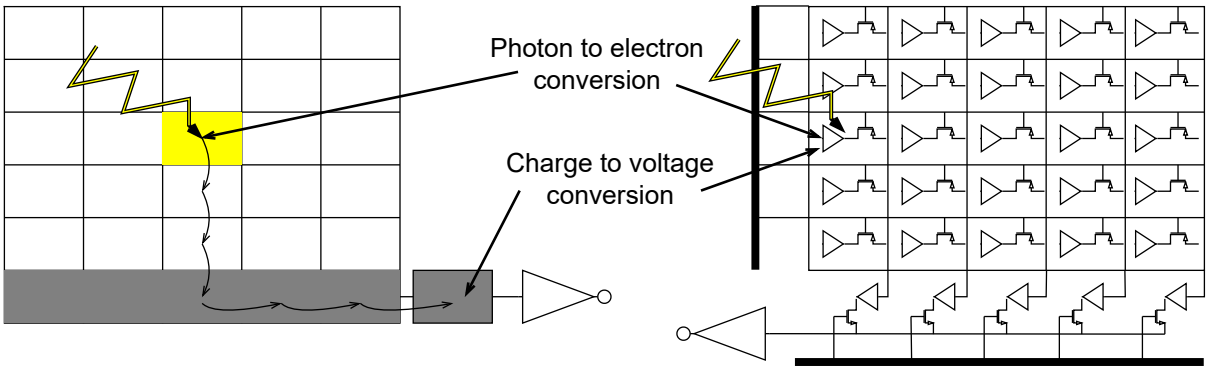

Figure 3.2: CCD vs CMOS structure, based on [29]

3.2.1. CCD architecture

CCDs can be designed in three different ways [50]: full frame, frame transfer, and interline transfer as it is shown in figure 3.3.

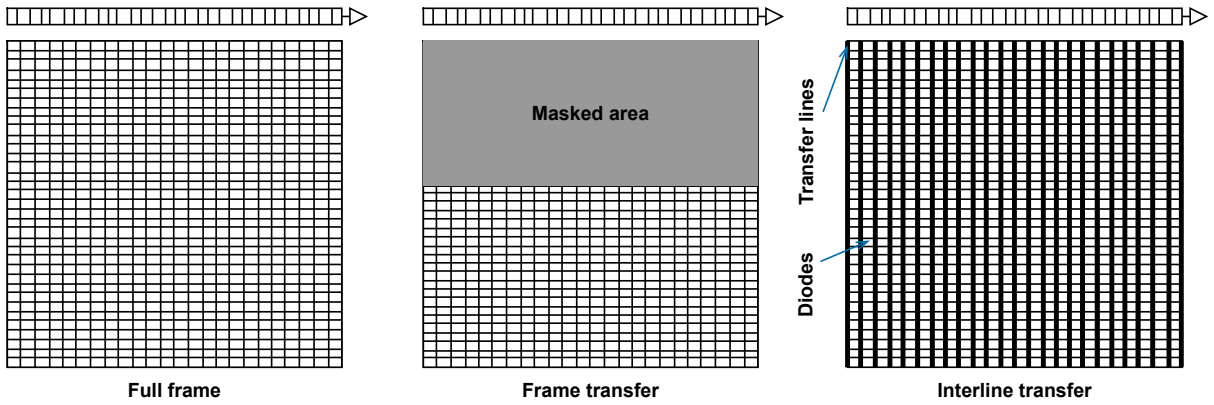

Figure 3.3: Three architectures of CDD, based on [50]

Full-frame CCDs have no gaps between the photosensitive areas but require a mechanical shutter to operate. It opens to allow the charge to accumulate and then closes to transfer and

read it out. Although full-frame devices have the largest photosensitive area, their operation is limited by the speed of the mechanical shutter, charge transfer and readout steps. Therefore, they are best suited for applications requiring long exposure time.

Frame-transfer CCDs are similar to full-frame CCDs but are divided vertically into two regions. The lower region is photosensitive, and the upper region is masked. After the charge accumulation, it is quickly moved vertically to the masked region. Then, the stored charge is systematically shifted and read out while the next image is being integrated.

Interline-transfer CCDs have a charge-transfer channel next to each photosensitive region, allowing stored charge to be shifted horizontally into the transfer channels. It is then moved vertically for readout, allowing short integration periods and electronic exposure control.

3.2.2. CMOS architecture

Every CMOS pixel is responsible for converting its collected signal charge into voltage, but there are differences in design. Eric. R. Fossum [19] discusses pixel circuit architectures division into passive and active pixels.

In a **passive pixel** CMOS image sensor, each pixel consists of a photodiode and a transistor. The photodiode absorbs the incident photons and generates an electron-hole pair. The generated electrons are collected in the potential well created by the transistor, which acts as a charge storage capacitor. The primary issues with passive pixels are their readout noise level and scalability, as they do not scale well to larger array sizes or faster pixel readout rates. The reason for this is that increased bus capacitance and quicker readout speed both can result in further increase in readout noise.

In an **active pixel** approach, there is an active amplifier in each pixel, which is only activated during readout. Because of that, power dissipation is usually less than in CCDs. Such solution trades off pixel fill factor to improve performance compared to passive pixels that use an in-pixel amplifier. Typically, in active approach, pixels are designed for a fill factor of 20-30%, similar to interline-transfer CCDs. This leads to increase in signal-to-noise ratio and dynamic range due to a reduction in read noise, even though there is a loss in optical signal, which can be slightly recovered by using microlenses.

3.2.3. CCD and CMOS performance characteristics

Both CCD and CMOS image sensors have their problems and limitations [29].

Dark current noise which is known as the accumulation of undesired charge in the CCD pixels due to natural thermal processes that occur at any temperature above absolute zero.

Smear is an error source that can occur when charge packets are shifted through the imaging region of the CCD towards the output node. During this readout time, the generation of electrons from photons continues, which can corrupt charge packets from one part of the image with additional electrons from another.

Blooming is an effect that occurs when a potential well becomes full of electrons during the integration period. When there is an overflow, the excess electrons flow into adjacent wells, resulting in a region of saturated pixels. If blooming is not controlled, the resulting image may have large, overexposed areas.

Fill factor refers to the percentage of each pixel that is sensitive to light. In an ideal scenario, the fill factor should be 100%. However, it is frequently less than this due to the need for electronics to control blooming and additional control elements for CMOS sensors, which occupy space within each pixel and make these regions insensitive to light.

Quantum efficiency is the efficiency with which incident photons are detected. Some photons may not be absorbed due to reflection or may be absorbed where the electrons cannot be

collected. The quantum efficiency is calculated by dividing the number of detected electrons by the product of the numbers of incident photons and electrons that each photon can generate.

The **charge transfer efficiency** is a measure of the percentage of electrons that are lost at each stage during the charge transfer process. Current buried channel CCDs have this efficiency exceeding 99.99%.

3.2.4. CFA

A colour image of a scene requires a minimum of three sensors, but using multiple detectors in a camera is not an ideal solution due to cost and design challenges. It is the most expensive component of a digital camera, making up a significant percentage of the total cost. Therefore, another approach is taken in order to reduce price and maintain the functionality [45]. A single sensor camera is preferred, where the sensor's surface is covered with a mosaic of coloured filters known as a **CFA (Colour Filter Array)**. Each filter is designed to allow only a specific range of wavelengths of light to pass through and be detected by the sensor underneath. As the pixel size is smaller than the significant details of the image, estimating missing colour values requires examining neighbouring pixels, a process known as CFA interpolation or "demosaicing", which is analysed in later section of Firmware's functions. Most commonly, a Bayer pattern [9] of CFA is used, described in figure 3.4. This pattern provides a mosaic of Red (R), Green (G), and Blue (B) colour components. It has twice as many green samples as red or blue, because the human eye is more sensitive to changes in green than the other two colours.

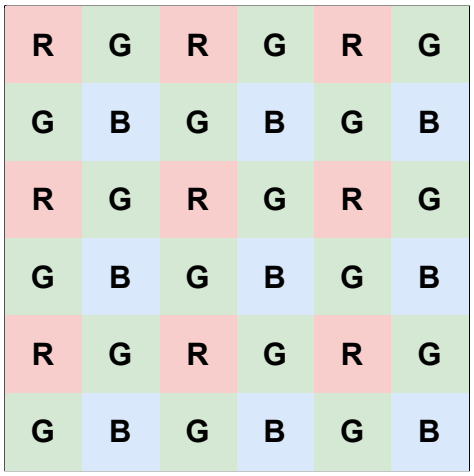

Figure 3.4: Bayer CFA pattern, based on [9]

3.2.5. Conclusion on image sensors

When comparing CCD and CMOS under standard lighting conditions with no motion, it is difficult to point out any difference in picture quality. The problem for CMOS image sensors appears in poor lighting conditions, when they tend to exhibit more noise than CCD sensors. It is due to sensor substrates with less on-chip circuitry. On the other hand, CMOS sensors do not suffer from smear when capturing fast-moving objects. While CCD technology is mature in terms of yield and performance, CMOS offer the advantage of lower power consumption, allowing the design of more compact cameras with similar picture quality to larger CCD-equipped ones. Furthermore, CMOS sensors enable the rapid readout of a large number of pixels, making them important for progressive HD (High Definition) imaging. In single CMOS sensor implementations, it is theoretically possible to read out R (red), G (green), and B (blue) signals simulta-

neously. While CCDs provide superior image quality and flexibility, they come at the expense of system size and remain the most suitable technology for high-end imaging applications in broadcast television and digital cinematography. In the table 3.1 several features were chosen to sum up some crucial characteristics of CDD and CMOS image sensors.

Table 3.1: CCD and CMOS chosen features comparison, based on [29]

Feature	CCD	CMOS
Signal out of pixel	Electron packet	Voltage
Signal out of chip	Voltage(analog)	Bits(digital)
Signal out of camera	Bits(digital)	Bits(digital)
Fill factor	High	Moderate
Amplifier mismatch	N/A	Moderate
System noise	Low	Moderate
System complexity	High	Low
Sensor complexity	Low	High
Relative R&D cost	Lower	Higher

3.3. Processing components

3.3.1. General characterisation

The technology of processors is related to the design of the computational engine implemented to execute the intended functions of a system. Although the word "processor" commonly refers to software processors that can be programmed, there are also various other digital systems that can be considered as processors despite being non-programmable. These processors are specialized in specific applications, such as digital cameras, and therefore have different design characteristics. Based on [52], processing components can be divided into three groups: general-purpose, application-specific or single-purpose.

The **general-purpose processors** are should be suitable for diverse applications. That universality might successfully increase the number of devices sold. As the exact use of the processor is not known in stage of development, some extensions such as a program memory, larger register file or arithmetic-logic units might be typically added for various possibilities. Presented approach is often referred to as software implementation, as there is no emphasis on matching hardware to the application. Besides flexibility, which is the main advantage, also the design time is short, because only a program is needed. In small quantities, the unit cost of a processor may be reasonably low since the manufacturer distributes the one time price of a product development project over numerous units by selling them in large quantities to other clients. For computation-intensive applications, performance can be excellent if a fast processor is used that features advanced architecture. However, there are also some disadvantages in terms of design metrics. The unit cost for larger amount may exceed a reasonable level, and also certain applications may experience poor performance. Additionally, the size and power of the processor may be needlessly large, because of unnecessary for specific purpose processor hardware.

In case of **single-purpose processors**, their function is to execute exactly one program. It is achieved mostly by designing a custom digital circuit. In the opposite to general-purpose, a single-purpose processor does not need a program memory. It can also have a simpler data path and controller since it does not have to support a large instruction set, which results in a smaller size. Regarding the performance, since the processor is customised for a specific application, it can be faster. The improvement comes from lack of requirement for using intermediate registers

when passing data or from excluding program memory fetches. On the drawbacks side, the design time and non-recurring engineering cost may be relatively high due to customising the device which also means limited flexibility. Such approach is often referred to as hardware implementation. Single-purpose processors are typically used as coprocessors and accelerators.

An **Application-Specific Instruction-set Processor (ASIP)** can be used as a compromise between aforementioned processors. It is designed for a specific category of applications that share common features. Regarding this work, a suitable example could be digital signal processing. Data path in ASIP can be customised for the particular implementation category, integrating special functional units for commonly used operations and removing less frequently used elements. ASIP can assure a good trade-off between flexibility and satisfying performance, power and size. The thing that can be discouraging is large non-recurring engineering cost of processor and compiler in case if no similar implementations are already applied.

3.3.2. Digital Signal Processor

DSPs belong to the category of ASIP. They are specifically designed to execute typical operations on digital signals, which represent analog signals like audio and video. These operations are commonly signal processing tasks such as filtering, transformation, or combination, and often involve math-intensive computations like multiplication, addition, or shifting. DSPs use special components in their hardware, such as a multiply-accumulate unit, to execute these operations quickly with just one instruction. Since the processor's programs frequently manipulate large arrays of data, a DSP might also include dedicated hardware to fetch data from memory in parallel with other operations, further improving execution speed [52].

Focusing strictly on the field of computer vision and image processing applications, DSP chips are usually less expensive than many other hardware accelerators used in portable devices. A lot of applications of this discipline uses sequential algorithms, and DSPs are built to handle such tasks. Multicore digital signal processors have been designed to add the ability to implement parallel processing of large, computationally demanding sub-tasks by multiple processing units for algorithms that are not too complex. Furthermore, the development time for algorithms in single core DSPs is relatively short and their energy consumption is low. On the other hand, multicore digital signal processors are not the best option for high data throughput and high speed applications. Even though they offer parallelism, in large extent, it is not optimal direction of work for them. To sum up, DSPs offer an energy-efficient and affordable solution for embedded systems and mobile/portable devices where computational requirements are not high-end, and the power consumption is crucial [23].

3.3.3. Field Programmable Gate Arrays

The FPGA chip contains arrays of reprogrammable logic gates that can be changed to do different tasks. Unlike DSPs or CPUs (Central Processing Units), FPGAs do not have a set structure or central processing unit already built in. To make the them work for a particular task, its logic gates need to be connected in a certain way. This hardware setup needs to be changed, every time a new algorithm is to be used. That's why FPGAs are often called reconfigurable devices. It is important to notice, that they are programmed using different languages than for CPUs or DSPs. In this case, low-level hardware description languages are used, which tend to be more complicated, especially for beginners. In terms of performance, FPGAs usually have faster processing speeds compared to other types of hardware accelerators. Their parallel and reconfigurable nature is a valuable feature that makes them suitable for creating high-performance hardware solutions for various applications. Nevertheless, excellent performance does not mean high energy consumption, as FPGAs are labelled as one of the highest processing power to power

consumption ratio according to [42]. Regarding the drawbacks, they are known for their long development time. Moreover, developing efficient codes for FPGAs can be difficult. Despite the availability of new tools designed to simplify and speed up the process, programmers still require a certain level of technical expertise to create reliable and effective codes. According to [40], FPGAs have also found use in implementing neural networks. Even though it is also possible to apply those networks in DSPs, the well-performing parallelism and efficient integer arithmetic of FPGAs can be advantageous in accelerating the training and use of neural networks. However, it can be difficult to use them for nonlinear operations and feedback loops in this application [23].

3.4. Firmware

3.4.1. Definition

According to [28], the definition of firmware is not universally fixed, but generally refers to software that is embedded in a product and is typically stored in ROM (Read Only Memory) or EEPROM (Electrically Erasable Programmable ROM). Firmware is not intended to be modified by customers under normal use. However, it is considered distinct from software that cannot be changed by customers, such as Microsoft Windows. Therefore, firmware can be user-upgradable, but tends to be more implicit and is rather associated with embedded systems. It is usually tightly coupled to the device's hardware and is designed to be highly reliable. Additionally, in some devices, this software acts as an operating system, allowing various additional functions to be run. In the case of digital cameras, the firmware may control features connected to digital image processing. Additionally, it can physically affect components such as lens, aperture or shutter to perform functions like zooming, focus adjustment, depth of field or exposition time.

3.4.2. Functions

As firmware is responsible for defining actions performed by processing units on acquired images, it has a wide variety of functions. In Texas Instruments application report [44], a DSP platform was programmed to acquire, process, compress and store image from digital still camera. A view of image pipeline describing whole process is shown in figure 3.5. The first step after obtaining the CFA signal from CCD is converting it to the digital form using ADC (Analog to Digital Converter). Next operations are strictly connected with affecting the appearance of image.

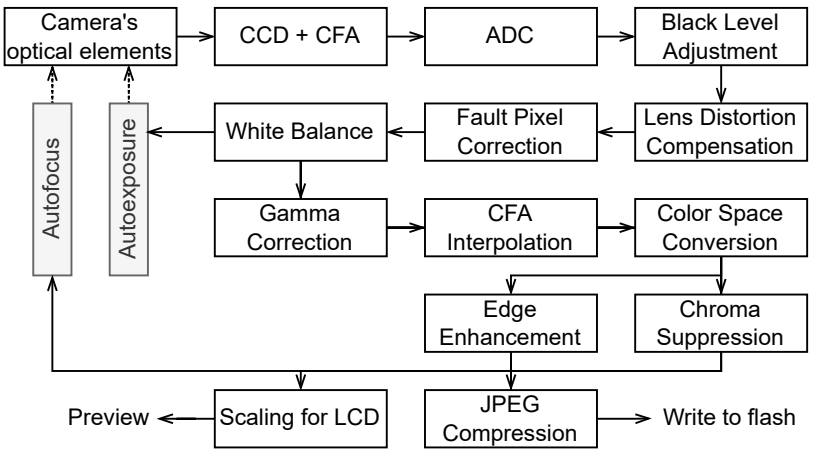

Figure 3.5: DSC Image Pipeline, based on [44]

Black Level Adjustment In order to enhance the dynamic range of the pixel values captured by the CCD imager, it is necessary to correct the black pixels, where due to imperfections of the sensor, non-zero current is recorded. To correct this issue, the black clamp function subtracts an offset from each pixel, effectively adjusting the values. However, to prevent negative outcomes, the function clamps or clips the value to zero.
Lens Distortion Compensation The quality of the lens used to capture an image can result in nonlinearities in the brightness, causing a decrease in this parameter from the center of the image towards the edges. To recompensate this issue, the brightness of each pixel is adjusted based on its location within the image.
Fault Pixel Correction CCD arrays with large pixels may contain defective pixels. To compensate for these missing values, an interpolation scheme is used to estimate and provide data values for the missing pixels at each location in the image.
White Balance When capturing a scene, the lighting conditions can be different from these when viewing the image. This can lead to a colour cast, such as a blue tint on a person's face or a reddish tint on the sky. Additionally, the sensitivity of each RGB channel can vary, causing neutral colours to appear inaccurate. To correct these imbalances, the gain of the red, green, and blue colour channels is adjusted. This is achieved by calculating the average brightness of each channel and determining a scaling factor. However, since the lighting conditions are often unknown, a common technique involves balancing the energy of the three colours, which requires an estimate of the imbalances.
Gamma Correction The mapping between image gray value and displayed pixel intensities is nonlinear in display devices used for image-viewing. To address this issue, the gamma correction is used to compensate for the variations between the images produced by the CCD sensor, and these presented on a screen.
CFA Interpolation The CFA only provides information about one colour pixel (R, G, or B) at a specific location. Therefore, in order to obtain full colour resolution, the missing colour pixel information is filled by interpolation with the neighbouring pixels.
Color Space Conversion Because image-compression algorithms usually handle the YCbCr color space, a conversion from RGB is needed. Each Y, Cb, and Cr value at a pixel location is converted into a weighted sum of the corresponding R, G, and B values, which represents a linear transformation.
Edge Enhancement CFA interpolation filters create a low-pass filter that can cause smoothing of image edges. To counteract this effect, an edge detector is applied in the image pipeline. By calculating the edge magnitude in the Y channel for every pixel and scaling it, the detector enhances the sharpness of the image when it adds it to the original luminance (Y) image.
Chroma Suppression It is important to note that the process of enhancing edges is specifically applied only to the Y channel of the image. As a result, the color channels (Cb and Cr) may become misaligned at the edges, leading to the appearance of rainbow-like artifacts. This effect can be reduced by suppressing the Cb and Cr color components at the edges.

Not only can firmware change the view of captured image, but it is also possible to use it to manipulate camera's components. As aforementioned, controlling effectors such as lens, aperture, or shutter opens a wide variety of opportunities to automatically choose favourable settings in order to obtain possibly the best quality of a picture. In currently analysed example, there are two functions of such purpose and they are performed in closed-loop feedback manner to provide sufficient information to the components to be set.

Autofocus This feature automatically adjusts the focus of the camera's lens to obtain a sharp image of the subject. This mechanism applies image processing to assess the quality of the lens focus, and then iteratively adjust the lens motor until the image is sharply focused.

Regarding autofocus, Canon's EOS (Electro-Optical System) mirrorless and newer DSLR cameras use Dual Pixel CMOS AF as their autofocus system when shooting in Live View mode [49]. This unique system employs image plane phase detection technology that enables all pixels on the image sensor to perform both phase detection and imaging functions. This results in several advantages, including excellent image quality without compromising AF performance, providing coverage of up to 100% of the image area, maximization of light information from fast lenses, and quick, smooth focusing and tracking during both still and video shooting.

In the Dual Pixel technology, each pixel has two photodiodes. When performing phase detection, the information from both of them is read independently and then compared. Each of the two points produces a parallax image, which captures a different perspective of the subject. By analyzing the amount of blur in these images, the AF system calculates the difference between them and determines how to adjust the lens position to align the images and bring the subject into focus. During imaging, the information from both photodiodes is merged and read as a single complete readout.

Autoexposure To achieve high-quality images that are consistent across varying scene brightness, controlling the exposure of the CCD is essential. It is achieved by detecting the average brightness of the scene and adjusting the CCD exposure time and/or gain accordingly.

JPEG Compression The last presented function of this image pipeline is connected to memory saving. As the space for storing images is limited, image compression allows to put more images in the same size memory. For most cameras, it is popular to use standardized JPEG compression, and usually a ratio of around 10:1 to 15:1 is achieved.

The crucial functions described so far most likely repeat in many applications. They can be spotted in the proposal [25] of a CMOS image signal processor for digital still cameras. The authors of this work described a new algorithm Joing demosaicing, see figure 3.6.

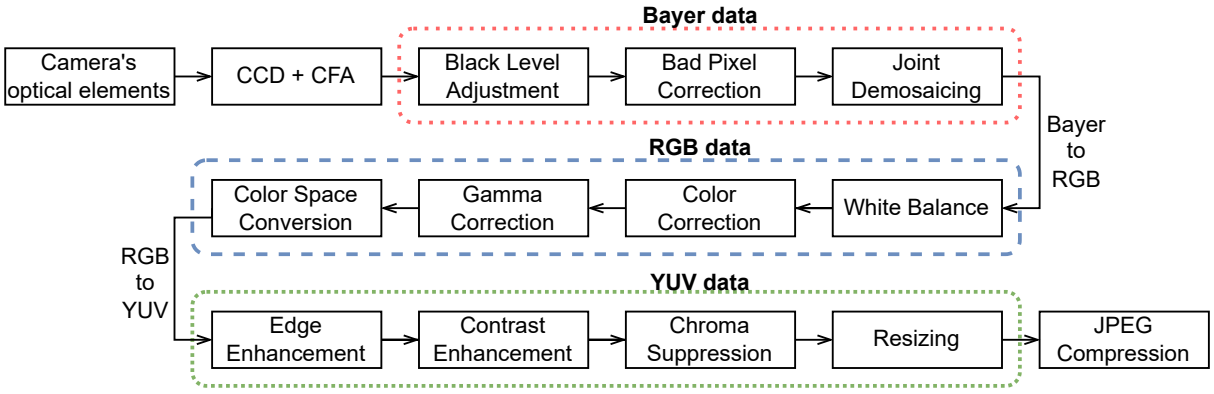

Figure 3.6: CMOS Image Signal Pipeline, based on [25]

Joint demosaicing It is an interpolation technique, which similarly as CFA interpolation, is used to reconstruct a full-colour image from a single colour-filtered image sensor. Joint demosaicing involves simultaneously estimating the missing colour values of neighbouring pixels using statistical models, such as sum of Gaussian component. This results in better colour accuracy, reduces artifacts such as colour fringing and noise for Bayer format image. This algorithm typically requires more computational resources than CFA interpolation, but it can produce higher-quality images with less noise and higher resolution.

3.4.3. Custom firmware and cross-compilation

Custom firmware can be a modified version of the original firmware that runs on a digital camera's hardware or its extension. It is typically created to add features or functions that are not available in the original implementation. Modifying the code can also remove limitations or restrictions imposed by the native version, allowing the camera to perform in ways that were not supported by the manufacturer.

A very special case of a custom firmware is Magic Lantern (ML) [33]. This open-source framework, licensed under GPL (General Public License), enhances the features of Canon DSLR cameras, operating as an independent program alongside with Canon's original software. It allows developers to create extensions while not touching any part of original, proprietary code.

Magic Lantern is deployed to and then loaded from a memory card. Available ML functions can be accessed and customized under the following main menu items: Audio, Expo, Overlay, Movie, Shoot, Focus, Display, Prefs, Modules, Debug, Help, Modified (see figure 3.7).

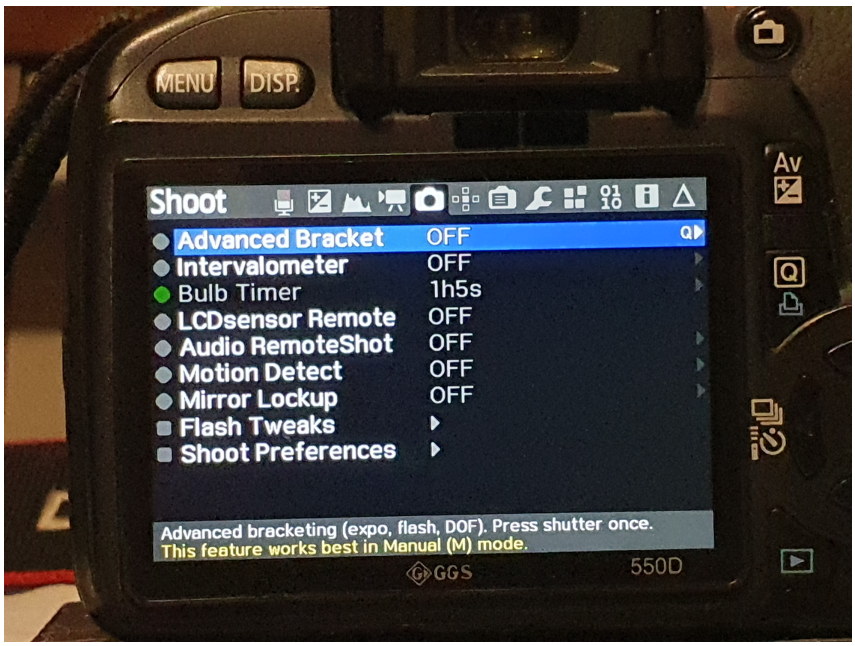

Figure 3.7: Content of ML Shoot menu item displayed on LCD screen of Canon 550D camera

Any implemented custom firmware that contains broadened or additional functions needs to be compatible with the source system architecture. On the other side, more convenient for programmers would be to develop new features in other systems than the ones running on digital cameras. To address these problems, the **cross-compilation** is a very practical solution.

The task of a compiler is to convert a high-level programming language, designed for human programmers, into a low-level machine language that computers can understand and execute without external assistance. The use of a high-level programming language has a significantly increases the speed of program development. One of the main reasons is because the syntax used in high-level programming languages is closer to human problem-solving methods compared to machine language. Additionally, the compiler will scan for and flag any obvious programming errors that it detects during this translation process. Lastly, such programs tend to be much shorter than assembly code equivalents [35].

The compilers can be divided into two categories, depending on the environments they work for. If the compiler generates code for the same platform on which it runs, it can be described as a native compiler. It optimizes code for the specific processor and operating system of the host machine, resulting in faster and more efficient code execution. It is also easy to use since it does

not require any additional setup or configuration. On the other hand, cross compilers generate code for different platform than they are running on, which is the main case of interest regarding compilers during this work.

A cross compiler operates on a host machine to generate machine code for a specific target system. In the case of embedded systems, two compilers are required: one for the host computer, which performs tasks like development, design, testing, and debugging, and another compiler known as a cross compiler, which generates machine code for the embedded system's processor. The cross compiler produces an object file that is designed only for the host machine, so it must reconfigure its settings from host specific to embedded system specific to generate the appropriate machine code [35].

An example of compiler infrastructure that supports development of compilers for multiple source languages and targets is GCC (GNU Compiler Collection) [20]. It is an open-source collection distributed under General Public License. It supports several source languages and can compile code for a lot of platforms, processors, and operating systems. The free access contributed to its large popularity, and due to that – flexibility of use.

The GCC compiler architecture is shown in figure 3.8. The program is composed of a dependent on the source code front end, independent on the source code back end, and a machine description. Each source language supported by GCC has its own syntax analyzer and can generate the same syntax tree for different source languages using the parsers. The goal of a parser is to determine whether the input text is syntactically valid and, if so, to build a data structure represents the input text's structure. The back end converts the syntax tree representation into an intermediate code called RTL (Register Transfer Language). This code is then subjected to optimization passes to improve its performance before generating the assembly language output file. The machine description contains a target micro file and a machine description file that separately describe the target operating system and the target architecture [22].

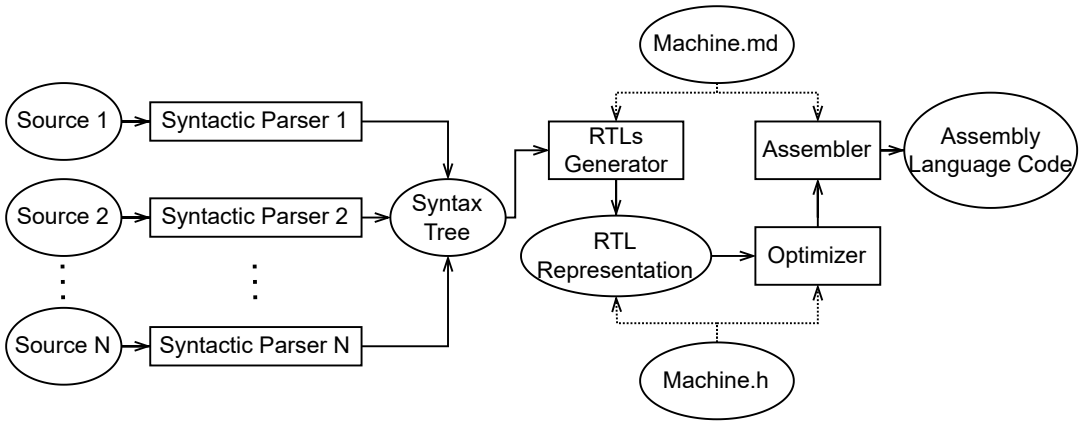

Figure 3.8: The GCC architecture, based on [22]

Different source languages, processors, and operating systems have unique features and requirements, which demand specialized techniques for describing the target environment, constructing code generators, and creating interfaces between the target environment descriptions and code generators. One of such techniques would be developing a suitable target description capable of describing both the general and specific aspects of systems and hardware. Moreover, creating an intermediate representation at an appropriate abstract level, which can be mapped into multiple source languages, adapted to different platforms, and optimized. Lastly, it is important to establish a uniform interface between target environment descriptions and code generators. GCC provides corresponding techniques to address these issues, which support the efficient development of compilers for multiple source languages and targets [22].

To sum up, with use of cross compilers it is possible to develop new features in well-suited environment and then convert them to work in desired source platform. Most embedded devices use ARM (Advanced RISC Machine) processors architecture due to its low power consumption and high efficiency. In the case of Canon digital cameras, the same approach is taken. The RISC (Reduced Instruction Set Computing) emphasizes simplicity in instruction execution. Unlike embedded devices, most PCs are based on CISC (Complex Instruction Set Computing), in which instructions can perform multiple operations and manipulate various types of data in a single set. Because of that, to be able to develop the firmware and build RISC binaries in different environments than the native to digital camera, the use of ARM cross compilation is mandatory. The Unix toolchains usually follow naming convention [architecture]-[vendor]-[OS]-[ABI]. The architecture refers to target platform, vendor specifies the tool chain supplier, and the OS indicates target operating system, if there is any on the platform. The ABI (Application Binary Interface) refers to which application binary interface convention is being used which ensures that binaries generated by different tools can inter-operate. For installation of Magic Lantern, a gcc-arm-none-eabi was implemented. This GNU tool chain [6] supports C and C++ languages, contains integrated and validated packages, has no vendor, does not target any specific operating system, and follows the ARM EABI (Embedded Application Binary Interface). This interface defines how functions interact considering how data is represented in memory. Therefore, all object files of features building the new firmware are cross compiled to generate respective files for ARM architecture.

When working with images, programmers might use help of already defined libraries such as OpenCV. This popular open-source tool provides a wide range of functions and algorithms that allow loading, saving and manipulation of visual data. However, any additional external libraries cannot be implemented in ARM platform without specifically building them for this environment. Because of that, again cross compilation tools are significantly helpful for the integration. Moreover, tools like Tengine not only allow multiple platform usage, but also provide low-level optimization and light weight [37].

3.5. Implementation examples

3.5.1. Kodak DC40

The architecture of Kodak DC40 camera described in [41] is shown in figure 3.9. Designers implemented an 8-bit ADC to amplify and digitise output signal from CCD image sensor. A 2 MHz pixel rate clock controls the conversion circuit. The images are stored temporarily in a buffer memory before being compressed by a programmable digital signal processor and saved in Flash EPROM memory. Then, the images can be downloaded to a host computer by a serial interface

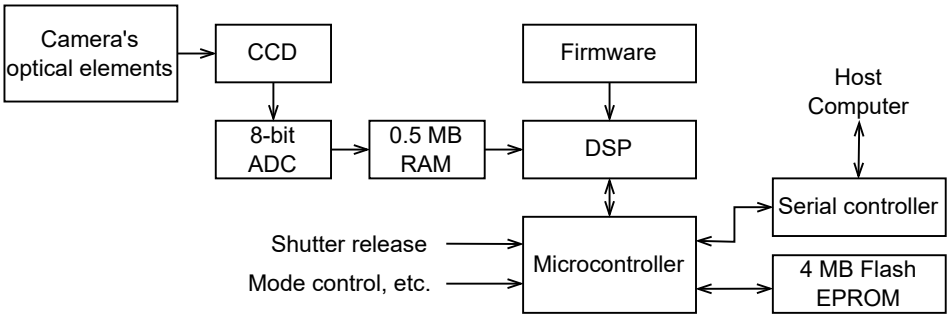

Figure 3.9: Kodak DC40 Camera block diagram, based on [41]

link. The solution is quite simple and straightforward, as it was designed almost 30 years ago. However, it is an example of system performing basic functionality of digital camera.

3.5.2. Kodak EOS-DCS 5

The author of description [41] states that exposure time, the lens and aperture are controlled automatically in this camera. Therefore, a slightly more advanced example is to be analysed. The output of CCD image sensor is similarly converted to digital signal, but in this case by 12-bit ADC, and the conversion circuit is controlled by a 5 MHz pixel rate clock. One of the biggest differences is lack of DSP in the architecture. Instead, a programmed by the camera firmware Programmable Logic Arrays module is implemented, which processes the output signal from ADC. The data is then temporarily stored in a 16 MB DRAM (Dynamic Random Access Memory). Sequently, the images are converted to a format compatible with TIFF (Tagged Image File Format), which is a file format for storing high-quality images. It allows lossless compression and handling them with a high dynamic range. Then, the data is saved on a removable PCMCIA-ATA card which often occurs to be a type of hard drive, as shown in picture 3.10.

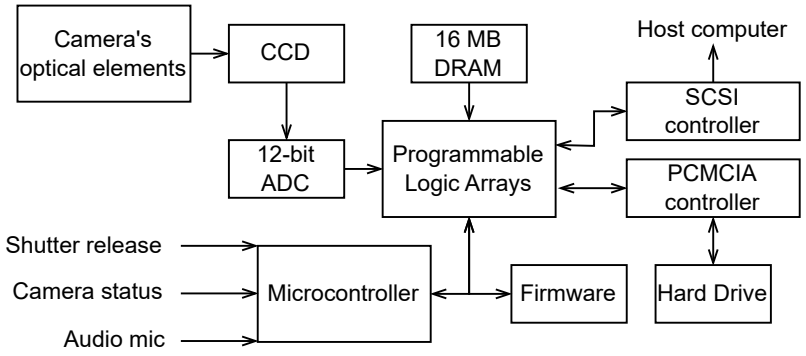

Figure 3.10: Kodak EOS-DCS 5 Camera block diagram, based on [41]

This interface allows the connection of ATA (Advanced Technology Attachment) devices, such as hard drives, to a computer via a PCMCIA (Personal Computer Memory Card International Association). Another option of external communication is achieved by a SCSI (Small Computer System Interface), which connects the camera to a host computer. Such solution allows downloading the images to PC. This model is overall further advanced than Kodak DC40, with more peripherals and better performance. However, also in this case, the design was done almost 30 years ago. Nevertheless, analysing this implementation, a slightly different approach of digital camera architecture can be studied.

In both Kodak cameras, designers resigned from operating on analog signals. To reduce noise and allow for more advanced image processing algorithms, cameras perform digital processing. Moreover, only a part of algorithms is applied to images by firmware before storing them. More complex computations are performed on the host computer after downloading the data. This allows the cameras to use advanced image processing algorithms that are easily modifiable for specific customer needs and can be upgraded conveniently while in use [41].

3.5.3. System-on-a-Chip for Digital Still Cameras

Okada et al. in [36] developed an advanced, small-sized, and energy-efficient system-on-a-chip, enabling a digital camera to execute all its required functions on a single chip. One of the main

requirements for the device is to allow high-quality still image and video clip shooting. Moreover, it should have high speed sequential shooting and a compact body. The chip architecture design is shown in figure 3.11.

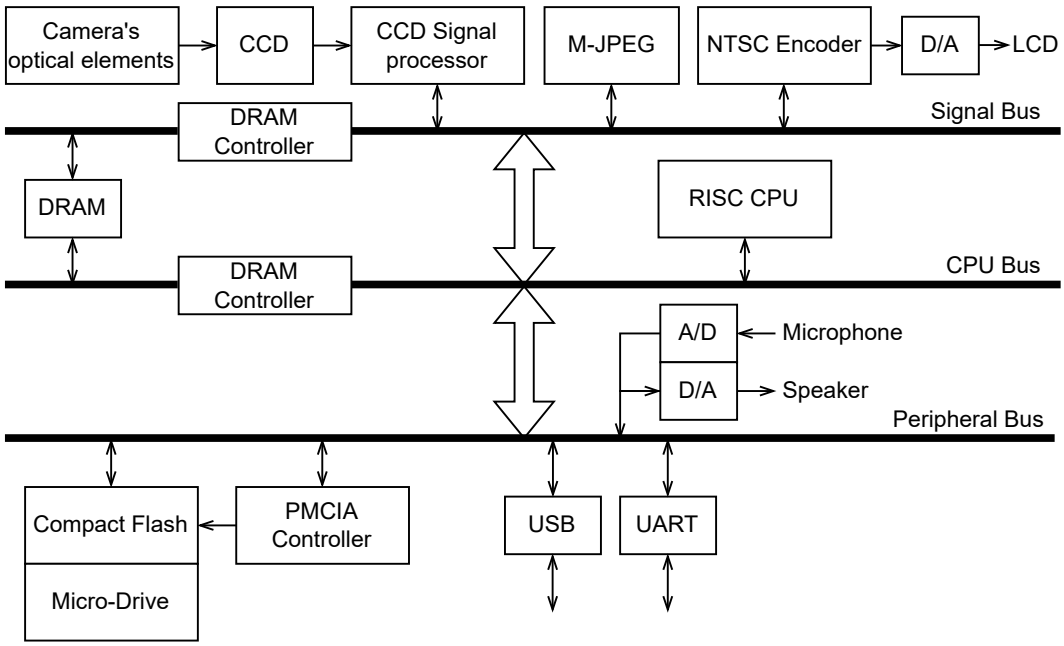

Figure 3.11: System-on-a-Chip for Digital Still Cameras architecture, based on [36]

Data provided by CCD image sensor is processed with correction functions and converted to YUV format, in which is it stored in DRAM. NTSC (National Television System Committee) format encoder enables the image to be seen on an LCD. After the shutter is released, the data is transferred to the Motion-JPEG codec and compressed, and then again stored into DRAM. Sequently, it is converted into file format data by the RISC-CPU (Reduced Instruction Set Computing). It allows recording the file on external storage. In this approach, a 32-bit RISC CPU performs crucial processing operations in the circuit. It operates with maximum frequency of 57 MHz, has an instruction and a data cache, each of 4 KB. The three-bus architecture allows processing high-resolution images in real time. The signal bus, which has a transfer rate of 228 MB/s, connects to the image processing devices, and is characterized by the highest transfer rate among the buses. The CPU bus is the second bus, and it serves as a local bus for the CPU, allowing it to directly access the DRAM. The third bus connects the peripheral circuits that operate at low speed. Such implementation mitigates the risk of any conflicts with slow data.

The peripheral part of the system is well developed, as it supports many interface standards. Describing the most crucial one, the PCMCIA interface allows communication with the micro drive. Moreover, a USB and UART interfaces are included, which are commonly used standards. Architecture of this system is different from previously analyzed ones, as it does not feature neither a digital signal processor nor programmable logic arrays. Instead, designers decided to use a RISC CPU. Nevertheless, they managed to achieve desired integration and performance of the circuit.

3.5.4. A 12-bit 4928 x 3264 pixel CMOS signal processor

Lastly, a much newer solution is presented. A 4928 x 3264 pixel CMOS image signal processor SoC (System-on-a-Chip) is proposed. The design aims for high performance and low complexity. The image pipeline of this system was already analysed and is shown in figure 3.5. Therefore, this subsection is focused on analysing the used hardware.

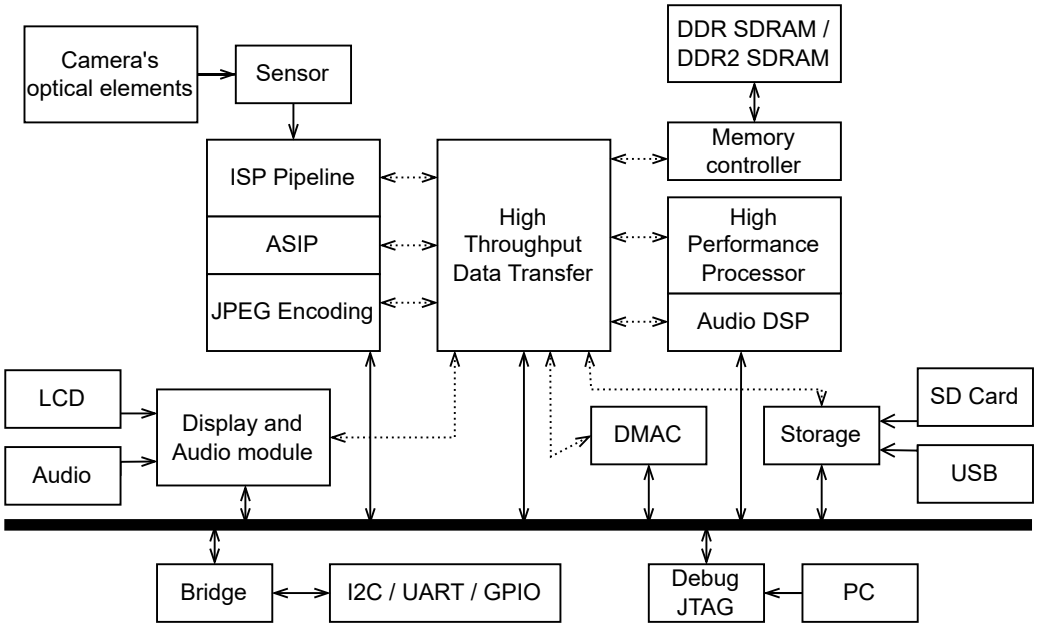

Figure 3.12: CMOS System-on-a-Chip architecture, based on [25]

For processing purposes, a main unit implemented in the design is a high performance processor. Jin el al. [25] discuss that a dedicated pipelined processor is the most suitable option for low power and small size requirements. An instruction driven and reconfigurable processors were also taken under consideration. To implement algorithms in a pipeline structure, an ASIC (Application-Specific Integrated Circuit) is also adopted. Interestingly, an DSP is provided for audio signal processing exclusively. The architecture is shown in picture 3.12. Dotted lines represent control signals and black lines are for data stream. The System-on-a-Chip acquires unprocessed image data directly from the lens and sensor. This raw data is then passed to be processed in the ISP (Image Signal Processor) pipeline before being transmitted to either the JPEG or video processing module. Additionally, the SoC offers a variety of input/output interfaces. The images can be displayed after resizing on an LCD.

For storage purposes, a SD (Secure Digital) card module is provided, where both raw and JPEG compressed data can be collected. Moreover, a connection to camera can be established via USB or UART. For debugging purposes, a JTAG (Joint Test Action Group) standard protocol can be used. Regarding the memory of the system, a DDR SDRAM (Double Data Rate Synchronous Dynamic Random Access Memory) of 1st and 2nd generation with a memory controller module were applied. The authors state that in contrast to other approaches, the suggested ISP demonstrates the highest throughput with the smallest space occupied. Meanwhile, its power efficiency remains almost unchanged compared to other options. The designers believe, that with its combination of low power consumption, high throughput, and small chip area, the proposed ISP shows great potential as a viable solution for both modern digital cameras and smartphones.

To sum up, the presented System-on-a-Chip is the most sophisticated of the analysed designs. With increased demand on available features and performance, it becomes harder to create a well-composed circuit. Nevertheless, with enough development, it was manageable to obtain a large scale integration system fulfilling its purpose.

Chapter 4

Base for improvements

4.1. Existing ML features

Magic Lantern offers several already implemented functions. Most of them can be customized through parameterisation. These functions can also be accessed programmatically. In the next parts of this section, the chosen available features are described. Every one of them can be accessed from the already mentioned Magic Lantern main menu, by selecting proper sub-menu and toggling or turning them on (see figure 3.7).

4.1.1. Exposure parameters

Exposure in digital cameras can be controlled in many ways. Magic Lantern extends proprietary firmware by offering possibility of customizing white balance, ISO, shutter speed, aperture size, picture and video styles using presets or automatic adjustments for specific environments.

In photography, ISO refers to the level of sensitivity of the camera's sensor to light. The lower the ISO, the lower the sensitivity to the light. It is one of the three key parameters of exposure, alongside aperture and shutter speed. This parameter is particularly useful when shooting in low-light conditions, as it enables photographers to use faster shutter speeds and smaller apertures while still achieving proper exposure. By enabling exposure override, non standard values of these parameters can be set allowing higher flexibility and experimenting.

4.1.2. Overlay

In this subset of functions, users can enable the following Magic Lantern overlay graphics. Every feature has its threshold and auxilary parameters adjustable by user.
Zebras enables zebra stripes, which show underexposed or overexposed areas of current view.
Focus peak shows which part of the image is in focus by indicating high-contrast lines with blinking dots. This feature is a focus assist function.
Magic zoom displays a zoom box for focus assist. Also shows a confimation when it believes that a perfect focus has been achieved.
Cropmarks allows using custom grids and cropmarks for framing and composition.
Ghost image shows a transparent overlay of chosen image.
Defishing uses rectilinear projection to show normal view of image captured with a fisheye lens.
False color shows different luma levels, which is a the weighted sum of gamma-corrected RGB components, using a color map.
Spotmeter measures brightness from a small spot in the frame.

Histogram and waveform - by those tools, a distribution of image brightness levels are shown. **Vectorscope** shows color distribution with and U-V plot. Can be practical for color grading.

4.1.3. Movie recording

Designers also propose some specific functions for movie mode.

Bit rate controls bitrate used for video recording.

Time indicator displays a small time counter in the upper right corner, which can indicate either duration of the current clip, approximated recording time remaining on the card or approximated recording time until reaching 4 GB of data.

Movie restart - movie recording is restarted automatically. However, it is important to notice, that a few seconds will be skipped during restarting.

Notifications such as red crossout, displaying when photographer forgets to press record, message indicating if camera is in recording of stand-by mode, beeping when recording starts or stops or lighting a blue LED.

Movie recording key starts/stops movie recording by half-pressing the shutter button.

Shutter lock - locks the shutter value in movie mode.

HDR allows shooting videos in high dynamic range by changing ISO value every other frame.

FPS override for all video modes. This section offer also several other image effects such as negative image or swapping U-V (red becomes blue).

4.1.4. Still shooting

Functions belonging to this category are mostly suitable for still shooting, however some of them can also be used for movies.

HDR Bracketing is a technique used to capture a high dynamic range image by taking multiple shots of the same scene at different exposure levels. The resulting HDR image is able to show a wider range of brightness and detail than a single exposure could capture. After turning on this function, a photographer needs to take only one shots, and Magic Lantern automatically continues the sequence.

Intervalometer takes pictures at fixed time intervals. User can define the delay between shots, which is up to 8 hours, and the number of pictures to be taken.

Bulb/focus ramping enables the gradual adjustment of exposure in a timelapse sequence, which helps to compensate for the transition from daytime to nighttime.

Bulb timer is a very useful function for night shots and astrophotography. It allows to take very long exposures, as the shutter remains open until it receive a signal to close.

LCD sensor remote is a feature that allows the user to trigger the shutter using an external device or sensor. This can be useful in situations where the photographer needs to trigger the shutter from a distance or when using a tripod, to avoid camera shake caused by manually pressing the shutter button.

Audio remote shot takes a picture or starts recording a movie when a loud noise is detected.

Motion detect working in two modes. Reactive to exposure change - takes action when the level of brightness significantly changes, which is meant to detect large object in the view of camera. The other mode is frame difference, which computes the difference between last two frames in luma channel. Such approach can detect smaller movements which do not change exposure.

Silent pictures takes picture in LiveView mode without moving the mirror.

Mirror lockup allows the user to lock up the camera's mirror before taking a photo. When the mirror flips out in normal conditions after pressing the button, such movement can cause vibration or camera shake, especially when using longer shutter speeds. By using the mirror lockup feature the amount of vibration is reduced and can result in sharper images.

Flash tweaks allow the photographer to control the amount and quality of light that is emitted by the flash to achieve the desired exposure and lighting effects.

4.1.5. Focus

Features especially connected to camera's focus. **Trap focus** enables the camera to automatically release the shutter when subject comes into focus. This feature is particularly useful for shooting moving subjects or subjects that are difficult to track with autofocus, such as wildlife or sports photography.

Focus patterns are custom shapes which allow the photographer to select the area of the image where the camera should focus. Such patterns can be used either with AF or trap focus.

Follow focus allows to keep a moving subject in focus by adjusting the lens as the subject moves closer or farther away from the camera.

Rack focus triggers a continuous focus shift between two previously set points in the camera's view.

Stack focus shoots a series of photographs of the same subject with different focus points. If the images are combined, the result has a much greater depth of field than would be possible with a single exposure and is in focus from front to back.

Focus distance function computes and shows distance to the focal point of current settings of optical components.

Hyperfocal distance shows the largest depth of field possible for the current f-number of aperture. It is the distance between the camera and the nearest point in the scene that appears to be in acceptably sharp focus when the lens are focused at infinity. This means that by setting the focus at the hyperfocal distance, the photographer can ensure that the maximum possible area of the scene is in focus. Additionally, developers also offer displaying nearest and farthest distances in which objects appear in focus.

4.1.6. Display

Lastly, a set of available features regarding display section is described. There is a set of upgrades for LiveView mode, such as contrast and saturation adjustment. These functions do not affect recording. Nevertheless, they might help the photographer when shooting in specific conditions. In case of contrast, it is useful when focusing with very flat picture styles. On the other hand, saturation reduces distraction by color. Additionally, there is a feature increasing brightness, making LiveView usable in very dark scenes, where normally the display would be almost totally black.

Color scheme also changes brightness and colors, but in this case of the on-screen information like menu.

Upside-down mode might become useful in certain conditions where camera needs to be mounted upside-down.

4.2. Programming opportunities

The Magic Lantern add-on is under continuous development. To make the interaction with the original firmware easier and provide a set of rules and standardisations, the contributors created APIs for C/C++ [14, 13] and Lua [31] languages. These tools define the methods and data formats that developers can use to access and utilize the functionality provided by Canon. It results in better integration and allow to use already pre-built modules, libraries, and functions for creating own implementations and features. More over, there exists also emulator on which developers can test some of the results of their work.

4.2.1. Buttons handling

Button codes are divided into two sets. In one of them, buttons are defined as constants as in listing 4.1. In this case, they are camera dependent and are received by `gui_main_task`. This handler retrieves events from the main message queue and calls a function from a jump table, where one of the elements is `GUI_control` responsible for button presses. On the other hand, another set of codes is received by `menu_handler` resulting in buttons having other effects depending on the menu that user is currently in, as shown in listing 4.2. This set is the same or almost the same across different Canon cameras. In general, some buttons can send and event when press, as well as when unpressed. For those, which can be only detected when pressed, it is possible to differentiate between short and long hold. To obtain full range of functionality, it is destined to use both sets. However, in certain cases some buttons send the same events, which makes it difficult to differentiate between them. Additionally, the codes may also vary depending on camera states. As for example, arrows cannot be detected when ISO adjustment mode is active. Magic Lantern offers a function simulating button press, which can be easily used for further software development.

Listing 4.1: Button constants

```
#define BGMT_PRESS_LEFT 0x1c
#define BGMT_UNPRESS_LEFT 0x1d
#define BGMT_PRESS_UP 0x1e
#define BGMT_UNPRESS_UP 0x1f
#define BGMT_PRESS_RIGHT 0x1a
#define BGMT_UNPRESS_RIGHT 0x1b
#define BGMT_PRESS_DOWN 0x20
#define BGMT_UNPRESS_DOWN 0x21
#define BGMT_PRESS_SET 0x4
#define BGMT_UNPRESS_SET 0x5
#define BGMT_TRASH 0xA
#define BGMT_MENU 6
#define BGMT_INFO 7
#define BGMT_Q 8
#define BGMT_Q_ALT 0xF
#define BGMT_PLAY 9
#define BGMT_PRESS_HALFSHUTTER 0x3F
#define BGMT_LV 0x18
#define BGMT_WHEEL_LEFT 2
#define BGMT_WHEEL_RIGHT 3
#define BGMT_WHEEL_UP 0
#define BGMT_WHEEL_DOWN 1
```

Listing 4.2: Button actions for menus

```
#define BGMT_AV (event->type == 0 && event->param == 0x56 && (
              (is_movie_mode() && event->arg == 0xe) ||
              (shooting_mode == SHOOTMODE_P && event->arg == 0xa) ||
              (shooting_mode == SHOOTMODE_AV && event->arg == 0xf) ||
              (shooting_mode == SHOOTMODE_M && event->arg == 0xe) ||
              (shooting_mode == SHOOTMODE_TV && event->arg == 0x10)) )
#define BGMT_AV_MOVIE (event->type == 0 && event->param == 0x56 && (
    ↪ is_movie_mode() && event->arg == 0xe))
#define BGMT_PRESS_AV (BGMT_AV && (*(int*)(event->obj) & 0x2000000) == 0)
#define BGMT_UNPRESS_AV (BGMT_AV && (*(int*)(event->obj) & 0x2000000))
#define BGMT_FLASH_MOVIE (event->type == 0 && event->param == 0x56 &&
    ↪ is_movie_mode() && event->arg == 9)
#define BGMT_PRESS_FLASH_MOVIE (BGMT_FLASH_MOVIE && (*(int*)(event->obj) &
    ↪ 0x4000000))
#define BGMT_UNPRESS_FLASH_MOVIE (BGMT_FLASH_MOVIE && (*(int*)(event->obj)
    ↪ & 0x4000000) == 0)
#define BGMT_ISO_MOVIE (event->type == 0 && event->param == 0x56 &&
    ↪ is_movie_mode() && event->arg == 0x1b)
#define BGMT_PRESS_ISO_MOVIE (BGMT_ISO_MOVIE && (*(int*)(event->obj) & 0
    ↪ xe0000))
```

4.2.2. Camera settings

Changing parameters like exposure time, aperture, iso, focus, white balance and many more is provided by built in functions available for programmers (see listing 4.3). Those properties are

variables in DryOS, which is a pre-emptive scheduled multithreaded RTOS (Real Time Operating System) running in the camera. They are used to exchange information between tasks. By setting up callback functions, other tasks can get notifications about value changes applied in function using `PROP_HANDLER` macros, working similarly as interrupts. Such usage from `lens` library is shown in listing 4.4. Programmer can access all the available properties exploring corresponding header filer.

Listing 4.3: Lens setting functions

```
static inline void lens_set_aperture(unsigned aperture){
    prop_request_change( PROP_APERTURE, &aperture, sizeof(aperture) );
}
lens_set_shutter(SHUTTER_250);
lens_set_ae(APERTURE_2_8);
lens_set_iso(ISO_1600);
```

Listing 4.4: Property handler

```
PROP_HANDLER( PROP_ISO ){
        const uint32_t raw = *(uint32_t *) buf;
        lens_info.raw_iso = raw;
        lens_info.iso = raw/2 < COUNT(iso_values)
                ? iso_values[ raw / 2 ]
                : 0;
        return prop_cleanup( token, property );
}
```

4.2.3. Customizing menus

Menus are divided into top level menus which can be added by `menu_add` function, and lower level menu entries. The latter ones can be split into three types. The first being simple menu items displaying a fixed text and executing a function when selected. Next, there are binary entries, setting on/off corresponding functions. Lastly, items with user-defined text and action, which need to be provided with functions both for displaying text and performing an action. Simple menu entries are demonstrated in listing 4.5. The first entry calls `focus_show_a` function when the menu should be displayed and `focus_reset_a` when selected. The second one displays `Rack focus` and calls `focus_toogle` when selected. `Menu_print` is a basic print function from menu source file, which can be used as template for creating more developed tasks. Having defined the entries, to then register a menu, it is needed to call `menu_add` when initializing a new task.

Listing 4.5: Menu entries

```
static struct menu_entry focus_menu[] = {
        ...
        {
                .display        = focus_show_a,
                .select         = focus_reset_a,
        },
        ...
        {
                .priv           = "Rack focus",
                .display        = menu_print,
                .select         = focus_toggle,
        },
        ...
};
```

4.2.4. Useful tips

Using Canon API functions Magic Lantern uses the possibility to call Canon functions by name. It can be done due to an existing API in the Canon firmware which allows to register, execute and un-register event procedures – functions associated with a text name. When calling a function name, the return is either -1 in case if it is not registered, or the return value of that function.

Tasks Magic Lantern provides a `TASK_CREATE` macro for creating new tasks for each feature, which run in parallel with the Canon firmware. In the arguments user can specify the name, entry (pointer to a function with this signature), priority, flags, and an additional argument, which is most commonly unused. Tasks can be also overridden with `TASK_OVERRIDE` macro, which might be useful in replacing running functions with own implementations. Another use of this macro is killing a task by overriding it with an empty one.

File input/output Interacting with files is controlled by `fio_ml` library containing functions responsible for most necessary operations. Some of the prototypes are shown in listing 4.6.

Listing 4.6: File interaction functions

```
FILE* FIO_Open(const char * name, unsigned flags);
FILE* FIO_CreateFile(const char * name);
FILE* FIO_CloseFile(FILE* file);
int FIO_GetFileSize(const char * filename, unsigned * size);
int FIO_WriteFile(FILE* file, const void * buf, size_t len_in_bytes);
int FIO_ReadFile(FILE* file, void * buf, size_t len_in_bytes);
void FIO_RemoveFile(const char * filename)
```

Audio The API grants interaction with audio from the device by reading its raw level from chosen channel (0 – left, 1 – right). Canon 550D camera contains a AK4646 chip, or a compatible one. Developers designed compatible functions presented in listing 4.7, which are wrappers to the DryOS API code, allowing reading and writing to the audio chip using audio registers shown in listing 4.8. Moreover, the audio can be amplified with chosen gain.

Listing 4.7: Audio constants

```
#define AUDIO_IC_PM2     0x2100        #define AUDIO_IC_MODE4   0x2F00
#define AUDIO_IC_SIG1    0x2200        #define AUDIO_IC_PM3     0x3000
#define AUDIO_IC_SIG2    0x2300        #define AUDIO_IC_FIL1    0x3100
#define AUDIO_IC_ALC1    0x2700        #define AUDIO_IC_HPF0    0x3C00
#define AUDIO_IC_ALC2    0x2800        #define AUDIO_IC_HPF1    0x3D00
#define AUDIO_IC_IVL     0x2900        #define AUDIO_IC_HPF2    0x3E00
#define AUDIO_IC_IVR     0x2C00        #define AUDIO_IC_HPF3    0x3F00
#define AUDIO_IC_OVL     0x2A00        #define AUDIO_IC_LPF0    0x6C00
#define AUDIO_IC_OVR     0x3500        #define AUDIO_IC_LPF1    0x6D00
#define AUDIO_IC_ALCVOL  0x2D00        #define AUDIO_IC_LPF2    0x6E00
#define AUDIO_IC_MODE3   0x2E00        #define AUDIO_IC_LPF3    0x6F00
```

Listing 4.8: Audio interaction functions

```
int16_t audio_read_level(int channel)
uint8_t audio_ic_read(unsigned cmd)
void audio_ic_write(unsigned cmd)
```

Screen Using `bmp` library, users can print custom objects, as well as text on the LCD screen with specified color and font as presenten in listing 4.9. The exact placement of print is defined by vertical and horizontal coordinates. The colors can be set by using pre-defined macros. Furthermore, with `bmp_vram` function it is possible to obtain bitmap overlay of VRAM (Video RAM) address. `bmp_fill` allows to fill a section of bitmap memory with solid color. Lastly, to load whole BMP (Bitmap) file into memory so that it can be drawn on screen, the `bmp_file_t` function can be applied. The latter is used for cropmarks and zebra (highlights under and over-exposed zones in image) functions, being drawn pixel by pixel on the BMP overlay.

Listing 4.9: Screen interaction function

```
void bmp_printf(fontspec, x, y, const char * fmt, ...)
uint8_t * bmp_vram(void)
void bmp_fill(color, x, y, w, h);
struct bmp_file_t* bmp_load(const char * name);
```

Lua API

Lua is a lightweight and embeddable scripting language typically used to add functionality and customization to existing software. Magic Lantern developers created an API for interacting with the firmware using Lua scripts. The files with code are saved to destinated folder and can be read by loading module. The authors differentiate two kinds of scripts. Those, regarded as simple, do not define custom menus and do not leave any background tasks running. They also do not leave any event or property handlers registered when finished loading. Such functions are executed when called by user, so they do not autorun when starting the camera. There is an exception for event and property handlers' usage – they need to be closed before the main task is finished. On the other hand, more complex scripts define custom menus operate on background tasks and impact the handlers. It results in any global variables declared during the execution to persist, and they cannot be unloaded, which means that they run in background as long as the camera is powered on. All scripts can be either executed or toggled on/off to autorun from the menu.

Lua state, which works similarly to virtual machine, is individual for each script. However, sharing a state is possible by loading and running only one of the scripts and have it call the others using `dofile()` or `require()` functions. Moreover, pre-emptive multithreading is not allowed. If other tasks attempt to call functions in the script, such as through event handlers, they will be blocked until the current execution completes or yields, or a timeout is reached. This blocking behavior is implemented using semaphores in the scripting backend. It's important to note that separate scripts can run concurrently because they have their own Lua states. The scripting engine operates by loading and running scripts in a dedicated task specifically created for script loading. If a script blocks during the loading process, it will prevent subsequent scripts from loading.

Event and menu handlers are invoked from distinct Magic Lantern or Canon tasks, and scripts can initiate their own tasks. It also can voluntarily pause its execution and yield control to another task that is calling it by using the "task.yield" function. Additionally, execution is automatically yielded when the script returns from a function call. When creating scripts, it is important to take into consideration from where they will be called, and when to yield or use separate tasks in order not to block itself or other Magic Lantern and Canon tasks.

Similarly to the C built in functions, also in Lua scripting handlers can be set to monitor changes of the Canon properties, as well as send request to modify their values. However, they are stored in non-volatile memory, which set with invalid values can potentially damage the camera. Therefore, to mitigate the risks, some modules provide safe getters and setters for commonly used properties.

The scope of Lua scripting modules is wide. A few significant ones are presented below.

Event By assigning functions to the `event` table, scripts can handle them. These event handler functions typically accept a single integer parameter and are expected to return a boolean value indicating whether the backend should proceed with executing other event handlers for the same event. The scripts can be called in different ways, such as periodically from another task, called once per second, when a key is pressed as shown in listing 4.10, or after a picture is taken with the intervalometer.

Listing 4.10: Lua keypress event handler

```
event.keypress = function(key)
print("You pressed a key: "..key)
return false
end
```

Camera In this section, basic camera operation and properties are contained. One of the most important available possibilities for scripting here is taking a picture by function, either single shoot, series of pictures in burst mode specified by input variable or shooting in bulb mode. The camera can be also commanded to wait until it processes all current pictures, or to restart itself. Some of the functions are shown in listing 4.11. Furthermore, in this module a few classes are defined, such as shutter, aperture, ISO, exposure compensation and GUI (Graphical User Interface). Each of them has several attributes and methods covering mostly needed parameters and interactions. Some attracting the most attention might be setting/getting in various units the shutter speed, aperture, ISO, or getting information about current mode of camera. To sum up, this section allows to interact with and set the most crucial parameters regarding shooting photographs.

Listing 4.11: Lua camera and shooting control functions

```
shoot([should_af=false])
bulb(duration)
burst(num_pictures)
wait()
```

Constants Provides information about important constants used in the system. There can be found numbers corresponding to every key, which is naturally useful for many applications. Also there is a color pallete, fonts, constants regarding camera shooting mode, menu icon types, menu value units and dependencies for menu items

Display Introduces a variety of functions connected to displaying and drawing [listing 4.12]. From simply toggling the display on and off, taking screenshots or printing to drawing different shapes with set colors. Additionally, there is a class for BMP format files with a few attributes defining it. Such picture can be load from file, altered and printed on the screen.

Listing 4.12: Lua display functions

```
screenshot([filename[, mode]])
print(text, x, y[, font[, fg[, bg[, max_width]]]])
pixel(x, y[, color])
line(x1, y1, x2, y2, color)
rect(x, y, w, h, stroke[, fill])
circle(x, y, r, stroke[, fill])
load(filename)
draw(draw_func)
```

Key Constants regarding key values are introduced in different module. However, in this one a keypress can be sent, as well as camera can be commanded to wait for a specific button to be pushed as shown in listing 4.13. Additionally, information about the last key pressed is stored and can be obtained.

Listing 4.13: Lua key functions

```
press(key)
wait([key[, timeout]])
```

Lens This module is crucial for lens. Its functions [listing 4.14] cover moving the focus motor for a specified number of steps, working only in LiveView mode. The parameters passed to this `focus()` function are also step size, boolean `wait` which ensures that previous focus command movements have finished. The last one is delay, which in case of setting `wait` true, starts counting when each focus command is executes, and otherwise when the command is started. Another function that this module provide is controlling autofocus in a manner similar to half-shutter pressing, which return boolean value based on the success of the operation. Also, valuable information can be accessed like focal or hyperfocal length of the lens, current focus distance or raw relative focus motor position. Moreover, distance to the near and far depth of field is provided. Lastly, for autofocus task the current mode can be returned or if this function is enabled/currently in use.

Listing 4.14: Lua lens functions

```
focus(num_steps[, step_size=2[, wait=true[, delay]]])
autofocus()
```

Live View Reaches out API for entering and controlling Live View mode and its overlays. Allows operations like waiting, zooming and getting access to many attributes of LV through `lvinfo` class.

Menu Includes functions for interacting with the Magic Lantern menu, which are listed in 4.15. Allows to add new menu items, as well as access to many advanced settings menus in submenus. The `get()` function takes parameters such as name of the parent menu and name of the menu entry. It returns either a string type with the current menu text, or 0 if the entry is not found. `Set()` works in similar way, but the outcome is changing the value of entry instead of getting that information. Furthermore, with `select()` an item from the menu can be called. The return value provides information whether the call was successful. To create a new menu item, a table with the instance content needs to be provided for `new()` function. An exemplary use of this task is presented in listing 4.16.

Listing 4.15: Lua menu functions

```
get(menu, entry[, ret_type])
set(menu, entry, value)
open()
close()
select([menu[, entry]])
new(definition)
```

Listing 4.16: Lua creating a new menu

```
mymenu = menu.new
{
    parent  = "Movie",
```

```
    name    = "Lua Test Script",
    help    = "Some help for this script.",
    submenu ={{
            name    = "Run",
            help    = "Run some action.",
            update  = "",
        },{
            name    = "Parameter Example",
            help    = "Help for Parameter Example",
            min     = 0,
            max     = 100,
            unit    = UNIT.DEC,
            warning = function(this) if this.value == 5 then return "this
                ↪ value is not supported" end end,
        },{
            name    = "Choices Example",
            choices = { "choice1", "choice2", "choice3" },
        }
    },
    update = function(this) return this.submenu["Choices Example"].value
        ↪ end,
}

mymenu.submenu["Run"].select = function()
    print("Parameter Example= "..mymenu.submenu["Parameter Example"].value)
    print("Choices Example= "..mymenu.submenu["Choices Example"].value)
end
```

Property Grants access to all the Canon properties, which were introduced in C/C++ API. Also, the property class is contained in this module. Its method can alter values of properties. However, this operation is not safe due to risk of damaging the camera in case of setting invalid values. Another feature, which is much safer and still extremely useful is property handler method. It gets called whenever the property value changes and returns the new number.

Besides the described modules, there are also several more available, which might also prove useful when scripting. The other modules cover subjects like battery, console, dryOS, global functions, intervalometer, movie recording, config, keys helper, logger and tasks described in earlier part this section.

4.2.5. QEMU emulator

QEMU [51] is an open-source emulator and virtualizer that offers a versatile solution for various applications. It provides full system emulation, which means it can simulate the entire hardware environment of a computer system, including the processor, memory, storage devices, and peripherals. It can also perform binary translation, allowing software compiled for one architecture to run on another. This feature makes QEMU versatile and useful for a variety of tasks, such as running legacy software on modern systems or testing software on different platforms.

QEMU supports multiple hypervisiors such as KVM (Kernel-based Virtual Machine) or TCG (Tiny Code Generator), often referred to as accelerators, that enhance performance. Both can run on a Linux host OS. In case of host architectures, both ARM and x86 can be used.

In this work, the emulator was installed on a Linux kernel using WSL (Windows Subsystem for Linux). To make sure that the QEMU is integrated smoothly, a few tools should be priorly acquired. Firstly, a toolchain for ARM architecture is needed, such as `gcc-arm-none-eabi` to enable files cross compilation for the Canon camera emulation. A MinGW (Minimalist GNU for

Windows) provides a collection of open-source software tools, including set of headers, libraries, compilers and related utilities. It allows building and porting software from the Unix/Linux environment for Windows. Also, a modular system for processing documentation into useful formats – `python3-docutils` should be obtained. Lastly, in some versions `python2` is needed to handle part of scripts during installation.

Additional files to integrate the QEMU environment with Canon camera's emulation needs to be obtained from Magic Lantern repository [30]. The developers recommend to run the process from the QEMU directory, because it should contain the latest developments. The installation is done in a separate location from the main Magic Lantern folder. This setup enables easier emulation of any branch without the need of merging QEMU into currently working one. Therefore, the first steps are to clone the repository and run `install.sh` from the QEMU directory as shown in listing 4.17.

Listing 4.17: Cloning repository and running QEMU installation

```
hg clone https://foss.heptapod.net/magic-lantern/magic-lantern
cd magic-lantern
/path/to/magic-lantern$  hg update qemu -C
/path/to/magic-lantern$  cd contrib/qemu
/path/to/magic-lantern/contrib/qemu$  ./install.sh
```

4.2.6. Firmware and Magic Lantern installation

After installation of the QEMU, it should be configured with booting files appropriate for the chosen camera model. In this work, the main focus is concentrated around the Canon EOS 550D. To arrange the environment for this device, its ROM files need to be supplied. To acquire them, a dumping module needs to be compiled and ran on the camera. Having the files provided, QEMU can be further built and configured (see listing 4.18).

Listing 4.18: Supplying ROM files and compiling QEMU

```
/path/to/qemu-eos$  cp /path/to/sdcard/ML/LOGS/ROM*.BIN 550D/
/path/to/qemu-eos$  cd qemu-2.5.0
/path/to/qemu-eos/qemu-2.5.0$  ../configure_eos.sh
/path/to/qemu-eos/qemu-2.5.0$  make
```

After achieving the so far presented steps, a basic Canon firmware can be run. To do this, a command from listing 4.19 needs to be executed from the `qemu-eos` path with value 0 for the boot argument. To gain access for the Magic Lantern extensions, it needs to be compiled and installed onto the virtual SD card using the second command from listing 4.19. If the installation was successful, the emulation can be launched either with basic functionality, or Magic Latern add-on by setting boot parameter as 1.

Listing 4.19: Running the firmware and Magic Lantern installation

```
/path/to/qemu-eos$  ./run_canon_fw.sh 550D,firmware="boot=0"
/path/to/qemu-eos$  make -C ../magic-lantern 550D_install_qemu
/path/to/qemu-eos$  ./run_canon_fw.sh 550D,firmware="boot=1"
```

Chapter 5

Proposals and implementation

5.1. Promising candidates

Analysing programming possibilities and the tools offered by the APIs resulted in gathering functional knowledge for further improvements. Based on that, a few ideas for implementations to extend the features are proposed.

The first method is intended to provide assistance for photographer during shooting. Consequently, it can be classified as supporting function for classic picture making. The idea for application is to improve currently existing cropmarks feature. It helps the person to adjust the composition by overlying the lens view with auxiliary images. It is meant to improve the initial software with aid of available tools described earlier. Furthermore, the task is suitable for conducting tests on cross-compilation.

The aim of subsequent approach is to solve problems known and described in the literature. Based on the carried out review, integration of a solution incorporated in a known problem into the digital camera native system is proposed. Performing such tasks allows to examine differences between environments and difficulties during conversion. Another important aspect which must be considered is strictly limited memory accessible in embedded devices. Therefore, application of an automatic white balance CNN is suggested to assess the possibilities of converting computer vision algorithms for use in compact devices offering less resources.

The last proposition is meant to perform a separate, practical task instead of being an augmentation or improvement to another feature. The aim is to employ the camera as a diagnostic device assessing damage of mobile phones protective film. To achieve this, the idea is to use an edge detection algorithm evaluating the cracks' progression. This approach, similarly to automatic WB DNN, also incorporates already known solutions. However, in this case the intention is to address a specific, short range of use application.

From the three proposed approaches, only the custom cropmarks feature fully relies on the already existing API. The other applications include external libraries such as OpenCV. Because of that, the objective is to firstly test their prototype versions. To achieve that, the aim is to rewrite the Python source codes based on implementations published in the literature into C++ language. After successful experiments, the programs could be further integrated into ARM platform. However as previously mentioned, assessment of the resources usage is crucial before deploying the new functions.

In the next sections, the stated proposals are analysed providing algorithms' mathematical background and source code of the methods.

5.2. Cropmarks

Cropmarks refer to visual guides or overlays displayed on the camera's LCD screen or viewfinder. They help photographers compose their shots and align their subjects within specific guidelines or aspect ratios. The cropmarks typically include a grid or a set of horizontal and vertical lines that divide the frame into equal sections. They can be useful for maintaining proper horizons, following specific composition rules, or dividing the image in equal segments. The overlays can also highlight how the currently seen image would look in certain aspect ratios, such as 4:3, 16:9 or 1:1. Moreover, other pictures can be used as these overlays with set percentage of transparency, which can be used to aid targeting a specific view or repeat perspectives.

Magic Lantern offers a few examples of cropmarks. Besides auxiliary grids, there is a passport image overlay helpful for accurate targeting of person's face for this specific purpose. They are stored in camera memory as `bmp` images, from where they can be loaded and displayed on the LCD screen. These actions are executed by functions shown in listing 5.1.

Listing 5.1: Functions for finding and loading cropmarks' files

```
static void find_cropmarks(){
    struct fio_file file;
    struct fio_dirent * dirent = FIO_FindFirstEx( "ML/CROPMKS/", &file );
    if( IS_ERROR(dirent) ){
        NotifyBox(2000, "ML/CROPMKS dir missing\n" "Please copy all ML
            ↪ files!" );
        return;
    }
    int k = 0;
    do {
        if (file.mode & ATTR_DIRECTORY) continue;
        if (is_valid_cropmark_filename(file.name)){
            if (k >= MAX_CROPMARKS){
                NotifyBox(2000, "TOO MANY CROPMARKS (max=%d)",
                    ↪ MAX_CROPMARKS);
                break;
            }
            snprintf(cropmark_names[k], MAX_CROP_NAME_LEN, "%s",file.name);
            k++;
        }
    } while( FIO_FindNextEx( dirent, &file ) == 0);
    FIO_FindClose(dirent);
    num_cropmarks = k;
    sort_cropmarks();
    cropmarks_initialized = 1;
}

static void reload_cropmark(){
    int i = crop_index;
    static int old_i = -1;
    if (i == old_i) return;
    old_i = i;
    if (cropmarks){
        void* old_crop = cropmarks;
        cropmarks = 0;
        free(old_crop);
    }
    cropmark_clear_cache();
    if (!num_cropmarks) return;
    i = COERCE(i, 0, num_cropmarks-1);
    char bmpname[100];
    snprintf(bmpname, sizeof(bmpname), "ML/CROPMKS/%s", cropmark_names[i]);
```

```
    cropmarks = bmp_load(bmpname,1);
    if (!cropmarks) bmp_printf(FONT_LARGE, 0, 50, "LOAD ERROR %d:%s   ", i,
        ↪ bmpname);
}
```

The `find_cropmarks()` function at first checks if they folder containing cropmark files exists. Then, it searches for valid images with proper names. Subsequently, they are sorted alphabetically and loaded into the camera memory by `reload_cropmarks()`. Besides using these methods, cropmarks can be drawn using functions explained in 4.2 screen subsection. Because of these possibilities, customised cropmarks can be provided. The loading function searches for all files in set folder, so newly added images can be automatically loaded into the module. However, besides being in `.bmp` format, for proper display they should be using the specified 8-bit palette. Modifying and cross compiling the `cropmarks.c` source file allows to increase the maximum number of cropmarks, which is initially set at 9 (listing 5.2).

Listing 5.2: Macros for cropmarks' constraints

```
#define MAX_CROP_NAME_LEN 15
#define MAX_CROPMARKS 11
```

Maximum length of its name can also be changed. This feature can be turned on in the overlay menu. In the entry's options user can enable a mode in which the cropmarks should be displayed. Available possibilities are photo mode and play mode, as shown in figure 5.1. This add-on is also used by another function – zebra, which draws lines in under and overexposed areas.

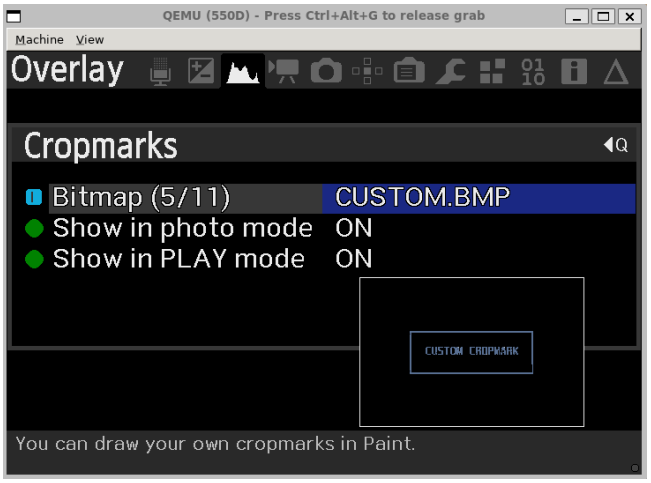

Figure 5.1: A view of cropmarks menu in camera

5.3. White Balance

In the [3], authors propose a single deep learning framework to achieve WB correction and manipulation. Such method allows post-capture picture editing to obtain a corrected result. Implementing such module into the camera firmware could possibly allow the photos to be automatically corrected before being stored in the memory.

For such model learning, lots of pictures are needed with both correct and invalid WB setting. The first problem which needs to be solved is getting pure images, taken with no prior setting of this parameter. Having these photos would significantly reduce complexity of the assignment. The ideal procedure would be to recover unprocessed image and then change the initial $WB(in)$ setting to $WB(t)$, which is the desired target outcome. Then, the image would be rendered again to the sRGB color space using a software-based Image Signal Processor. The ideal procedure

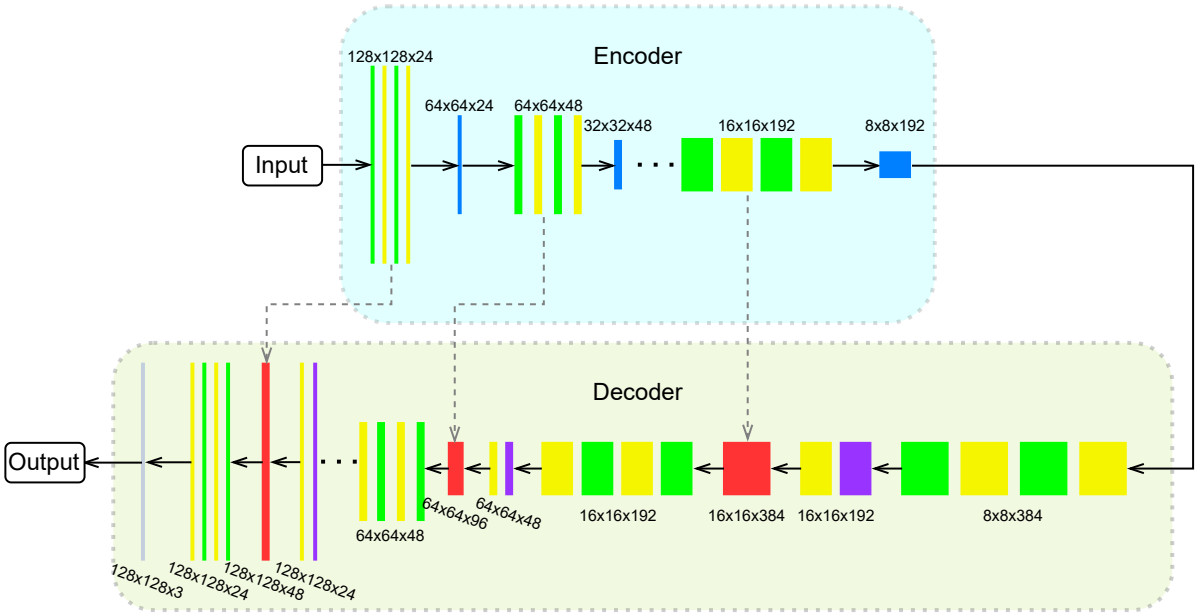

Figure 5.2: Architecture of the automatic WB DNN, based on [3]

would be characterised by equation $I_{WB(t)} = G(F(I_{WB(in)}))$, where F is an unknown function reconstructing the camera-rendered sRGB image to raw-RGB with $WB(in)$ and G is a unknown process that edits the WB and re-renders the outcome.

However, the main objective is not to reconstruct or recreate the original raw-RGB values directly. Instead, the aim is to generate the final sRGB image using the desired white balance setting, $WB(t)$. To achieve this, operation of $G(F(*))$ can be described as an encoder/decoder system, where the input image is converted into latent representation by encoder f, while decoders g^1, g^2, or g^3 produce final pictures with various WB settings depending on the chosen mood of the image. Besides automatic lighting correction of scene, moods corresponding to the indoor and outdoor lighting conditions can be chosen. Therefore, three different decoders are implemented to achieve this functionality.

The architecture of this DNN is based on U-net [46] and is shown in the figure 5.2, with multi-scale skip connectors between encoders and decoders being added. Both encoders and decoders units consist of four levels. Each unit includes bottleneck and transposed convolutional layers. The latter ones consist of 24 channels in first level, doubling this number with each step resulting in 192 channels in fourth level. Outputs of 3x3 convolutional layers with both stride and padding parameters set to value 1 are represented in green colour. Yellow corresponds to outcomes of ReLu layers. Then, in blue outputs of 2x2 max-pooling layers with stride value of 2 has been shown. All those elements are building the encoder unit, whose output is proceeded to the decoders. On the side of images reconstruction, purple colour represents output of 2x2 transposed convolutional layers with stride of 2. In red there are products of depth concatenation, and lastly grey colour indicates result of 1x1 convolution with both stride and padding of 1. Dashed arrows specify the skip connectors.

For training and validation of the model, a Rendered WB [4] dataset was applied. It contains approximately 65,000 sRGB images with different WB settings, generated using various devices, mostly seven different DSLR cameras, and small number of pictures taken by mobile phones. For each image, there is a corresponding ground truth image rendered, considered as the accurate WB setting. During each epoch, four 128x128 patches were randomly selected from every training image with their respective ground truth representations for different encoders. Additionally, to prevent overfitting, geometric augmentation such as rotation and flipping were

used. The model was trained considering L1-norm loss function, which is also knows as least absolute deviations error. Its target is to minimize the sum of the absolute differences between the true and estimated values of pixels. This DNN has the capability to handle input images in their original dimensions, provided that they are multiplies of 16. This requirement is caused by the architecture constraints, and more precisely the use of 4-level encoder/decoder with 2x2 max-pooling and transposed convolutional layers. However, to maintain a consistent run time regardless of size, all input images are resized to a maximum dimension of 656 pixels.

Embedding this neural network framework into the camera could be done as a toggled module taking one of the three settings as an input parameter. However, it is important to note that neural networks consume lots of processing power, so further optimalisations might be crucial for usage. One of the approaches is to change the PyTorch/Python environment to LibTorch/C++. Such conversion to lower level language might reduce the memory needed for performing described functions. Furthermore, from this point, a cross-compilation of the code with auxiliary libraries such as Libtorch and OpenCV into ARM platform can be achieved with setting up proper toolchains or employing tools like Tengine [37]. This DNN model remains sophisticated with its multi-layer architecture. Because of that the implementation might not be optimal for many cases. Nevertheless, it is an example providing good functionality in terms of WB, which can be further modified or reduced to produce satisfactory results.

5.3.1. Converting PyTorch model to Torch Script

To use a PyTorch model pre-trained in Python in the C++ API, it has to be converted into Torch Script. There are two approaches for such conversion [43]. The first one, known as tracing, involves evaluating the model once with example inputs and recording the input flow to capture its structure. This method is suitable for models that have minimal control flow. The second method involves adding explicit annotations to the model, which inform the Torch Script compiler that it can directly parse and compile the model code, considering the limitations of the Torch Script language.

Converting via tracing In order to convert a PyTorch model to Torch Script using the tracing method, one needs to provide the `torch.jit.trace` function with an instance of the model and an example input. This process generates a `torch.jit.ScriptModule` object that contains the trace of the model evaluation into the forward method of the module. A part of code performing this process is shown in listing 5.3. The outcome can be evaluated in the same way as a regular PyTorch module. The `deepWBnet()` method creates an instance of model. A random tensor of exemplary input dimensions and sizes is generated and used to trace the neural network and obtain script module.

Listing 5.3: Converting PyTorch model to Torch Script using tracing method

```
import torch
import torch.nn as nn
model = deep_wb_single_task.deepWBnet()
example = torch.rand(1, 3, 640, 640)
traced_script_module = torch.jit.trace(model, example)
```

Converting via annotation In certain situations, such as when a model involves specific types of control flow, the model can be directly written in Torch Script and be annotated accordingly. As shown in listing 5.4, the forward method changes the control flow based on the input data. Because of that, it is not suitable for the first method. To convert the model properly, it must be compiled using `torch.jit.script` in a manner when the forward function is exposed.

Listing 5.4: Converting PyTorch model to Torch Script using annotation method

```
class MyModule(torch.nn.Module):
```

```
    def __init__(self, N, M):
        super(MyModule, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(N, M))
    def forward(self, input):
        if input.sum() > 0:
          output = self.weight.mv(input)
        else:
          output = self.weight + input
        return output
my_module = MyModule(10,20)
script_module = torch.jit.script(my_module)
```

5.3.2. Serializing the Script Module to a file

As the next step, the module needs to be serialized into a file. This operation is rather simple, as it consists of using one function, which saves the model into `.zip` or `.pt` format [listing 5.5]. This allows to load the file into C++ environment and execute the module without any dependency on Python.

Listing 5.5: Function for saving traced Torch Script to a file

```
traced_script_module.save("traced_AWB_module.zip")
```

5.3.3. Loading and executing the Script Module in C++

In order to load a serialized PyTorch model in C++, the application needs to rely on the PyTorch C++ API, which is commonly referred to as LibTorch. It consists of a set of shared libraries, header files, and CMake build configuration files. Listing 5.7 shows a program written to load and convert an image into a vector used as an input to the auto WB network. Then, the output is transformed back into an image with corrected WB settings and saved into a new file. The code starts with importing necessary libraries for OpenCV and Torch [listing 5.6]. In the main part, after loading the image from given path, it needs to be converted into float type and have its values scaled in the range from 0 to 1. It is necessary due to the model having float type weights. Then, a tensor is created using newly obtained data, which needs to have its dimensions rearranged to be compatible with the model. With this data, finally an input vector can be created and passed to the loaded module. After successful evaluation, output vector can be converted back into an image and saved.

Listing 5.6: OpenCV and Torch libraries needed for autoWB in C++

```
#include <iostream>
#include <memory>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/imgproc.hpp>
#include <torch/script.h>
#include <torch/torch.h>
```

Listing 5.7: Loading and executing Torch Script autoWB module

```
auto ToInput(at::Tensor tensor_image){
    return std::vector<torch::jit::IValue>{tensor_image};
}

int main(int argc, const char* argv[])
{
    std::string imagePath = argv[1];
                cv::Mat img = cv::imread(imagePath);
    img.convertTo(img, CV_32FC3, 1.0f / 255.0f);
```

```
    auto tensor = torch::from_blob(img.data, {1, img.size().height, img.
        ↪ size().width, 3});
    tensor = tensor.permute({0, 3, 1, 2});
    auto input_to_net = ToInput(tensor);
    std::string AWB_model_path = "traced_AWB_module.zip";
    torch::jit::script::Module AWB = torch::jit::load(AWB_model_path);
    at::Tensor out_tensor = AWB.forward(input_to_net).toTensor();
    out_tensor = out_tensor.squeeze().detach().permute({1, 2, 0});
    out_tensor = out_tensor.mul(255).clamp(0, 255).to(torch::kU8);
    out_tensor = out_tensor.to(torch::kCPU);
    cv::Mat resultImg(img.size().height, img.size().width, CV_8UC3);
    std::memcpy((void *) resultImg.data, out_tensor.data_ptr(), sizeof(
        ↪ torch::kU8) * out_tensor.numel());
    cv::imwrite("AWB_output_image.jpg", resultImg);
    return 0;
}
```

The presented function can be cross compiled for ARM architecture and added into the customised firmware. One of the approaches could be automatic correction of images saved onto the camera's memory. Such function could be toggled from its own entry added to the menu. Additionally, with further integration, the auto WB could operate on data from camera sensor adjusting this parameter in real time. However, it is important to highlight that the second method would be more demanding computationally. Another possibility is calling this function with a Lua script.

5.4. Cracks detection on phone screen

Edge detection techniques can be used in order to assess wear or quality of the film used to protect phone screen. Such functions could be implemented as features to existing firmware. The state of damage could be evaluated by taking a picture of examined phone and applying algorithms highlighting and evaluating cracks. Based on defined conditions, a feedback would be provided whether the film should be changed. In this section, implementation of Canny and HED (Holistically-Nested Edge Detection) methods will be presented and compared.

5.4.1. Canny edge detector

This technique was introduced by John F. Canny [10] in 1986. Despite its age, it is still widely used in computer vision and image analysis applications. It is a multi-step algorithm, which starts with convolving the image with a Gaussian filter in order to reduce noise. The next step is finding intensity of gradient image. To do that, the smoothened image is filtered with Sobel kernel in both horizontal and vertical axes. It results in obtaining the first derivative, and from those two auxilary arrays, edge gradient and its direction can be found (see eq. 5.1 and eq. 5.2). The gradient direction is always perpendicular to edges. Once the gradient magnitude and direction are obtained, a thorough scan of the image is conducted to eliminate any undesirable pixels that may not contribute to forming an appropriate edge. During this process, each pixel is examined to determine if it represents a local maximum within its surrounding neighbourhood, aligned with the direction of the gradient. If the checked pixel is not maximum, it is suppressed – its value is set to 0. It results in creating thin and distinct edges. Then, hysteresis thresholding determines which true edges from the non-edges. To accomplish this, two threshold values - for minimum value and maximum value are used. Edges having an intensity gradient exceeding the maximum value are recognized as genuine edges, while those falling below minimum threshold are definitively classified as non-edges and are disregarded. Furthermore, edged that lie between these two thresholds are classified depending on their connectivity. If they are connected to the

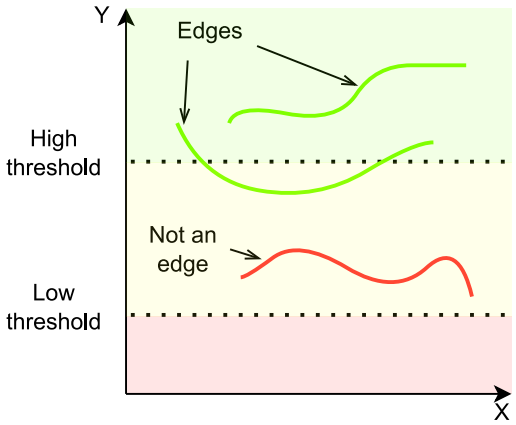

Figure 5.3: Concept of hysteresis thresholding

true edges, then they are considered as parts of them. In case of the opposite, they are also discarded (see figure 5.3). Also, during this stage any small pixel noises are removed because it is assumed that edges in the image are typically long lines rather than short, scattered dots [39].

$$G = \sqrt{G_x^2 + G_y^2} \tag{5.1}$$

$$\theta = tan^{-1}\left(\frac{G_y}{G_x}\right) \tag{5.2}$$

This approach is still used due to its accuracy and robustness in noisy environments. Moreover, selecting appropriate thresholds allows to control the trade-offs between edge detection accuracy and noise suppression. Also, the OpenCV library has included this function making it easily available for usage. However, it also has some drawbacks such as setting the lower and upper values to the hysteresis thresholding. It is a manual process requiring experimentation and visual validation. Also, these values might give satisfactory results on one image, and might not necessarily produce good outcome for another one.

5.4.2. Holistically-nested Edge Detection

Holistically-nested Edge Detection [55] method aims to detect and extract edges from an image in a holistic manner. It is an end-to-end deep neural network approach. It takes an RGB image as an input and returns its edge map. HED, in contrast to conventional techniques for edge detection, that concentrate on local edge details, adopts a global strategy by leveraging multi-scale and multi-level characteristics within a CNN framework. HED uses a "side-output" approach, where intermediate feature maps from different layers of the network are extracted and combined to produce holistic edge map. This integration of multi-scale features helps to capture edges at different scales and complexities, leading to more comprehensive and precise detection.

Regarding the architecture, the selection of hierarchy for the framework requires careful consideration. It is important to have a deep architecture that can efficiently produce features at multiple levels for better perception. Additionally, using multiple stages with different strides is necessary to capture the various scales of edge maps. Therefore, the authors based architecture of their network on VGGNet [47], which achieved state-of-the-art performance in the ImageNet challenge. However, to make the adapted net suitable for purposes of their work, they made a few changes in the architecture. Firstly, the side output layers were linked to the last convolutional layer in each stage. The receptive field size of these convolutional layers matches that of the corresponding side-output layer. The second change was to remove the last stage

of VGGNet containing 5th pooling layer and all fully connected layers. The side outputs are meant to be aggregated to obtain final result and having a layer with stride 32 would produce an output that is too small, resulting in a distorted interpolated prediction map. Another reason for change was to reduce computational costs, but also save memory and time during training and testing processes. The final HED network architecture consists of five stages nested within the VGGNet. They have strides of 1, 2, 4, 8, and 16 respectively, and different receptive field sizes. The architecture schematic of this neural network is shown in figure 5.4. Green colour represents convolutional layers, blue – pooling, orange – the side outputs, grey – the resized side outputs, and black colour is for weighted-fusion layer. The latter one evaluates the final output of network based on results from each stage having different sizes. It is important to highlight, that the ground truth is duplicated, and the error is back propagated at each side-output layer. Additionally, during aggregating the side-output predictions, the weighted-fusion layer learns the fusion weights.

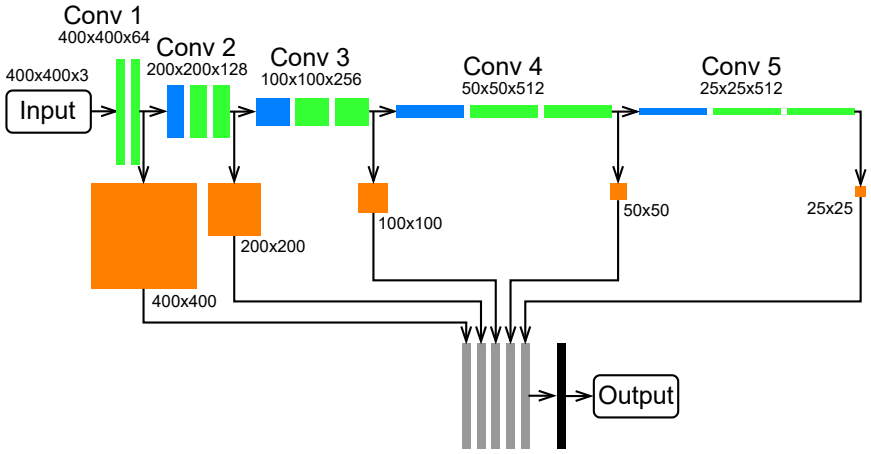

Figure 5.4: Holistically-nested Edge Detection DNN architecture, based on [55]

To conclude, this network achieves state-of-the-art performance on natural images with excellent speed compared to other CNN approaches. The advantages for implementing it in screen cracks detection is the automatic prediction with no prior parameters setting like in case of Canny filter. Moreover, it is also included in the OpenCV library since version 3.4.3.

5.4.3. Edge detection implementation

Canny In order to test the described methods, the code was implemented in C++ language based on OpenCV application [38]. In listing 5.8 variables and functions used for Canny edge detection are descibed. Firstly, a few objects for storing images were constructed. One for the original loaded image, then another one for its converted grayscale version. The last two are meant to store the detected edges and the final output from this algorithm. The variables describe lower threshold used in hysteresis thresholding, a ratio from which the upper threshold is determined by multiplying it with the value of lower one. The kernel size determines size of the mask performing detection. All those variables can be adjusted to obtain satisfactory results. In the case of screen cracks detects, it was decided to set both ratio and kernel sizes to 3. The function was run with different low threshold values to compare results, because there is no ideal value for every conditions. During the Canny operation, the input grayscale image is firstly blurred using a mask of size 3x3. Having the blurred picture, the main body of edge detection function is executed providing previously defined parameters. Then, a template for 4-element vector is created and the outcome from the algorithm is copied into final image container.

Listing 5.8: Canny edge detection algorithm implementation

```
#include <opencv2/opencv.hpp>

cv::Mat canny_source, canny_source_gray;
cv::Mat canny_result_img, canny_detected_edges;
int canny_low_thresh = 15;
const int canny_ratio = 3;
const int canny_kernel_size = 3;

static void CannyThreshold(int, void*){
    blur(canny_source_gray, canny_detected_edges, Size(3,3));
    Canny(canny_detected_edges, canny_detected_edges, canny_low_thresh,
        ↪ canny_low_thresh*canny_ratio, canny_kernel_size);
    canny_result_img = Scalar::all(0);
    canny_source.copyTo(canny_result_img, canny_detected_edges);
}

int main( int argc, char* argv[] ){
    canny_source = cv::imread("../input_images/phone_screen_crack.png");
    cv::cvtColor(canny_source, canny_source_gray, COLOR_BGR2GRAY);
    CannyThreshold(0, 0);
}
```

HED The HED C++ code used in this work was also based on the OpenCV code and the original implementation from the article [54], from where the pretrained model is loaded. HED was implemented using the publicly available Caffe Library. Therefore, it is encoded into two files, a `deploy.prototxt` Caffe JSON (JavaScript Object Notation) text file with the model definition and `hed_pretrained_bsds.caffemodel` containing the neural network weights. After loading necessary libraries, a crop layer is added to prevent shifting the result image to the right and bottom corner. Then, the forward function is defined to properly handle the input image to the network. After having loaded and set the model, the picture to be passed as an input is converted to a vector, which is standardised by subtracting from each channel its mean value and scaling by a defined factor. Similarly, the HED neural network output is reversed back to an image which is then saved to a file. The whole process is presented in listing 5.9.

Listing 5.9: HED algorithm implementation

```
#include <opencv2/opencv.hpp>
#include <opencv2/dnn.hpp>
#include <opencv2/dnn/layer.details.hpp>
#include <opencv2/dnn/shape_utils.hpp>
#include <iostream>
using namespace cv;
using namespace std;
using namespace cv::dnn;
class myCropLayer : public Layer{
public:
    myCropLayer(const LayerParams &params) : Layer(params){}
    static cv::Ptr<Layer> create(LayerParams& params){
        return cv::Ptr<Layer>(new myCropLayer(params));
    }
    virtual bool getMemoryShapes(const std::vector<std::vector<int>> &
        ↪ inputs, const int requiredOutputs, std::vector<std::vector<int>>
        ↪ &outputs, std::vector<std::vector<int>> &internals)
        ↪ const CV_OVERRIDE{
        CV_UNUSED(requiredOutputs); CV_UNUSED(internals);
        std::vector<int> outShape(4);
        outShape[0] = inputs[0][0];
```

```
        outShape[1] = inputs[0][1];
        outShape[2] = inputs[1][2];
        outShape[3] = inputs[1][3];
        outputs.assign(1, outShape);
        return false;
    }
    virtual void forward(std::vector<Mat*> &input,
        ↪ std::vector<Mat> &output, std::vector<Mat> &internals)
        ↪ CV_OVERRIDE{
        cv::Mat * inp = input[0];
        cv::Mat  out = output[0];
        int ystart = (inp->size[2] - out.size[2]) / 2;
        int xstart = (inp->size[3] - out.size[3]) / 2;
        int yend = ystart + out.size[2];
        int xend = xstart + out.size[3];
        const int batchSize = inp->size[0];
        const int numChannels = inp->size[1];
        const int height = out.size[2];
        const int width = out.size[3];
        int sz[] = {(int)batchSize, numChannels, height, width};
        out.create(4, sz, CV_32F);
        for(int i = 0; i < batchSize; i++){
            for(int j = 0; j < numChannels; j++){
                cv::Mat plane(inp->size[2], inp->size[3], CV_32F,
                    ↪ inp->ptr<float>(i, j));
                cv::Mat crop = plane(cv::Range(ystart, yend),
                    ↪ cv::Range(xstart, xend));
                cv::Mat targ(height, width, CV_32F, out.ptr<float>(i,j));
                crop.copyTo(targ);
            }
        }
    }
};

int main(int argc, char* argv[]){
    CV_DNN_REGISTER_LAYER_CLASS(Crop, myCropLayer);
    Net net = readNet("deploy.prototxt", "hed_pretrained_bsds.caffemodel");
    cv::Mat img = cv::imread("../input_images/phone_screen_crack.png");
    cv::Mat HED_input;
    resize(img, HED_input, img.size());
    cv::Mat blob = blobFromImage(HED_input, 1.0, img.size(), cv::Scalar(cv
        ↪ ::mean(img)[0],cv::mean(img)[1],cv::mean(img)[2]), false, false);
    net.setInput(blob);
    cv::Mat HED_output = net.forward();
    std::vector<cv::Mat> vectorOfImagesFromBlob;
    imagesFromBlob(HED_output, vectorOfImagesFromBlob);
    cv::Mat tmpMat = vectorOfImagesFromBlob[0] * 255;
    cv::Mat tmpMatUchar;
    tmpMat.convertTo(tmpMatUchar, CV_8U);
    cv::resize(tmpMatUchar, HED_output, img.size());
}
```

Image segmentation The cases with pictures showing a mobile phone with different backgrounds or only the screen of phone is desired to be extracted are harder to analyse. In those situations, HED can be further improved with image segmentation (see listing 5.10). It is typically done by blurring the binary HED output image and applying thresholding again. The original image and result of this operation is presented in figures 5.5a and 5.5b. Then, the connected components are labelled. During this phase, the number of labels, their statistical description

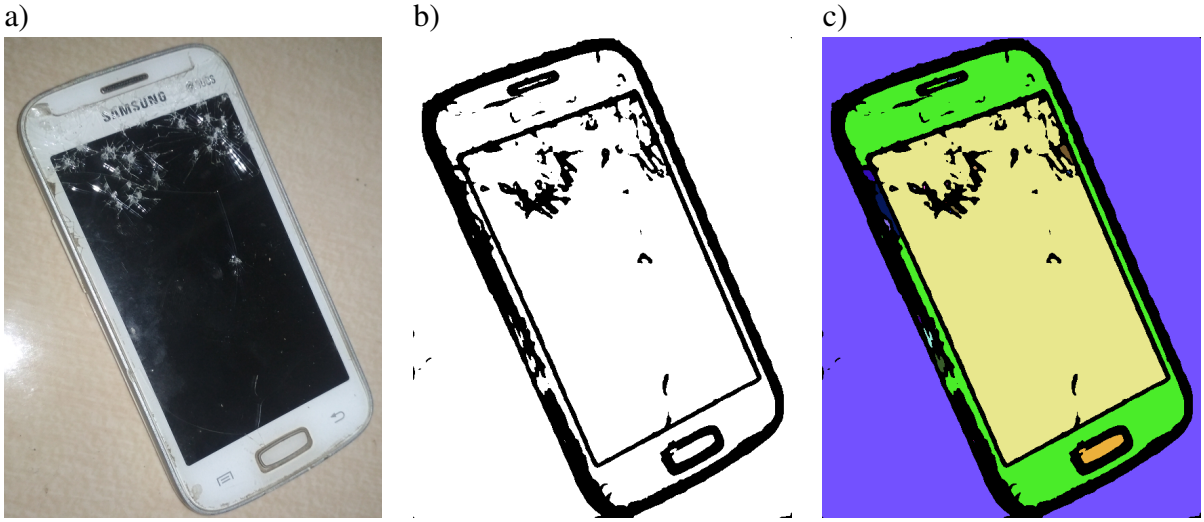

Figure 5.5: Mobile phone screen crack image segmentation in: a) original image b) blurred and thresholded, c) colour segmented

and position of centroids can be obtained. The `connectedComponentsWithStats()` function takes as the last argument connectivity parameter. When set to 4, it connects the pixels horizontally and vertically. However, putting the value as 8 also adds diagonal linking. Based on the obtained information from this stage, further filtering can be implemented to discard unwanted parts, for instance very small segments. Next, these segments are represented in different colours (see figure 5.5c). From this point, is it easier to analyse different parts of image separately, which might be a vast improvement for the phone screen crack detection feature.

Listing 5.10: Image segmentation implementation

```
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;
using namespace std;

int main(){
    cv::Mat blurred_HED_img;
    cv::GaussianBlur(HED_output, blurred_HED_img, cv::Size(3,3), 0);
    cv::Mat thresholded_HED_img;
    cv::threshold(blurred_HED_img, thresholded_HED_img, 0, 255,
       ↪ cv::THRESH_BINARY_INV + cv::THRESH_OTSU);
    cv::Mat labels, stats, centroids;
    int n_labels = connectedComponentsWithStats(thresholded_HED_img,
       ↪ labels, stats, centroids, 4);
    std::vector<cv::Vec3b> colors(n_labels);
    for(int i = 0; i < n_labels; i++)
        colors[i] = cv::Vec3b(rand()&255,rand()&255,rand()&255);
    cv::Mat colour_segmented_HED_img =
       ↪ cv::Mat::zeros(thresholded_HED_img.size(), CV_8UC3);
    for(int i = 0; i < thresholded_HED_img.cols; i++){
        for(int j = 0; j < thresholded_HED_img.rows; j++){
            if(labels.at<int>(cv::Point(i,j)) != 0){
                colour_segmented_HED_img.at<cv::Vec3b>(cv::Point(i,j)) =
                   ↪ colors[(int)labels.at<int>(cv::Point(i,j))];
            }
        }
    }
}
```

Chapter 6

Results

6.1. Cropmarks

The custom cropmarks `bmp` files were prepared and stored in the folder where they are loaded from. Then, by modifying the `cropmarks.c` source file, the number of available overlays which can be loaded onto the card was increased to 11. After cross compilation, the number of slots in cropmarks submenu increased, so the new picture could be displayed on the screen. A personalised image was used in the Live View mode, as shown in figure 6.1. Therefore, as the final result, the customisation was successfully achieved.

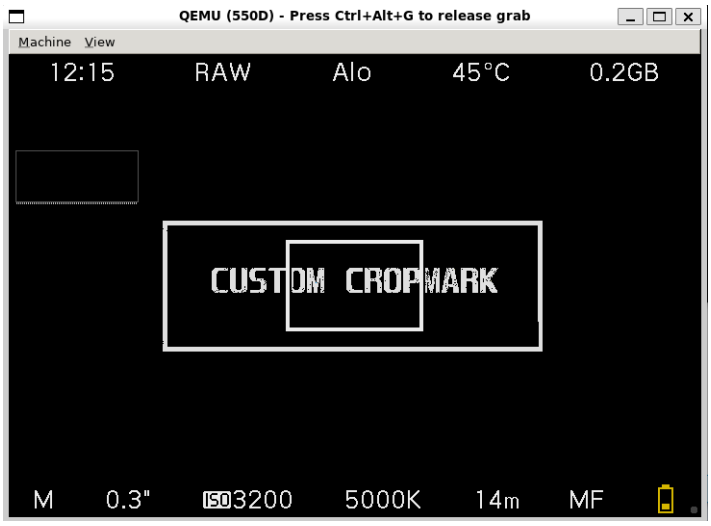

Figure 6.1: A custom cropmark loaded into camera's Live View

6.2. Automatic WB Deep Neural Network

The C++ API application of auto WB neural network model outputs transformed images. However, they are not the same as the ones obtained by Python even though the C++ implementation has not yet been converted into the ARM environment, and is in its prototype version. Additionally, the same outcome is obtained for models traced for CPU and GPU/CUDA (Compute Unified Device Architecture) types of weights. The comparison performed with pictures provided in [3] is shown in the figure 6.2. In the first column original images with incorrent WB are shown. In the second one, results from C++ implementation are represented. The last column contains outcomes of Python implementation of the program. As the results are not completely

Figure 6.2: AutoWB DNN outputs comparison

distinct and the C++ program does not act with images in a random way, the differences might be caused by numerical constraints of the environments.

6.3. Edge detection

The results of edge detection algorithms are firstly compared separately, but with different input parameters. For Canny function, the three main variables are low threshold, high threshold and kernel size. The latter one was set to 3. The high threshold is calculated by multiplying the lower one, whose influence will be examined, with value of 3. Therefore, in the first comparison presented in figure 6.3, the differences between outputs of Canny edge detection algorithm with different low threshold values are shown. The chosen image has one significant crack and a few smaller ones. It was selected purposely to inspect detection of various, not obvious edges. Also,

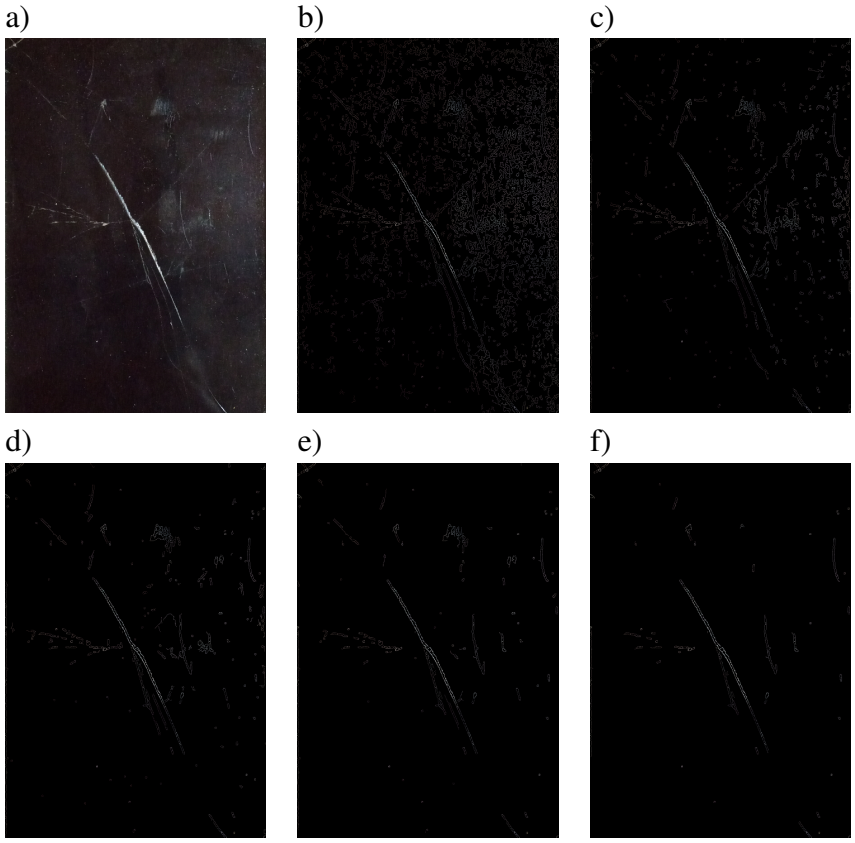

Figure 6.3: Comparison of Canny edge detection for different low threshold values: a) input image, b) threshold = 10, c) threshold = 20, d) threshold = 30, e) threshold = 40, f) threshold = 50

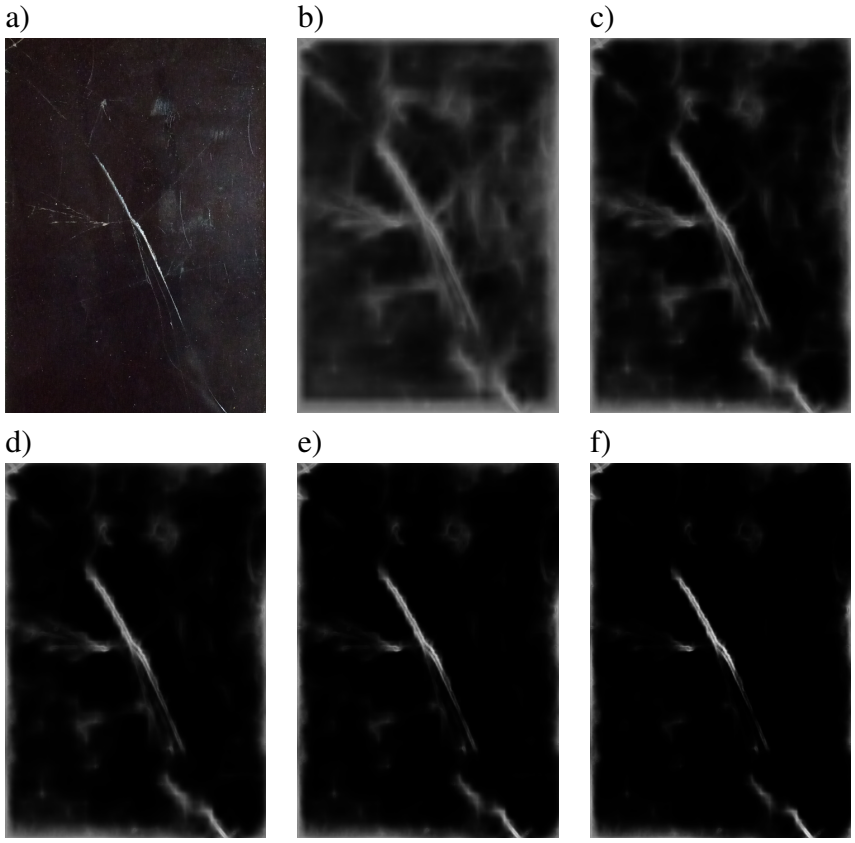

Figure 6.4: Comparison of HED for different scaling factor values: a) input image, b) f = 0.1, c) f = 0.3, d) f = 0.5, e) f = 0.7, f) f = 1

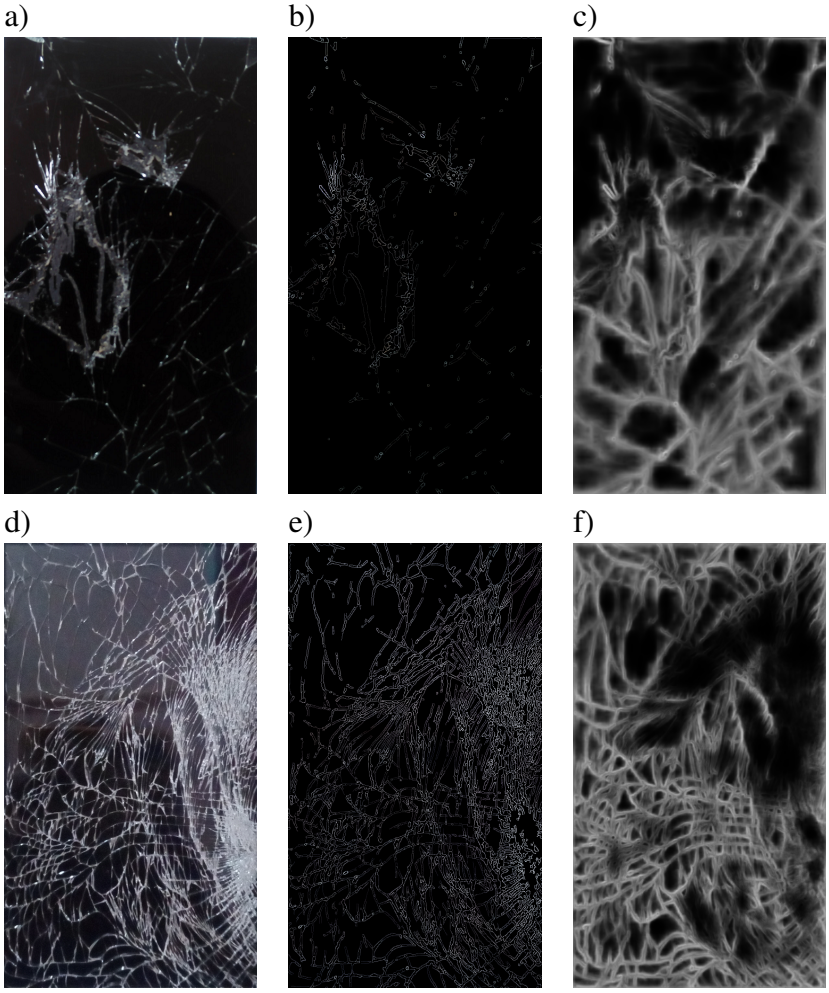

Figure 6.5: Comparison of HED and Canny edge detectors for heavily cracked screens: a) & d) input images, b) & e) Canny - low threshold = 40, c) & f) HED - scaling factor = 0.5

it is important to note that lighting conditions are not homogenous for whole image. In figure 6.3a, a reflection of light can be seen in the right part of image. For threshold value of 10, some noise is generated in this part. The best outcome seems to be in figures 6.3c and 6.3d. These two achieve a trade-off between noise intesity and information loss. The last two results discard some of the important edges.

Regarding HED, due to the DNN architecture, most of the settings are automatic. However, the input vector can be converted using different scaling factor. Its distinct values are compared in figure 6.4. For lower numbers, the images are blurrier and more edges are detected. However, the algorithm is more vulnerable to noise. Depending on the environment conditions, choosing scaling factor in between of 0.3 - 0.7 seems resonable. Similarly to Canny, when using high values, HED detector loses important information.

In the next two observations, the two investigated methods are checked for pictures with different crack densities. The parameters are fixed – low threshold at 40 and scaling factor at 0.5. For results presented in figures 6.5b and 6.5c, both approaches are not precise. However, disregarding edges intensity, Canny detection is more similar to input image. For heavily damaged screen (see figure 6.5e and 6.5f), HED is less precise in sections where edges are very dense. On the other hand, Canny performs worse when edges are scattered. Further results (see figure 6.6) lead to similar conclusion. The biggest contrast occurs between figures 6.6e and 6.6f. Canny algorithm barely detects anything, while HED spots all the major cracks, as well as the shape of another mobile phone shooting that picture.

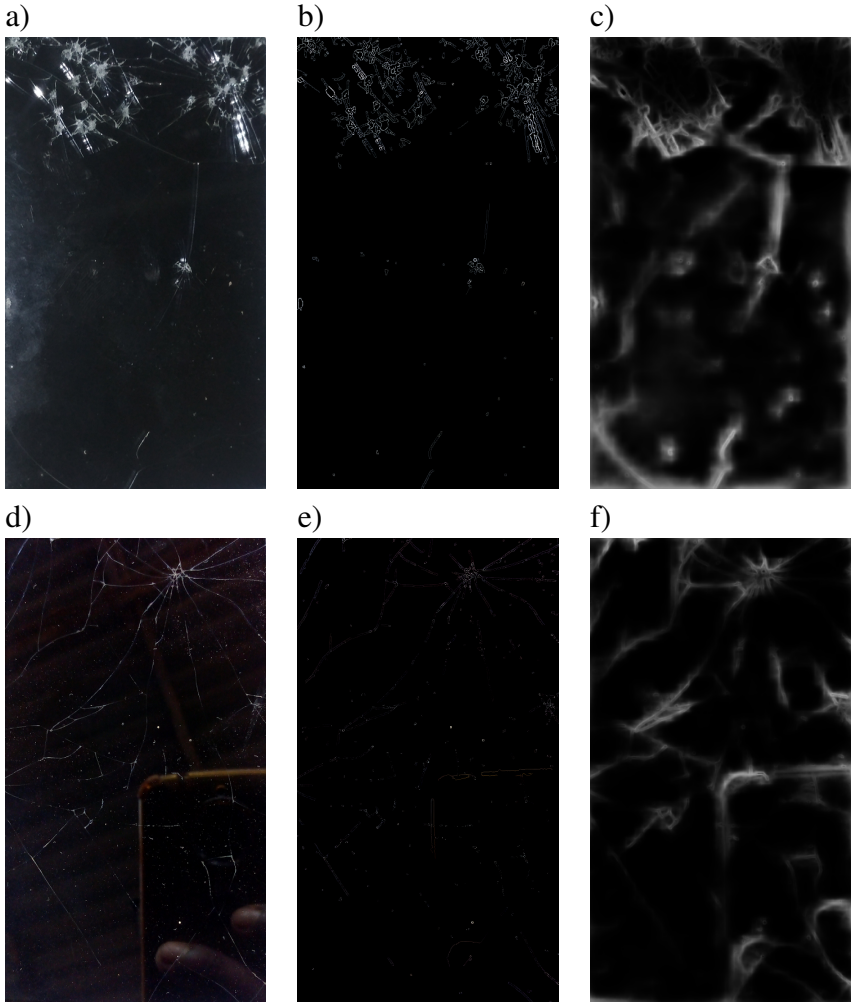

Figure 6.6: Comparison of HED and Canny edge detectors for lightly cracked screens: a) & d) input images, b) & e) Canny - low threshold = 40, c) & f) HED - scaling factor = 0.5

To sum up, the task of detecting cracks on mobile phone screens is not straightforward. Depending on density of cracks and pictures shooting conditions, different values of these algorithms' input parameters lead to satisfactory results. The HED method is stated to outperform Canny edge detection in case of natural images. However, in case of the analysed function it is not so obvious. Unifying the external conditions or setting constraints regarding screen quality would allow to choose well fitting parameters, which would surely lead to better results. For instance, narrowing down the detection for cases with thinner scratches, which might be harded to detect by human eye.

Chapter 7

Conclusion

In this thesis a significant potential of machine learning methods in enhancing the capabilities of digital cameras has been explored and demonstrated. By leveraging the power of these algorithms, it has been shown that digital cameras can surpass their traditional limitations and achieve remarkable advancements in image quality, scene recognition, image processing, and photographer's convenience.

Through an extensive literature review, a comprehensive understanding of the current state-of-the-art techniques in digital photography and machine learning was provided. It identified the challenges faced by conventional camera systems and highlighted the need for innovative solutions that broaden the capabilities of machine learning algorithms. The review delivered information about various techniques used for performing functions such as autofocus, white balance and edge detection. All of these methods has been widely analysed.

The hardware of digital cameras was also studied, starting with concepts of optical elements. It provided neccessary information about which camera settings can be continuously adjusted to control the device and obtain satisfactory results. Based on that information, various systems managing all the hardware could be implemented. Moreover, throughout the analysis of different components and firmware functions, the thesis content helps to understand and extend the general knowledge about the digital cameras and aspects of programming them.

The thesis contains proper preparation of environment needed for emulation. The camera source files, as well as all the menu entries can be accessed through the QEMU, but a significant limitation is the lack of processing the real time view from camera lens. Because of that, applications relying on manipulating the images before capturing them by releasing the shutter are more difficult to evaluate.

The extensions to firmware possibilities were successfully applied. As for improving the already existing functions, the work in this part has been mainly concentrated on boosting the cropmarks feature, allowing greater number of custom pictures to be used. Also, the prototypes of own solutions were implemented – the automatic white balance and phone screen protective film cracks detection deep neural networks. The first one allows to automatically adjust colour balance on the post capture image to look natural for human eye regardless of the actual shooting conditions. In the second method, the camera is employed as a specialistic diagnostic device. It detects the damage on screen surface and might be further commanded to alert about imperfections or provide feedback about their progression.

While this thesis has made significant steps in enhancing the capabilities of digital cameras through machine learning, there are still challenges that need to be addressed. The two own approaches were presented as prototype versions and were not tested on a physical camera. Performing such evaluation might be a good next step for further works. Moreover, there is a huge pool of other algorithms which could be converted to be compatible with native environment of cameras. Therefore, this work specifies an outline for many additional extensions, which could

be introduced. The limitation needing supplementary studies is memory allocation. The computational complexity and resource requirements of machine learning algorithms pose constraints on real-time implementation in camera systems. The applications need to be critically assessed in terms of memory needed for smooth operations. The results in this case might vary depending on device specifications. Moreover, some features might be too demanding to use in any of these embedded systems.

In conclusion, the findings of this thesis demonstrate the immense potential of machine learning methods in revolutionising digital photography. The integration of advanced algorithms has unlocked new possibilities for image quality enhancement, scene recognition, and image processing. As technology continues to be developed, it is expected that machine learning will take up an increasingly crucial role in shaping the future of digital cameras, enabling users to capture and create images with extraordinary precision and creativity.

Bibliography

[1] J. E. Adams. A fully automatic digital camera image refocusing algorithm. In *2011 IEEE 10th IVMSP Workshop: Perception and Visual Signal Analysis*, pages 81–86, 2011.

[2] M. Afifi. Semantic White Balance: Semantic Color Constancy Using Convolutional Neural Network. arXiv:1802.00153v5.

[3] M. Afifi and M. S. Brown. Deep white-balance editing. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1394–1403, Los Alamitos, CA, USA, jun 2020. IEEE Computer Society.

[4] M. Afifi, B. Price, S. Cohen, and M. S. Brown. When color constancy goes wrong: Correcting improperly white-balanced images. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1535–1544, 2019.

[5] S. Ansia and A. Aswathy. Single Image Haze Removal Using White Balancing and Saliency Map. *Procedia Computer Science*, 46:12–19, 2015. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace & Island Resort, Kochi, India.

[6] Arm GNU Toolchain. `https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain`, Date accessed: 20.05.2023.

[7] H. V. Athauda and N. Balasuriya. A New DSLR Camera Autofocusing Algorithm Based on Colour Information. In *14th International Conference on Industrial and Information Systems (ICIIS)*. IEEE, 2019.

[8] B. J. Baek, H. K. Lee, Y. Kim, and T. Kim. Mirrorless interchangeable-lens light field digital photography camera system. In *2013 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 2013.

[9] B. E. Bayer. Color imaging array, July 1976. US Patent 3971065A.

[10] J. Canny. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[11] K. Chen, A. Yadav, A. Khan, Y. Meng, and K. Zhu. Improved Crack Detection and Recognition Based on Convolutional Neural Network. *Modelling and Simulation in Engineering*, 2019:1–8, oct 2019.

[12] W. Chen, F. Kou, C. Wen, and Z. Li. Automatic Synthetic Background Defocus for a Single Portrait Image. Technical report, Hangzhou,, 2017.

[13] F. L. Community. Magic Lantern Firmware Wiki. Extending Magic Lantern. `https://magiclantern.fandom.com/wiki/Extending_Magic_Lantern`, Date accessed: 19.05.2023.

[14] F. L. Community. Magic Lantern Firmware Wiki. Magic Lantern API. https://magiclantern.fandom.com/wiki/Magic_Lantern_API, Date accessed: 19.05.2023.

[15] H. Eugen, Codrin, Donciu, and G. Asachi. Video distance measurement based on focus. In *11th International Conference and Exposition on Electrical and Power Engineering (EPE 2020)*. IEEE, 2020.

[16] C. Fan. Evaluation of machine learning in recognizing images of reinforced concrete damage. *Multimedia Tools and Applications*, Feb. 2023.

[17] M. Farghaly, R. Mansour, and A. Sewisy. Two-stage deep learning framework for sRGB image white balance. *Signal, Image and Video Processing*, 17:277–284, 2023.

[18] G. D. Finlayson and E. Trezzi. Shades of Gray and Colour Constancy. In *12th Color Imaging Conference: Color Science and Engineering Systems, Technologies, and Applications*, Nov. 2004.

[19] E. Fossum. Cmos image sensors: electronic camera-on-a-chip. *IEEE Transactions on Electron Devices*, 44(10):1689–1698, 1997.

[20] I. Free Software Foundation. GCC, the GNU Compiler Collection. https://gcc.gnu.org/, Date accessed: 16.04.2023.

[21] A. Fukuda. Image processing apparatus, method for controlling the same, and image capture apparatus, 2018. US Patent 2018/0124377.

[22] D. Guilan, T. Jinlan, Z. Suqin, J. Weidu, and D. Jun. Retargetable cross compilation techniques. *ACM SIGPLAN Notices*, 37(6):38–44, jun 2002.

[23] A. HajiRassouliha, A. J. Taberner, M. P. Nash, and P. M. F. Nielsen. Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms. *Signal Processing: Image Communication*, 68:101–119, 10 2018.

[24] H. Huang, C. Hu, T. Wang, L. Zhang, F. Li, and P. Guo. Surface defects detection for mobilephone panel workpieces based on machine vision and machine learning. In *2017 IEEE International Conference on Information and Automation (ICIA)*, pages 370–375, 2017.

[25] W. Jin, G. He, W. He, and Z. Mao. A 12-bit 4928 x 3264 pixel CMOS image signal processor for digital still cameras. *Integration*, 59:206–217, sep 2017.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[27] G. Li and Y. Yu. Deep Contrast Learning for Salient Object Detection, 2016. arXiv:1603.01976.

[28] Liu. *Application Specific Instruction Set Processors: Embedded DSP Processor Design*. 2007.

[29] A. Luštica. CCD and CMOS Image Sensors in New HD Cameras. In *Proceedings ELMAR-2011*, pages 133–136. IEEE, 2011.

[30] Magic Lantern installation on QEMU. https://foss.heptapod.net/magic-lantern/magic-lantern/-/blob/branch/qemu/contrib/qemu/README.rst, Date accessed: 23.05.2023.

[31] Magic Lantern Lua API Documentation. https://builds.magiclantern.fm/lua_api/, Date accessed: 19.05.2023.

[32] Magic Lantern Nightly - Install Guide. https://wiki.magiclantern.fm/install, Date accessed: 23.06.2023.

[33] Magic Lantern Website. https://magiclantern.fm/index.html, Date accessed: 15.04.2023.

[34] H. Mir, P. Xu, R. Chen, and P. van Beek. An autofocus heuristic for digital cameras based on supervised machine learning. *Journal of Heuristics*, 21(5):599–616, apr 2015.

[35] F. Mulla, S. Nair, and A. Chhabria. Cross Platform C Compiler. In *2016 International Conference on Computing Communication Control and automation (ICCUBEA)*. IEEE, 2023.

[36] S. Okada, S. Okada, N. Takada, H. Miura, and T. Asaeda. System-on-a-chip for digital still cameras with VGA-size video clip shooting. *IEEE Transactions on Consumer Electronics*, 46(3):622–627, 2000.

[37] OPEN AI LAB. Tengine. Documentation. https://github.com/OAID/Tengine, Date accessed: 20.05.2023.

[38] OpenCV. Canny Edge Detection. https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html, Date accessed: 05.06.2023.

[39] OpenCV. Canny Edge Detector. https://docs.opencv.org/3.4/da/d5c/tutorial_canny_detector.html, Date accessed: 06.06.2023.

[40] T. Orlowska-Kowalska and M. Kaminski. FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System. *IEEE Transactions on Industrial Informatics*, 7(3):436–445, aug 2011.

[41] K. A. Parulski and P. H. Jameson. Enabling technologies for a family of digital cameras. In C. N. Anagnostopoulos, M. M. Blouke, and M. P. Lesser, editors, *Solid State Sensor Arrays and CCD Cameras*, volume 2654, pages 156 – 163. International Society for Optics and Photonics, SPIE, 1996.

[42] K. Pauwels, M. Tomasi, J. Diaz Alonso, E. Ros, and M. M. Van Hulle. A Comparison of FPGA and GPU for Real-Time Phase-Based Optical Flow, Stereo, and Local Image Features. *IEEE Transactions on Computers*, 61(7):999–1012, jul 2012.

[43] PyTorch. Loading a TorchScript Model in C++. https://pytorch.org/tutorials/advanced/cpp_export.html, Date accessed: 23.05.2023.

[44] W. Rabadi, R. Talluri, K. Illgner, J. Liang, and Y. Yoo. Programmable DSP Platform for Digital Still Cameras. Technical Report 17,, 4 2000.

[45] C. G. Rahim, B. S. K. Soundra, and K. Rajan. Ricean Code Based Compression Method for Bayer CFA Images. Technical report, 2023.

[46] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[47] K. Simonyan and A. Zisserman. Very Deep Convolutional networks for large-scale image recognition, 2015.

[48] P. Siva, C. Russell, T. Xiang, and L. Agapito. Looking Beyond the Image: Unsupervised Learning for Object Saliency and Detection. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, jun 2013.

[49] SNAPSHOT. What is Dual Pixel CMOS AF?, December 2022. `https://snapshot.canon-asia.com/article/eng/canon-technology-explainer-what-is-dual-pixel-cmos-af`, Date accessed: 15.04.2023.

[50] K. R. Spring. Cameras for Digital Microscopy. In *Methods in Cell Biology*, pages 163–178. Elsevier, 2013.

[51] The QEMU Project Developers. Welcome to QEMU's documentation! `https://www.qemu.org/docs/master/#`, Date accessed: 23.05.2023.

[52] F. Vahid and T. Givargis. Embedded System Design: A Unified Hardware/Software Approach. In *Embedded System Design: A Unified Hardware/Software Approach*, 1999.

[53] C. Wang, Q. Huang, M. Cheng, Z. Ma, and D. J. Brady. Deep Learning for Camera Autofocus. *IEEE Transactions on Computational Imaging*, 7:258–271, 2021.

[54] S. Xie. Holistically-Nested Edge Detection. Source code for HED provided in the research article, `https://github.com/s9xie/hed`, Date accessed: 06.06.2023.

[55] S. Xie and Z. Tu. Holistically-nested edge detection. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1395–1403, 2015.

Appendix A

Description of attached CD/DVD

The attached memory disc contains all the important data related to this thesis in digital form. Among the contents, there is a `pdf` file with the text of the work. Moreover, source codes with test data for all the used algorithms are stored in the respective folders – `autoWBCNN` for automatic white balance deep CNN, `cracksDetectionCNN` for phone screen protective film cracks detection and `cropmarks` for the custom cropmarks feature. In case of functions incorporating neural networks, the data with architectures and weights of the models is also provided. Additionally, to adress the use of external libraries such as OpenCV by these methods, a text make file is included for compilation. Regarding the automatic WB deep CNN, code for tracing the model for C++ implementation is also attached.

Appendix B

Implementation manual

Instructions regarding configuring emulation of firmware with ML on QEMU are described in subsection 4.2.6. To include modified features, their source files needs to be stored in `/magic-lantern/src` folder before the ML compilation. Custom cropmarks images are loaded from `/magic-lantern/data/cropmks` directory.

This appendix was created with intention of explaining the process on implementing extensions on a physical digital camera. According to the ML guide [32], it is advised to fully charge the battery and restore the camera to default setting before starting the ML installation process. Also, all accessories should be removed for this operation. After completing the preparations, the first action is to rotate mode dial to Manual (M) position. It is crucial to carefully format the memory card, because the firmware add-on will be stored there. After acquiring proper Magic Lantern version archive file, its whole content needs to be unzipped to the `root` directory of the card. The installation starts with launching `Firmware Update` option from the Canon menu. Consequently following the instructions on screen leads to a green confirmation message eventually being displayed. Then, the camera needs to be restarted.

During the ML startup the user preferences should be restored from Canon menus entries. In LiveView mode the extensions can be toggled using `INFO/DISP` buttons. To access the additional menus, partially described in section 4.1, press `DELETE`. It is important to note that in case of using multiple memory cards, ML needs to be installed in all of them to maintain access to the firmware add-on. To update Magic Lantern in the camera, the old files need to be replaced with the new ones.