TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MATIAS LEHTINEN

VERIFICATION OF A MODULAR GRAPH BASED IMAGE PROCESSING SYSTEM

Master of Science thesis

Examiner: Prof. Karri Palovuori
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 9th December 2015

ABSTRACT

MATIAS LEHTINEN: Verification of a modular graph based image processing system
Tampere University of Technology
Master of Science thesis, 61 pages, 0 Appendix pages
May 2016
Master's Degree Programme in Electrical Engineering
Major: Electronics product design
Examiner: Prof. Karri Palovuori
Keywords: Verification, graph, image processing system, heterogeneous

Electronic devices today have become complex. Any non-trivial device consists of both hardware and software. Tightening time to market and cost requirements put pressure on the development process of the devices. Software and hardware needs to be developed concurrently and must be verified in an early phase of product development.

This thesis introduces a graph based image processing system. Image processing system is a complex system that usually consists of software, firmware and hardware. The possibilities and methods of graph verification are investigated in this thesis. Graphs can be used to handle the complexity of the system by encapsulating the functionality of the underlying implementations. Graphs provide modularity and configurability that can be utilized in the development and verification of the system. Reuse of software is increased due to the consistent and defined nature of graphs and their vertices. Software development shift left can be enabled by performing graph vertex verification in isolation by using pre-silicon development platforms.

In this thesis, image processing system graphs were also used in a real life product development project. Graph verification was initiated early in the product development. Shift left was exercised by utilizing the graph verification in several pre-silicon platforms. Functional, performance and stability testing was implemented. Both complete graphs and their vertices were verified in isolation. Graph verification provided many benefits to the product development. Implementations could be tested in several different environments in isolation using only a light test framework. Issues could be found and fixed early. Performance bottlenecks could be pinpointed and acted upon.

With the foundations laid in this project, it would be possible in the future to take more advantage of graphs. More advanced automated image quality testing would allow efficient verification. Finer granularity graphs would allow more configurability and more focused testing. Shift left could be further increased by adapting the development of the algorithms to use graphs. This would lower the gap between algorithms and actual vertex implementations and also introduce the available test infrastructure to algorithm development.

TIIVISTELMÄ

MATIAS LEHTINEN: Modulaarisen graafipohjaisen kuvankäsittelyjärjestelmän verifiointi
Tampereen teknillinen yliopisto
Diplomityö, 61 sivua, 0 liitesivua
Toukokuu 2016
Sähkötekniikan koulutusohjelma
Pääaine: Elektroniikan tuotesuunnittelu
Tarkastajat: Prof. Karri Palovuori
Avainsanat: Verifiointi, graafi, kuvankäsittelyjärjestelmä, heterogeeninen

Nykyaikaiset elektroniset laitteet ovat monimutkaistuneet. Mikä tahansa arkipäiväinenkin laite koostuu sekä laitteistosta että ohjelmistosta. Samalla vaatimukset kustannuksista ja ajasta alkukehityksestä markkinoille siirtymiseen ovat kiristäneet kehitysprosessia. Ohjelmisto ja laitteisto on kehitettävä rinnakkain, ja nämä on kyettävä verifioimaan laitekehityksen varhaisessa vaiheessa.

Tämä diplomityö esittelee graafipohjaisen kuvankäsittelyjärjestelmän, joka koostuu tyypillisesti ohjelmistosta, sulautetusta ohjelmistosta ja laitteistosta. Työssä tutkitaan graafien verifioinnin mahdollisuuksia ja menetelmiä. Graafien avulla järjestelmän monimutkaisuutta hallitaan eristämällä ja piilottamalla itse toteutus. Graafien modulaarinen lähestymistapa tuo etuja järjestelmän kehittämiseen ja verifioimiseen. Ohjelmiston uudelleenkäyttömahdollisuudet kasvavat graafien määritellystä ja johdonmukaisesta luonteesta johtuen. Ohjelmistokehityksen aikaistaminen on mahdollista verifioimalla graafien solmuja eristyksissä käyttäen kehitysympäristöjä ilman valmista laitteistoa.

Tässä työssä kuvankäsittelyjärjestelmägraafeja käytettiin myös todellisessa tuotekehityshankkeessa. Graafien verifiointi aloitettiin tuotekehityksen varhaisessa vaiheessa. Ohjelmistokehityksen aikaistaminen oli mahdollista hyödyntämällä graafien verifioimista useissa kehitysympäristöissä. Toiminnallisuus-, suorituskyky- ja vakaustestaus toteutettiin. Sekä kokonaisia graafeja että yksittäisiä solmuja verifioitiin eristyksissä. Graafien verifiointi mahdollisti useita etuja tuotekehityksessä. Toteutukset oli mahdollista testata useissa eri ympäristöissä käyttäen hyödyksi kevyttä testiohjelmistoa. Ongelmat löydettin ja korjattiin varhaisessa vaiheessa. Suorituskykypullonkaulat oli mahdollista havaita ja purkaa.

Tämän työn pohjalta on tulevaisuudessa mahdollista hyödyntää graafeja yhä paremmin. Kehittyneemmät automatisoidut kuvanlaatutestit mahdollistaisivat tehokkaan verifionnin. Hienojakoisemmat graafit mahdollistaisivat joustavamman konfiguroinnin ja kohdistetumman testauksen. Ohjelmistokehityksen aikaistamista olisi mahdollista lisätä entisestään ottamalla graafit käyttöön myös algoritmikehityksessä. Tämä pienentäisi kuilua algoritmien ja solmutoteutusten välillä tuoden samalla graafitestauksen hyödyt algoritmikehitykseen.

PREFACE

This thesis has given me an opportunity to research the area that I have been working on, and writing the thesis has been a great experience. I have been able to broaden my knowledge on the subject, and I have learned much new that can be applied in the future.

I would like to thank my colleagues for all the support that I have received during the writing of my thesis. I would especially like to thank Teemu Tuominen and Janne Kotka for instructing me and supporting me to keep the thesis writing ongoing.

I would also like to thank my family and my friends for their support. Finally, I would like to thank my beloved fiancée Enni for supporting, loving and understanding.

Tampere, 17.4.2016

Matias Lehtinen

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

3A	3 Algorithms
ADC	Analog-To-Digital Converter
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BSD	Berkeley Software Distribution
CCD	Charge Coupled Device
CFA	Color Filter Array
CMOS	Complementary Metal-Oxide Semiconductor
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DMOS	Difference Mean Opinion Score
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
HDR	High Dynamic Range
IQM	Image Quality Metric
MOS	Mean Opinion Score
MSE	Mean Squared Error
MTBF	Mean Time Between Failures
OS	Operating System
PSNR	Peak Signal-to-Noise Ratio
RTL	Register-Transfer Level
SoC	System on a Chip
SSIM	Structural Similarity Index
V4L2	Video4Linux2
VIF	Visual Information Fidelity
VLIW	Very Long Instruction Word
VSNR	Visual Signal-to-Noise Ratio

1. INTRODUCTION

Electronic devices today are becoming more and more complex, including more functionality and more performance demanding features than ever. However, the cost to create such devices is not allowed to rise. Rather, the costs need to go down, while the time to market for products is squeezed into a minimum. Competition, especially in the lower segment of devices, makes sure of this with more and more companies emerging and competing in the same technological area.

Software is becoming pervasive in the products we use today. Any non-trivial device today consists of both hardware and software. There are several reasons for the popularity of software. Software is flexible, it is in general easy to develop, relatively easy to reuse and most importantly cheap when compared to pure hardware [28]. In other words, the easiest way to add features is to use software. However, software has its disadvantages when compared to hardware. The two most important factors are performance and energy efficiency [28]. A hardware implementation surpasses software in these and will be present in devices requiring either of the advantages.

To make things more complicated, software can be run on several platforms. There may be multiple CPUs (Central Processing Unit), GPUs (Graphics Processing Unit) or DSPs (Digital Signal Processor) that all are programmed differently. Or the software may only be some control logic for a fixed function hardware block in a SoC (System on a Chip), an ASIC (Application Specific Integrated Circuit) or an FPGA (Field-Programmable Gate Array). [10]

All the different processors and hardware blocks have their advantages and disadvantages. Depending on the requirements of a certain computation task, the most suitable compute device is chosen. As the nature of the tasks that need to be performed vary from one to another, the best results can usually be received by utilizing multiple compute devices. The combination of different compute devices is called heterogeneous computing, and several standards are emerging with the aim of simplifying the usage and unifying the APIs (Application Programming Interface) [24, 30].

In order to succeed in the increasing demands to the cost of the device and the time to market requirements, while still being capable of handling the complexity of the devices, several aspects in the development process need to be thought of. Existing implementations need to be reused as much as possible with the minimum required changes. Different use cases need to be easily configured from whatever building blocks are available. Most importantly, components need to be functionally verified as early as possible.

Traditionally hardware and software has been developed sequentially [53]. Software development has only been able to commence after the real hardware is available. In the past, when the software stacks were relatively simple, this may not have been a problem. However, with today's complex hardware and software stacks in addition to the ever more demanding time to market requirements, this process has become infeasible.

In order to squeeze the total time required to develop a product into the time frame required, it is necessary to start developing software earlier than traditionally is done. In this context, shift left means moving development that has traditionally begun after the actual hardware is available in parallel with the development of the hardware. This however raises important questions. How to develop program code for a platform that does not exist yet? How to verify that the hardware is going to be implemented in a way that the users expect? How are the different blocks consisting of software, firmware and hardware going to work together in the final product?

The purpose of this thesis is to investigate the aforementioned problems in the context of an image processing system for mobile devices. Image processing pipeline is an example of a highly complex and computationally demanding system and usually includes software, firmare and hardware. Especially in mobile devices, both the small physical size of the device and the rather restricted hardware budget create pressure on the actual image processing. Not only does the image format need to be converted and compressed, but also multiple correction algorithms need to be applied in order to provide an image with satisfactory quality [48].

Graphs are introduced as an option to handle the requirements of different use cases and products. Graphs help to cope with the increasing complexity of both hardware and software. They provide a generic way to integrate complex processing entities into the software stack. At the same time graphs allow the required modularity and flexibility without the burden of increasingly complex software exposed to the software stack above. The increased modularity allows convenient reuse of software and hardware building blocks. Moreover, graphs allow development and testing to begin very early in the development phase continuing all the way until the final product.

In this thesis, a standard image processing pipeline is first explained. Graphs are then introduced and their usage in image processing pipelines is explained. The possibilities and methods of verifying image processing graphs are investigated. Verification is then used in a real product development project utilizing the modularity obtained from the graphs, starting from the early phases of development and continuing all the way until the final product. Finally, several future improvements and possibilities are considered.

The thesis is organized as follows. Chapter 2 introduces and explains the image capture pipeline. Chapter 3 provides information about graphs and their usage in the image capture pipelines. Verification of image processing graphs is then investigated in chapter 4. Chapter 5 follows a product development project using image processing graphs. In chapter 6, future improvements are discussed. Finally, chapter 7 concludes the thesis.

2. IMAGE CAPTURE PIPELINE

In digital photography, image processing plays a key role in the process of obtaining actual human readable images to the screen of a camera. What might seem a simple process from the point of view of the user, is in fact often a very complex one, requiring seamless functionality between different hardware and software blocks. [3]

The image capture process begins with capturing light through the lens system onto a sensor. The light received by the sensor will then be converted into electric signals and is furthermore passed forward to be processed. A series of analyzing and processing is performed, until finally a complete image with the required corrections, resolution and format is displayed on the screen or saved to the memory of the device. [48] An example image pipeline is shown in figure 2.1.

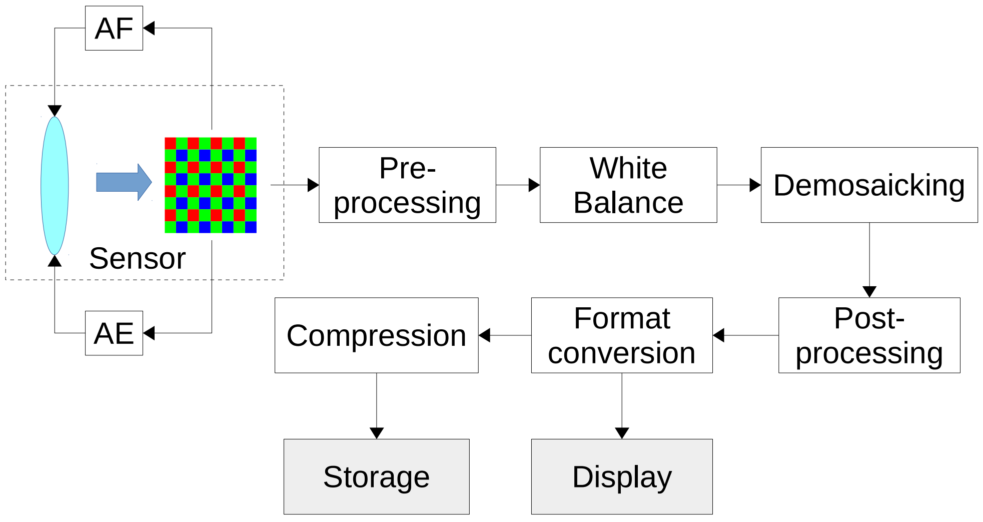


Figure 2.1 *An example camera pipeline*

This example pipeline is only a basic representation of what an image capture pipeline may look like. The amount of different computations, and the order of those computations, change from one product to another. There is no standard image capture pipeline, but each pipeline is designed to fit the needs and constraints of a certain product. However, using some common elements found in many real image capture pipelines, a basic example pipeline can be created. The following sections describe the different blocks found in the figure 2.1 in more detail.

2.1 Image quality

In an ideal case, the captured image that is viewed by the user is an exact representation of the scene that was photographed. However, in a real world the image is subject to many different distortions that happen during the various stages of the image capture pipeline. Image acquisition, processing, transmission and compressing all add some kind of an error to the data. Many of the steps in the processing pipeline try to correct these errors aiming to an enhanced image quality. Because of the limitations in each product, the final outputted image will always contain some errors. [58]

In order to investigate the quality of a captured image, evaluation is required. Image quality assessment can be split into two methods: subjective and objective assessment. In a subjective evaluation [26] the image quality is assessed by humans, while in an objective evaluation [55] the quality is assessed via metrics predicting the image quality automatically. Both methods have their strengths and weaknesses. Depending on the need, either or both methods can be used.

Since most of the images are processed to be eventually viewed by human beings, subjective assessment produces the most reliable outcome [58]. Traditionally, in a subjective assessment the overall quality for the images is evaluated by multiple viewers. Test sequences are presented to the subjects using a specified method and are then rated for quality [26]. Of these ratings a metric called MOS (Mean Opinion Score) or DMOS (Difference Mean Opinion Score) can be calculated to determine the quality of the image [45]. While subjective assessment is reliable, it is slow and expensive and can be very inconvenient, therefore being not suitable for many applications.

Objective assessment on the other hand aims to accurately and automatically measure the different aspects of what human viewers perceive as being good quality. Different metrics are available to be used to measure the quality. The metrics can be classified based on the availability of the perfect quality reference image. Full reference quality assessment assumes that a perfect quality reference image is available, reduced reference quality assessment only has partial data available from the reference image, and no reference quality assessment has no reference image at all. Objective image quality assessment without any reference data is very difficult and hence full reference assessment is mostly used. [58]

Many objective image quality assessment algorithms have been proposed. Traditionally, the most common full reference image quality assessment metrics have been MSE (Mean Squared Error) and PSNR (Peak Signal-to-Noise Ratio). MSE can be presented the following way [58]:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2 \tag{2.1}$$

where N is the number of pixels in the image and x_i and y_i are the i-th pixels in the reference and distorted images, respectively.

PSNR can be presented based on MSE the following way [58]:

$$10log_{10}\frac{L^2}{\text{MSE}} \tag{2.2}$$

where L is the dynamic range of the pixel values.

The issue with MSE and PSNR is that while they do measure the signal fidelity, they do not accurately predict the human perception of the image quality [56]. This difference is a result of the human visual system characteristics which perceive certain errors in the image differently than what could be interpreted directly from the data.

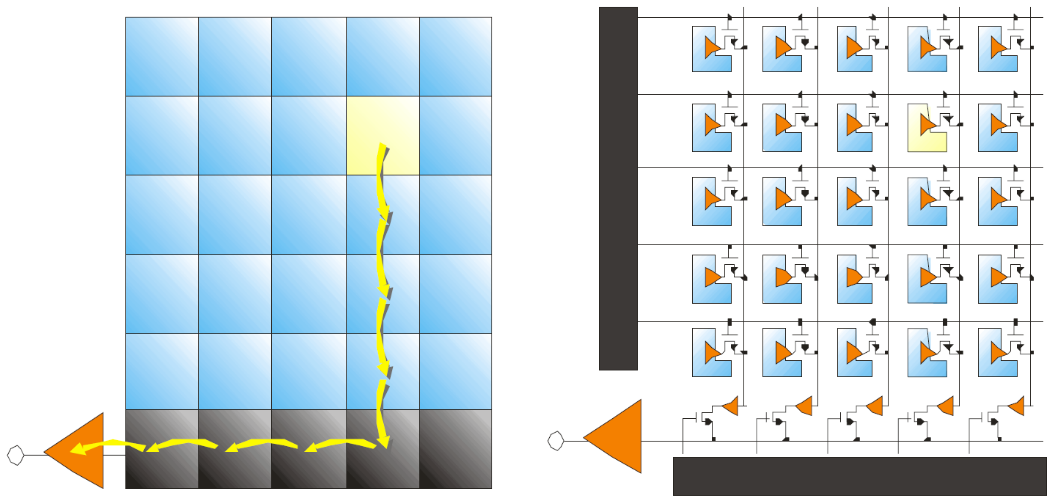
More advanced metrics that correlate better with the actual perceived image quality have been proposed [15, 40, 45, 56]. SSIM [57] (Structural Similarity Index) takes advantage of the fact that the human visual system is highly adapted for extracting structural information. VIF [49] (Visual Information Fidelity) measures the loss of information extracted by the brain by modeling a natural image source, distortion channel and the human visual system. VSNR [9] (Visual Signal-to-Noise Ratio) is a wavelet-based measurement that takes into account the human visual system capabilities of seeing distortions.

Research shows that with the more advanced objective image quality metrics a better correlation can be achieved between objective and subjective evaluation results than what is achievable with the traditional metrics [40]. However, objective assessment still does not match the results of subjective evaluation. Nevertheless, especially with the recent developments in the field, objective assessment can be utilized in the image quality evaluation.

2.2 Image sensors

Digital cameras use most commonly two types of sensors: CCD (Charge Coupled Device) and CMOS (Complementary Metal-Oxide Semiconductor). The two types of sensors both sense light through similar mechanics utilizing the photoelectric effect. For each pixel in the sensor, received photons are transformed to an electric charge. The difference between these sensors is how the information stored as a charge is converted to voltage and transferred out of the pixel array. [47, pp. 55-61].

For a CCD sensor, charge is shifted from each capacitor (pixel) to another and the conversion from the charge to voltage is done in the last capacitor. Once this process has been repeated enough, the entire contents of the sensor have been read. For a CMOS sensor, each pixel itself contains a circuit that converts the charge to a voltage. These values can then be read one pixel at a time, allowing also to only read part of the image data if required. [12, p. 5] Both types of sensors can be seen in figure 2.2.



(a) CCD **(b)** CMOS

Figure *2.2 CCD and CMOS image sensors [12, p. 6]*

Both sensors have their strengths and weaknesses. CCD has been around in digital cameras for a longer period and has been traditionally providing a better image quality. However, CMOS technology has been rapidly developing and has narrowed down much of the quality gap between the two types of sensors. Because of the limitations in the CCD technology, such as price and power usage, CMOS has been widely used in the lower-end segment of digital cameras, while CCD has been the choice for higher end products. [12]

Both of the sensor types described above are monochromatic and hence cannot make a distinction between different wavelengths of light. There exists several ways to overcome this issue. One way is to use multiple separate sensors, each dedicated to a single color, so that when all the images are combined a full colored image can be reproduced. Another method is to use the silicon itself as a filter since the light penetration of the silicon is dependent on the wavelength of the light. This method was used in the Foveon X3 sensor [17], where three photodiodes were stacked on top of each other, each capturing red, green or blue color. However, the most common solution is to pass the light through a CFA (Color Filter Array), so that each pixel only receives light of a certain color, and then later interpolate the full colored image. [12, pp. 8-9]

The most common color filter array filter is the Bayer filter [4], which has 50% green, 25% red and 25% blue pixels [42, pp. 3-7]. This filter is also called GBRG, GRBG, BGGR or RGGB depending on the order of the pixels in one 2x2 bayer quad. The reason for twice the amount of pixels used for green as is for red or blue, is because the human eye behaves in a similar way, being more sensitive to green than to other colors [22, pp. 44-45]. An example color filter array can be seen in figure 2.3. This color filter array is RGGB, as can be seen from the pixel order starting from the upper left corner.

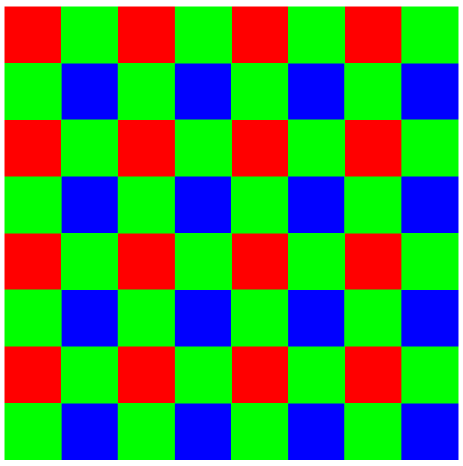


Figure 2.3 *Bayer RGGB color filter array*

Once all the voltage data is read, the pixel values are passed through an ADC (Analog-To-Digital Converter), which transfers the generated digital information further to be processed. [12, p. 5]

Two types of sensor solutions exist in the digital camera domain. SoC sensors contain an integrated SoC, which performs all or part of the required processing internally in the sensor module, and only needs to be controlled by the host as required, providing images ready to be displayed on the screen or saved to the device. Raw image sensors on the other hand only output the captured raw data for which additional processing is still needed. [12, p. 5] In this thesis, the word sensor refers to a raw sensor without any additional processing capabilities.

2.3 3A

3A (3 Algorithms) is the automatic selection of the control values for focus, exposure and white-balance [1]. These three key values greatly affect the quality of the output images [38, p. 694]. Especially in mobile cameras, a majority of the images is taken by simply pointing the device and clicking the shutter release button instead of carefully selecting the suitable values for each by hand. This makes the functionality of 3A, its convergence speeds and human noticeable behaviors in different scenarios crucial to a camera device.

2.3.1 Auto exposure

Auto exposure means adjusting the amount of light on the sensor in order to be able to utilize the full dynamic range of the sensor. The adjustment can be done via aperture setting (unless the device has a fixed aperture), shutter speed and ISO speed. Over-exposed and under-exposed images result in lost details that are impossible to be corrected later, so being able to set the exposure values correctly is crucial. [47, p. 238] [48, p. 36]

The image received from the sensor is divided into multiple sub-blocks that are also called AE windows. For each window the average and peak luminance values are calculated. The best fitting combination of the configuration values are then generated automatically by a special algorithm. The values are used in the image sensor for the next image to be captured. This loop is repeated continuosly, as the environment can change from one frame to another. [47, pp. 238-239] [48, p. 36]

In a scene that has large brightness variations, the limitations of the sensor results in a situation in which details are inevitably lost, as the dynamic range of the sensor is insufficient. A solution to this problem is called HDR (High Dynamic Range). In HDR images, multiple frames are captured vith varying exposure settings, each frame having the correct values to capture a certain section of the scene. The information from the images is then combined together using a special algorithm, resulting in an image with a substantially higher dynamic range than what would have been achievable with capturing only a single frame. [29]

2.3.2 Auto focus

Auto focus, as the name suggests, is focusing the image automatically to a certain point by moving the lens, so that the desired object in the image is not blurred. Two types of auto focus exists: passive auto focus and active auto focus. The difference between these two types is that active auto focus sends some kind of a signal to estimate the distance to the subject, while passive auto focus relies on the information obtained from the image to achieve the desired focus. [48, pp. 36-37]

Active auto focus can send ultrasound [6] or light [59], either in the infrared or visible range, to the object where the focus is desired. The signal is then reflected back from the object to the camera. From the intensity or angle of the returning signal, or the time it takes the signal to return back to the camera, distance between the camera and the object can be calculated, and the lens moved accordingly. [48, p. 36]

In passive auto focus, the captured image is first analyzed for the focus, after which the lens is moved a bit and the image is analyzed again. Depending on whether the move resulted in a better or worse focus, the lens is moved accordingly trying to achieve the correct focus. This loop is iterated as long as it is required to achieve a satisfactory focus. [47, pp. 240-241] [48, pp. 36-37]

Two types of analysis techniques are currently used in passive auto focus. One way is to use the contrast found in the image to find out the current focus [20]. The intensity between adjacent pixels increases when the area is focused, so the correct focus can be found by looking for the maximum intensity difference. The other method is to use phase detection auto focus [51], which compares the light coming through the opposite sides of the lens by using two or more small image sensors. If the object is not in focus, the images are out of phase. The correct lens position can be calculated from the resulted values. [48, pp. 36-37]

2.3.3 Auto white-balance

In the human vision, white is perceived as white even under different lighting. Fluorescent, incandescent and daylight are all of different color, but for example white walls are still seen as white. This is called color constancy. For digital cameras the same is not true. Color intensities will be captured exactly the way the objects emit the different wavelengths of light. A white wall in a red light will be a red wall in the picture. In order to correct this, white balance correction is required. [42, pp. 267-268]

White balance correction is all about neutralizing the color of the light source in the image. First the image needs to be analyzed and the color of the surrounding light needs to be detected. As the automatic detection has its flaws, this can usually be also selected by the user from a set of possible pre-defined light sources. After the color of the surrounding light is known, the image can be corrected to look like it had been taken in white light. [42, p. 268]

The actual correction usually happens by setting the gains of each independent three color signals. Various algorithms exist to produce the correct gain values. One example is the gray world assumption, in which it is assumed that the mean values for all the three color channels should be equal. The gains for two of the three channels can hence be adjusted so that all the three channels have an equal mean value. [42, p. 280]

2.4 Preprocessing

The image received from the sensor is in a raw bayer format, each pixel only containing the amount of light received through the specific color filter on top of the given pixel as was described in section 2.2. This data is by no means perfect and requires a substantial amount of processing to produce a satisfactory image [48, p. 38].

Preprocessing deals with the defects in the captured bayer image before a full color image is produced by demosaicking. As all the original pixel values are still available, defects related to individual pixels need to be handled while the image is still in this format and the errors are not mixed with other pixels during interpolation. Another reason for performing corrections as early in the processing chain as possible is to avoid amplifying the defects by algorithms later on. On the other hand, some corrections cannot be performed in the bayer format since the algorithms require the full color image and need to be performed after the demosaicking. [42, pp. 7-9] [48, p. 38]

Some common corrections usually performed in the preprocessing phase are listed below.

Defective pixel correction Some of the pixels in the sensor may be defective. Without correction these pixel values would show up as errors in the image after the demosaicking, as the values of the color image pixels are interpolated from the neighboring pixels. Defective pixel values can be replaced using average values from the surrounding pixels. [44]

Linearization Not all the data received from the sensor is linear because of the electronics involved in the sensor. Therefore, linearization is required, which means transforming the raw measured nonlinear data into a linear space. [48, p. 38]

Dark current compensation Receiving no light at all into the sensor does not actually produce zero values as could be expected. This phenomenon happens because a current is flowing through the photodiodes even when no photons are received. To account for this, an average dark current value can be calculated, which then needs to be subtracted from the values received from the sensor in order to capture the black level correctly. [42, p. 70] [48, p. 38]

Lens shading correction The optics of the camera also produce some artifacts. Lens shading, also called vignetting, appears in the corners of the image due to the physical properties of the lens. This needs to be linearized, requiring the usage of lens shading correction. [60]

Flare compensation In images that have a high source of light, light scatters in the optics of the camera, which causes a shift in the received energy. This distortion is called the lens flare, for which a flare compensation is needed. Plainly, flare compensation reduces the pixel values in the image near the affected pixels. [48, p. 38]

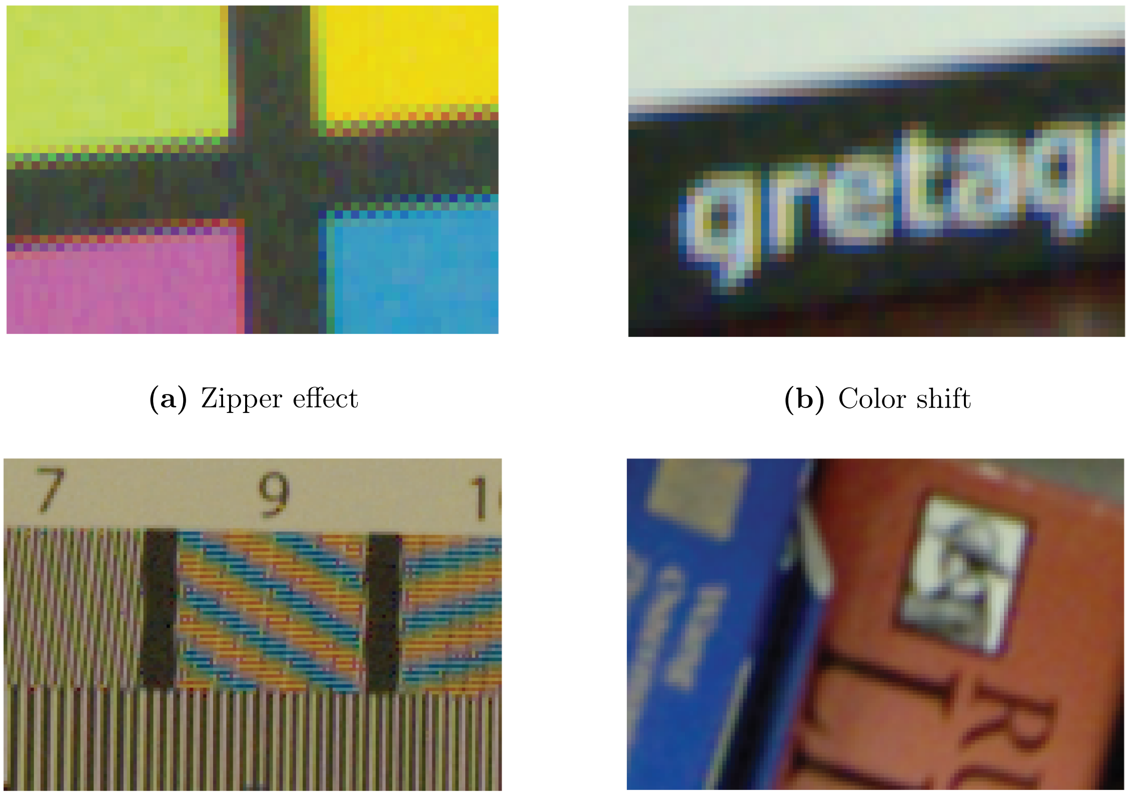
2.5 Demosaicking

After the required analyzations and corrections are finished in the RAW bayer format, the image needs to be converted from a grayscale bayer to a full color image. This step is called demosaicking, and is usually the most compute intensive step in the processing pipeline [22, p. 44].

In order to be able to represent a full color image, each pixel needs three color values. Traditionally in computer images the values have been red, green and blue. In single sensor systems, a color filter array is used as is explained in section 2.2. This means that for each pixel in the image, two out of three color image values are missing. The missing values need to be estimated based on the rest of the image. [22, p. 44]

Demosaicking is also called color filter array interpolation since each missing value needs to be interpolated from the surrounding pixels. Naturally, whenever data is interpolated, accuracy is lost as the result is only an estimation of the original. Several different algorithms exist to perform the demosaicking. Different algorithms may trade image quality in favor of performance while some may produce more artifacts than others to the image that need to be corrected in post-processing. [22]

Demosaicking has a high impact to the rest of the pipeline and to the overall quality of the outputted image. The output quality has to meet the requirements, and any artifacts generated by the algorithm need to be corrected later. Some common artifacts are shown in figure 2.4.



(a) Zipper effect **(b)** Color shift

(c) Aliasing artifact **(d)** Blur

Figure 2.4 *Common demosaicking artifacts [42, p. 14]*

2.6 Postprocessing

After all the required preprocessing steps and the successive demosaicking is complete a viewable image exists. This image is in a color space that has all the required color components for each pixel in order to reproduce a complete color image. However, some corrections are still to be done in this step now that the proper image with separate color channels is available. Also, the previous steps in the pipeline introduce some artifacts and distortions that need to be corrected in order to produce an image with an acceptable quality. This part of the pipeline is called postprocessing.

Some common corrections usually performed in the postprocessing phase are listed below.

Demosaicking artifact correction Some of the artifacts present in the image depend on the processing algorithms used. For example, the demosaicking algorithm may produce, depending on the input image and the algorithm used, zipper effect, color shift, aliasing artifacts or blur (figure 2.4). These artifacts need to be corrected while still retaining the sharpness of the image. [48, pp. 40-41]

Geometric distortion correction The lens optics introduce distortion that affects the image. Geometric distortion, for example barrel distortion, misplaces information geometrically. In other words, pixels in the image present information that should be elsewhere. [61] In addition, various wavelengths behave differently in the lens because of different refraction indexes, creating chromatic aberration where different colors are not focusing correctly in the same place. [36] Geometric distortion correction is performed to correct these artifacts.

Color correction Colors captured by the sensors do not match that what is considered a pleasant image by humans since the color sensitivity of the sensor is different than that of the human eye. The captured image needs to be transformed into a calibrated color space which is the purpose of color correction. [42, pp. 77-78] [48, p. 40]

Edge enhancement The human eye is very sensitive to sharp edges [48, p. 41]. Therefore, a step that performs edge enhancement, or in other words sharpening, is usually performed. The purpose of this processing is to make the image look sharper by amplifying the high-frequency components of the image. [42, p. 80]

As part of the correction algorithms required depend on the previous steps in the pipeline, postprocessing needs to be adapted to the sensor, preprocessing and demosaicking algorithms used. Therefore, there may be great variance in the post processing between devices, and each device needs to be configured separately.

2.7 Format conversion

Depending on the device and the use case of the processing pipeline, additional color space conversions may be required. While the RGB format may be useful in some of the aforementioned image processing algorithms, it is not a practical format to store the data.

For RGB formats, each color channel has the equal amount of information. For example RGB888 uses a total of 24 bits to store the information for each pixel [52]. The larger the images are, the higher the burden is to store, transmit and display the data. Therefore, other formats with a lower bit rate are needed.

2.7.1 YUV

A frequently used way to reduce the amount of data used for an image is to take advantage of the biological capabilities of the human vision system. The human eye is less sensitive to color differences than it is to luminance differences [25]. The method taking advantage of this fact reduces the amount of data of color in the image, while still keeping the same amount of data for the brightness. This is called chroma subsampling [14].

Obviously this kind of data reduction is impossible to do in the RGB color space as the brightness of each pixel is a combination of each color value, all three represented separately. The solution for this is to use an image format where luminance and chrominance values are separated. These formats are referred as $Y'C'_BC'_R$, where Y' is the luminance, C'_B is the difference between blue and the luminance, and C'_R is the difference between red and the luminance. [27, pp. 16-18]

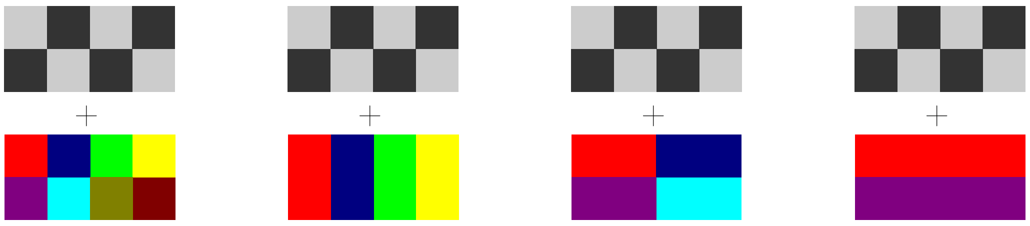
After the chrominance has been separated from the luminance, the data of the chrominance can be reduced while still sustaining the original luminance information. Usually this chroma subsampling is notated as a three part ratio *A:B:C* [52]. The ratio tells us that in a sample region that is A pixels wide and 2 pixels high, there are B horizontal chroma samples, and C changes in the chrominance values between the first and the second row. Some commonly used examples are given below and can be seen in figure 2.5.

4:4:4 Each pixel has both luminance and chrominance samples. This format has no chroma subsampling at all. Full horizontal and vertical resolution.

4:4:0 Each pixel in the first row has chrominance samples, but these are shared between two rows. Full horizontal resolution, ½ vertical resolution.

4:2:2 For 4 pixels in the first row, two chrominance samples exist. Each row has its own samples. ½ horizontal resolution, full vertical resolution.

4:1:1 For 4 pixels in the first row, only a single chrominance sample exists. Each row has its own samples. ¼ horizontal resolution, full vertical resolution.



(a) 4:4:4 **(b)** 4:4:0 **(c)** 4:2:2 **(d)** 4:1:1

Figure 2.5 *Different YUV subsampling ratios. The above pixel array describes the luma samples and the below pixel array describes the chroma sample pairs.*

Depending on the required quality of the output and the restraints of the storage size of the image, an appropriate subsampling ratio is chosen. Of course it goes without saying that the less there are chroma samples, the lower the quality of the image will be.

Different YUV formats exist even with the same subsampling ratio. Luma and chroma values can be combined into macro pixels (packed formats), or they can be in separate planes (planar formats). The chroma planes may also be interleaved in different ways or the image may have some padding. The range of different YUV formats is vast. [16]

2.7.2 Compression

While the chroma subsampling in the YUV formats lowers the bandwidth needed for the images and may be enough to satisfy the needs of storing, transferring and displaying a single image, this still leaves the images with a storage size too large for many uses. For example, in mobile devices the available disk space for storage may be a bottleneck especially in the lower end segment. Compression is needed to overcome this problem.

A commonly used compression method for digital images is JPEG (Joint Photographic Experts Group) [54]. This method is lossy, meaning that information is lost in the process. However, JPEG takes advantage of the human visual system in a way that makes high compression possible without reducing the quality of the image below a satisfactory level.

JPEG compression is done to images in the YUV color space. The chroma subsampling already explained in the section 2.7.1 is also usually used in the JPEG compression. The YUV image is then split into 8x8 blocks called macroblocks. After this, macroblocks are converted into frequency domain by DCT (Discrete Cosine Transform). The values are then quantized by dividing each component by a constant and rounding them to the nearest integer, reducing information in the high frequency domain. The resulted values are finally encoded using run-length encoding and huffman coding. [54]

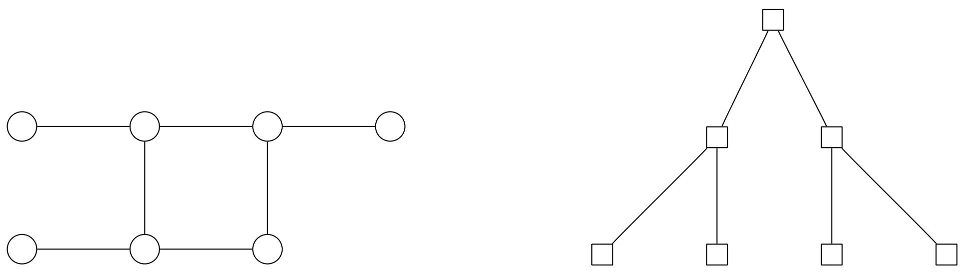
JPEG standard supports different compression ratios enabling the user to select the best tradeoff between image quality and the resulting image size. In practice, the compression ratio is chosen by setting the quantization table [54], which is the only lossless part of the JPEG encoding apart from the chroma subsampling. Images compressed with a low ratio are usually indistinguishable from uncompressed ones by the naked eye, while higher ratios might still be adequate for certain use cases but have noticeable artifacts.

Video capture, where multiple frames are captured and combined together, requires much more information to be stored than what is the case with still images. Therefore, compression is especially important with videos. JPEG can also be used for video compression [39]. However, it is quite inefficient since no temporal redundancy is used. This means that pixels having the same data from one frame to another is not exploited in the compression.

Several compression standards specifically aimed to be used for video compression exist. Common video coding formats include Motion JPEG [39], MPEG-4 [46] and H.264 [2] among others. Motion JPEG is the video compression based on the JPEG image compression. MPEG-4 and H.264 on the other hand use several frames to compress the stream, taking advantage also of the temporal redundancy in addition to the spatial redundancy.

3. GRAPHS

Graph is a set of points and lines connecting the points. Graphs can be used to describe many different situations in real word and have applications in a variety of different areas [8]. An example application of the graphs in the physical world is to represent all the possible routes from one place to another like a subway map. Another everyday example is a company organization chart. Example illustrations of these graphs can be seen in figure 3.1



(a) Route map **(b)** Organization chart

Figure 3.1 *Example applications of graphs*

These graphs can be used to solve problems like what is the fastest or shortest route from one place to another, or who is the manager of a given person in the organization. Of everyday graphs similar to these, a mathematics branch called graph theory has emerged, dating back to 1735 and the Königsberg bridge problem [8]. The following sections give an overview of the graph theory and their applications in the computing domain.

3.1 Basic graph theory

A graph consists of a finite set of objects called vertices and a finite set of unordered pairs of the vertices. A graph can be represented as a diagram where vertices are points, and the elements of the ordered set, called edges, are lines connecting the points. [7, p. 1] [13, p. 2] An example graph can be seen in figure 3.2.

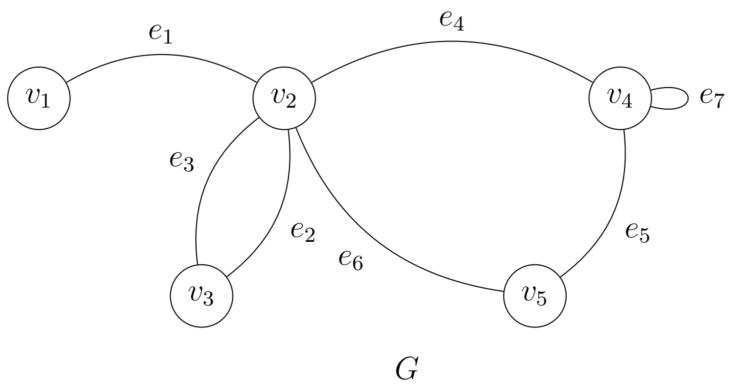


Figure 3.2 *Example graph diagram*

In this graph, V is the set of vertices, and E is the set of edges. The graph itself is labeled with a letter, in this case $G = (V, E)$. The two vertices of an edge are called the end vertices. An edge, which has the same vertex as both of its end vertices, is a loop. Edges with the same end vertices are parallel. Edges that share one end vertex are adjacent. The amount of adjacent edges to a vertex is the degree of the vertex. A graph that has neither loops nor parallel edges is simple. [7, pp. 1-3] [13, pp. 2-5] Similar labeling and terminology will be used throughout this thesis.

In the graph in figure 3.2, we have both parallel edges and a loop, therefore the graph is not simple. Edges e_2 and e_3 both have the same end vertices and are hence parallel. Edge e_7 has the same vertex as its both end vertices, which means the edge is a loop.

Subgraph is a graph that is part of another graph. In other words, all its vertices and edges are in the other graph. More formally, $g \subseteq G$ if $E(g) \subseteq E(G)$ and $V(g) \subseteq V(G)$ [13, p. 4]. Two example subgraph diagrams of the graph in figure 3.2 are presented in figure 3.3. It should be noted that the graph representation in the graph diagram, meaning the placement of the vertices and edges in the drawing, has no meaning. Two graphs are equal if they both contain the same edges and vertices [7, p. 2].

A walk is a finite sequence of alternating vertices and edges. Each walk begins and ends with a vertex, and each edge is incident with both the preceding and the following edge. Starting from the initial vertex and concluding to the terminal vertex, the length of the walk equals the number of edges in the walk. Vertices v_{it-1} and v_{ik} are the end vertices of $e_{jt}(t = 1, ..., k)$. A walk with the length of 0 only

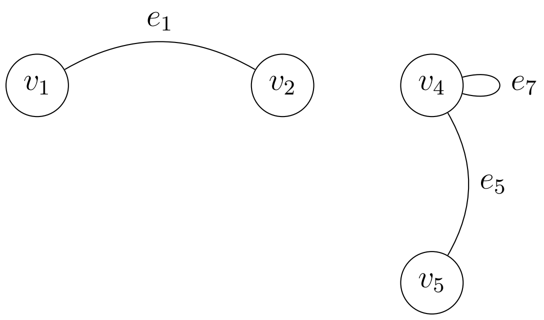


Figure 3.3 *Example subgraph diagrams of graph in figure 3.2*

consists of a single vertex. A walk is open if $v_{i0} \neq v_{ik}$ and closed if $v_{i0} = v_{ik}$. [7, p. 12] For example a walk in the graph in figure 3.2:

v1,e1,v2,e3,v3,e2,v2,e4,v4

has the length of 4 and is open since the end vertices are not the same ($v_1 \neq v_4$).

Another walk:

v2,e4,v4,e5,v5,e6,v2

has the length of 3 and is closed since both the end vertices are the same ($v_2 = v_2$).

A trail is a walk, which contains each edge at most once. Both examples above are trails, since no edge appears more than once in either walks. A trail is a path if it does not contain any vertex more than once, not including the end vertices that can be the same. [7, p. 12] The second example is a path since no vertex except the end vertices appears more than once. The first example on the other hand contains v_2 twice and is not a path. Path is a cycle if both the start and the end vertices are the same [7, p. 7]. The path in the second example is a cycle. Trails and paths in a graph G are also subgraphs of the graph G.

A graph is connected if any of the vertices can be reached from any of the other vertices by traversing the edges [7, p. 13]. In other words, a path exists from every vertex to every other vertex. The graph in figure 3.2 is connected.

3.2 Directed graphs

The graphs presented earlier, in which all the edges are bidirectional, are undirected graphs. Directed graphs, also called digraphs, are graphs in which the edges are directed from one vertex to another. One vertex is a start, and the other is a finish. These edges are also called arcs. [7, p. 171] Directed graphs can be drawn as undirected graphs but with arrows instead of lines representing edges. An example directed graph is seen in 3.4.

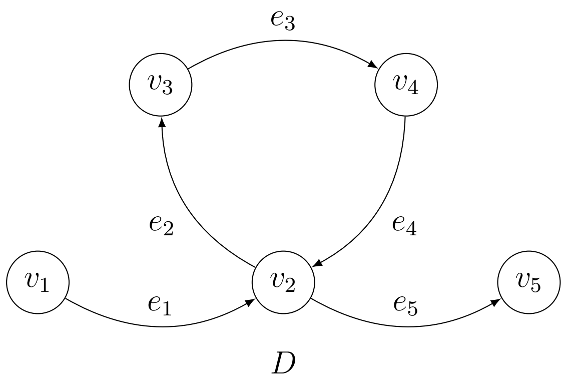


Figure 3.4 *Example directed graph diagram*

Directed graphs also allow some additional definitions. A vertex that only has arcs leading away from the vertex (indegree is zero) is called a source. Likewise, a vertex that only has arcs leading into the vertex (outdegree is zero) is called a sink. Walks, trails and paths work similarly with directed graphs as they do with undirected graphs, expect that they take into account the direction of the graph. [13, p. 125]

One useful special case of graphs is a DAG (Directed Acyclic Graph). This graph is directed and does not allow any cycles to occur. DAGs are useful in situations where certain tasks have a strict priority relationship - a task cannot be done before another is finished. All the precedence requirements are automatically described in the graph by following the directions. The reason why some applications benefit from DAGs that do not allow cycles is because a cycle would indicate a task that is a prerequisite for itself and the prerequisites would be infinite. An example DAG can be seen in 3.5.

3.3 Data flow

For applications that perform processing to data, the flow of the data can be represented using directed graphs [5]. At simplest, the graph consists of a source, vertex, sink and arcs combining the elements together. Source represents the interface to which the data is fed into, vertex represents the computation performed to the data, and the sink represents the interface where the processed data is finally outputted. Data flow graphs can naturally be much more complex, consisting of multiple sources, vertices and sinks. The example directed graph D in figure 3.5 could also represent a data flow, v_1 being the source and v_4 the sink.

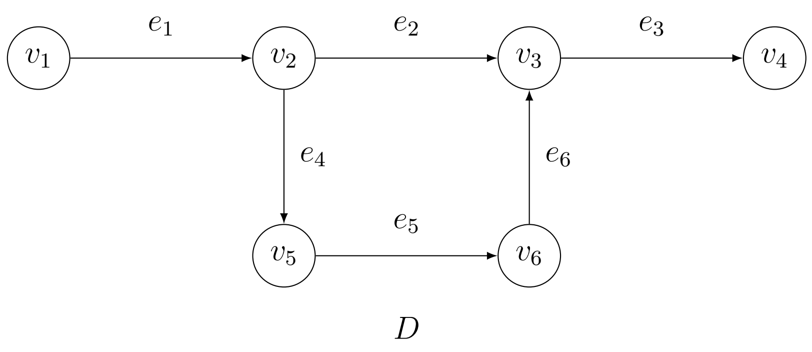


Figure 3.5 *Example directed acyclic graph diagram*

In order to ease the usage of the data flow graphs in any real world applications, several restrictions can be defined. Naturally, the vertices can only be dependent on the data of the inbound arcs. However, it is typically allowed that data staying constant over all the graph cycles can be provided by other means. This allows for example configuring complex hardware to operate in a certain way. In addition to the data resource restrictions, also the when and how the vertices are fired is typically a part of the data flow graph models.

A widely known data flow graph model is SDF (Synchronous Data Flow) [37]. Each vertex in the SDF graph needs to be a specific function that can be invoked after the input data for that vertex is available. There cannot be any side effects for any vertices. The only interactions between separate vertices can happen through the arcs connecting them. The amount of items consumed and produced by each vertex is known statically. Many applications and APIs especially in the imaging context are defined after these basic foundations.

The independence of the vertices allows the graphs to be modular. Vertices can be reused in different configurations and their ordering in the graph does not affect the internal computation of a single vertex. In other words, the computations are deterministic. Also, the granularity of the graph can vary significantly. A vertex can be implemented by a subgraph that performs the computation in an arbitrary number of vertices or all the subgraph vertices can be included in the SDF graph.

Since the vertices are black boxes to the outside observer, any form of operation may occur behind the interfaces of the vertex. The implementation of a certain vertex can be done on many different platforms like CPU, DSP or fixed hardware blocks. Two graphs can be functionally equivalent even if different implementations are used. What matters is that for a given input to a graph, a known output can be received.

SDF allows efficient scheduling of the different vertices since it is known exactly when the vertices are ready to be executed. SDF graphs also provide the ability to execute separate vertices concurrently. Since each vertex is independent, given that the inputs for all the vertices going to be fired are ready and hardware resources are available, they can be executed in parallel. [37]

3.4 Software frameworks

In order to take the aforementioned data flow graphs to a practical level in imaging, a software based framework capable of presenting operations as a graph to programmers is valuable. The frameworks can also allow running consecutive elements in parallel which is called pipelining [23]. Depending on the resource availability, pipelining can be used achieve notable performance increases.

There exists many different frameworks publicly available. All frameworks have their own capabilities and limitations, and a framework best suiting the needs of the product should be chosen. A short overview of widely used pipeline frameworks available is given below.

OpenMAX

OpenMAX is a set of open cross-platform APIs defined by the Khronos Group [31]. The main purpose of these APIs is to provide a set of standardized layers enabling the development and integration of multimedia components across different operating systems and platforms. The standard is split into three parts, OpenMAX AL, OpenMAX IL and OpenMAX DL. Moving from the bottom of the stack upwards, the OpenMAX DL is the development layer which defines a set of low-level multimedia kernels that can be used to accelerate traditional computational hotspots in media codecs [32]. OpenMAX IL is the integration layer which defines the standardized API for different audio, video and imaging components providing encapsulation and portability [33]. OpenMAX AL is the application layer that defines a set of multimedia use cases to be used by applications, like video playback or image capture, that can be constructed by connecting multiple IL components together [34].

OpenVX

OpenVX is an open standard for computer vision application acceleration defined by the Khronos Group. The idea is to allow cross-OS (Operating System) and cross-platform portability with minimum effects on the applications. OpenVX is graph based, and the APIs are defined in C. The standard does not define the implementation of the framework, but the APIs for building, verifying and coordinating graph execution, and for accessing memory objects. [35]

Gstreamer

Gstreamer is a cross-OS multimedia framework written in C. The framework is distributed under GNU Lesser General Public License, and is therefore open source and copylefted. The supported platforms include Linux, Windows and OSX among others. Gstreamer includes many plugins of its own, providing an extensive library of elements, and supports extending its capabilities through plugins. Gstreamer also has API bindings to many other programming languages than C which it natively supports. [21]

DirectShow

DirectShow is a multimedia framework for Windows developed by Microsoft. DirectShow consists of filters, which are the software components performing some operations for the multimedia stream. Applications then perform tasks by creating filter graphs which are constructed by connecting different filters together. DirectShow can be used for video and audio playback and capture. [43]

V4L2

V4L2 (Video4Linux2) is a Linux kernel video capture and output driver framework. It is written in C and is licensed as GNU General Public License. Different hardware blocks are exposed as V4L2 devices, which can be configured and connected as requested. Using media controller that is a part of V4L2, these devices can then be used and connected together to form a graph. V4L2 can be used in a variety of applications like image capture, radio controlling or digital video broadcast receiving. [41]

3.5 Graph implementation

Each graph is defined to a certain task. Data flows from the source through each vertex and finally reaches the sink. With a sane input a valid output is received. The vertices in the graph define what is to be done to the data. However, the graph does not dictate how each vertex is to be implemented. The graph only governs the order and the dependencies of the vertices.

The implementations for each vertex can be done on any supported compute resource. With today's SoCs, devices have become heterogeneous and implementations can be for example pure hardware, firmware running on specialized processors, or just software executed on a CPU [53]. Based on the product requirements, appropriate implementations are designed. A figure showing the tradeoff between efficiency and flexibility of some of the different technologies can be seen in 3.6.

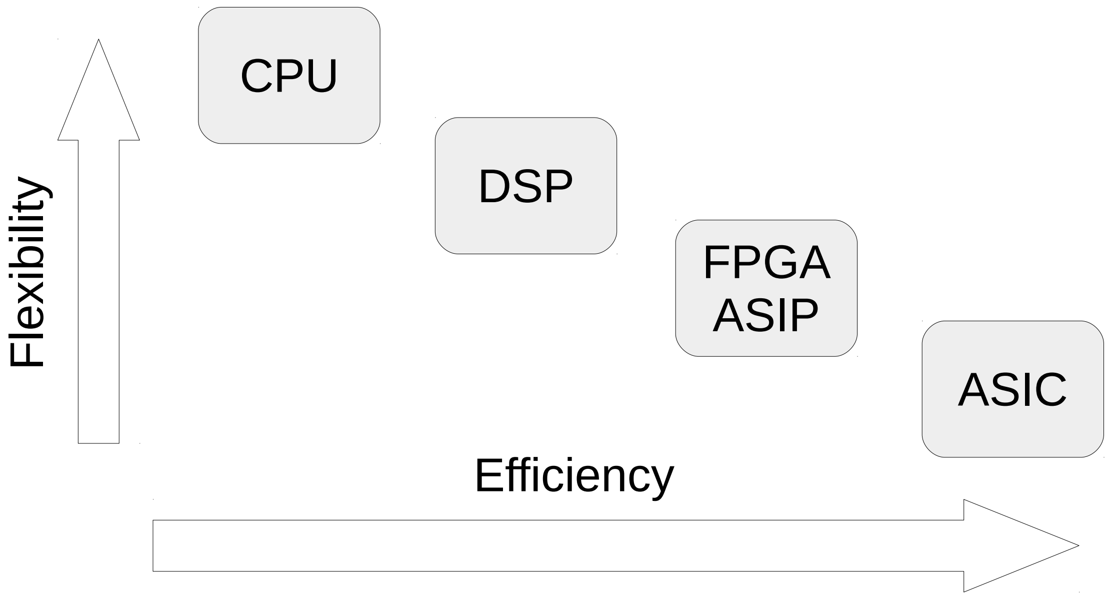


Figure 3.6 *Flexibility in relation to efficiency in different technologies*

Implementations can also combine multiple vertices into a single vertex if required. This can be useful for example to allow performance enhancements via reducing the memory bandwidth between vertices by performing all steps of an algorithm on the same target without the need for host control. While the implementation can be highly optimized for a given platform, this makes the granularity of the graph coarser.

Encapsulating the vertices behind a standardized API allows the development of separate vertices in isolation. Vertex implementations could potentially be done by separate teams or even companies. Ready implementations can then be connected together using a common framework as the interoperability of the vertices is guaranteed by the specification. Encapsulation also provides benefits related to reuse. As the requirements for each product differ from one another, being able to flexibly create new graphs reusing the already existing implementations is a major advantage.

With software development starting very early in the process of developing the device when the actual hardware implementation is not ready, the modularity provided by the graphs is an efficient way to enable development of the software earlier than traditionally. Traditionally software development has only begun after a prototype hardware is available [53]. An example traditional schedule can be seen in figure 3.7. However, with the complexity of today's systems, this no longer fits the needs of the time to market requirement of the products. A shift left approach is needed, in which software development is performed concurrently with the hardware development. The shift left software schedule can be seen infigure 3.8.

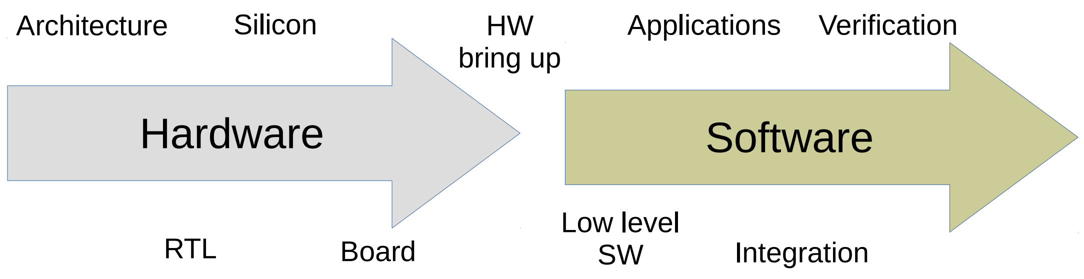


Figure 3.7 *Traditional hardware and software development schedule*

Graphs provide a systematic way to utilize shift left. As the implementation of the vertices in a graph is meaningless from the point of view of the user, work with several of the technologies used in the development phase of the device can be enabled. At the beginning of the product development, only rudimentary implementations of different processing blocks may be available. Some of the implementations may be missing and may need to be replaced for example with generic CPU software that is lacking the image quality and processing speed present in the real implementations.

Gradually new implementations will be available. New development platforms will be enabled. Development moves from software stubs to emulation and finally to real prototypes. The hardware will start resembling more and more what it will be in the final product.

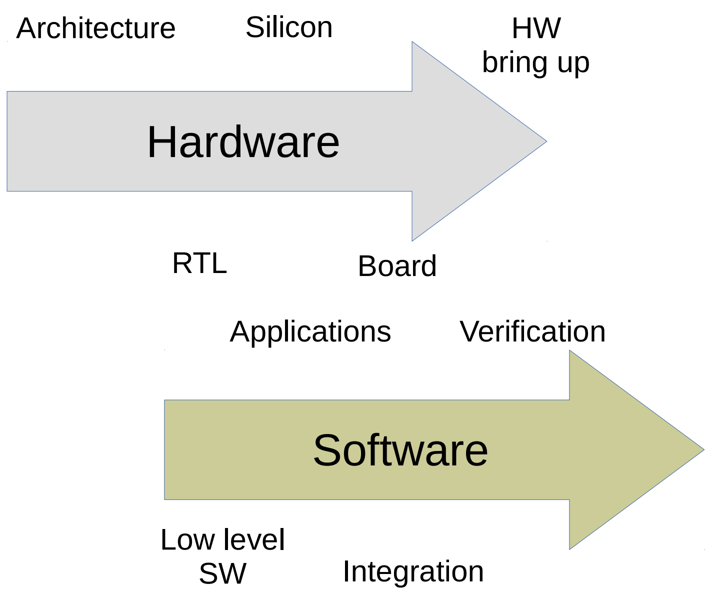


Figure 3.8 *Shift left software development schedule*

A list of commonly used development platforms is given below.

Software stub can be used to begin development of software without any real software or hardware implementations below. Stubs can implement the required low level APIs by just providing some initial responses regardless of the input.

Simulation is simulating the behavior of a system. Simulation does not perfectly replicate the actual target being simulated but provides behavior similar to that.

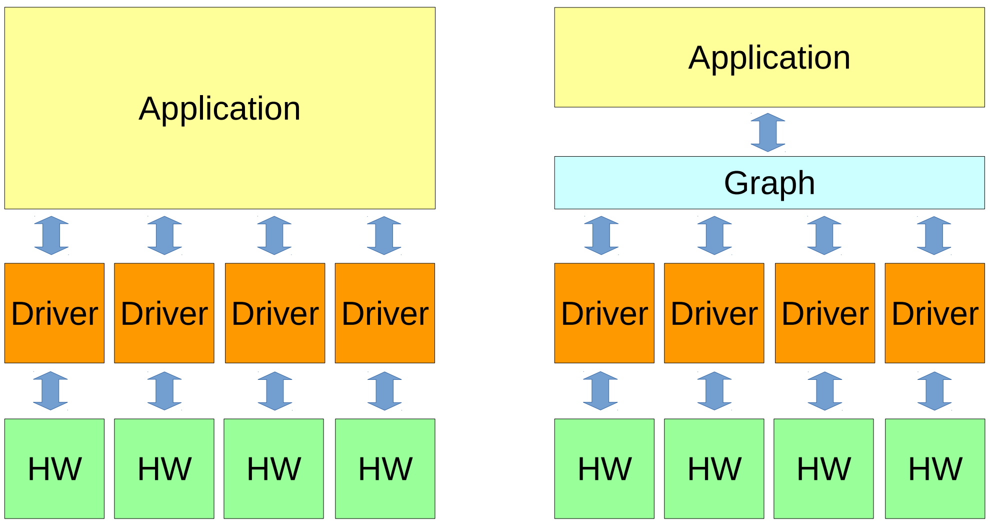
Software emulation is emulating the behavior of a system using software. This can for example emulate the behaviour of lower level APIs to allow the development and verification of higher level software layers.

Hardware emulation is emulating the behavior of a system using hardware. For example, an FPGA can be used to emulate the system accurately with impacts only to the performance.

Hardware prototype is a prototype of the actual device. Prototypes are the development platforms closest to the actual finished products. They contain the real hardware that is going to be in the final product although some changes can still be done to later revisions.

These development platforms can be used to enable software shift left and verification beginning from the early stages of the product development until the final product by using implementations for platforms available at each time. Traditionally handling the complexity of different implementations and compute resources has been hard. Individual blocks may have been tested independently, but it has not been consistent across all algorithms. This means less reuse and higher risk in further development phases.

Utilizing different resources require changes in the whole software stack which potentially has an effect all the way up to the application layer. This makes development and maintaining the software stack cumbersome. With graphs, however, the complexity of the underlying computing can be abstracted and managed in a simple way. Example software stacks without and with graphs can be seen in figure 3.9.



(a) Traditional software stack **(b)** Software stack using graphs

Figure 3.9 *Example software stacks without and with graphs*

Graphs can expose any external requirements needed, for example to allocate the required memory buffers, to the layers above in a generic way independent of the vertex implementations. This allows us to encapsulate the whole functional complexity behind generic interfaces. In other words, the software stack above the graph does not need to know about the implementation details of the graph, and any changes to the implementations can be hidden from the applications.

3.6 Graphs in an image processing system

The camera capabilities of a product can range from simple fixed surveillance cameras to highly complex devices providing vast amounts of features and different operating modes. As the devices have become more capable and rich in features, handling the complexity of subsystems (e.g. camera) has become increasingly difficult. For example in mobile devices, camera requires seamless cooperation between traditionally separate domains like display, graphics, storage and a variety of other sensors. As described in the previous section, graphs can be used to handle and hide the complexity from the above software layers.

Camera pipelines introduced in chapter 2 can be modeled as data flow graphs. Image comes in from a source, goes through the required processing vertices, and is finally outputted through a sink. The example pipeline shown in figure 2.1 is represented as a data flow graph in figure 3.10.

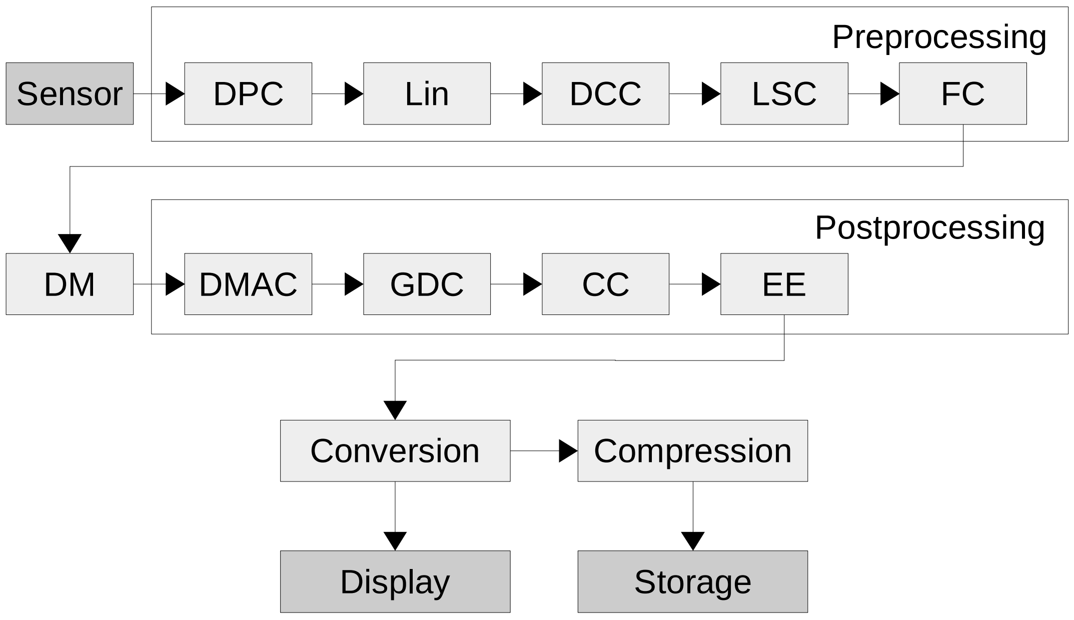


Figure 3.10 *Image processing data flow graph*

Capturing and processing images can be divided into multiple different use cases. Differences between use cases may be subtle or major, but nevertheless require implementations that differ from each other. Graphs provide a means to handle these different configurations.

Some of the differences may be supported by just reconfiguring the graphs while others require completely new graphs and vertices. For example, in order to obtain better performance, some enhancements may be turned off by removing the related vertices from the graph. Vertices may also support different configuration options, for example to change the resolution or the intensity of the correction algorithms.

The following are examples of requirements in the graphs that may change from one use case to another.

Frame rate is the amount of frames that pass through the processing system in a given amount of time. Usually measured as FPS (Frames Per Second) that is the average number of frames processed in a second.

Latency is the amount of time that it takes for a frame to travel from a source to the sink. Can vary from one frame to another even if the average frame rate over a longer time period is relatively constant.

Resolution is the amount of pixels in the images. Higher resolutions produce images with more details and allow more aggressive cropping without resulting in a pixelated image. On the downside, higher resolutions result in more requirements for processing power and memory bandwidth.

Image quality is the quality of the outputted processed image. Depending on the use case, different aspects of the image quality may be required or sacrificed in favor for other requirements.

Number of sinks can vary from one use case to another. In addition to the main image that is outputted, other versions of that image may be required. When the use of the sink is known, a less or more demanding compute path may be selected. The result may also be a side product of an intermediate algorithm in the main path.

Additional processing may also be introduced in special imaging cases to analyze the image in ways that the standard use cases do not. For example, computer vision may perform a wide variety of algorithms to produce specific metadata from the images.

The three basic use cases in a mobile camera are examples that have different requirements from one another. When the user is preparing to take a picture, a preview of the image to be captured is shown on the screen of the device (viewfinder) before the user presses the button to do the actual capture. Since each frame is only shown on the screen for a fraction of a second, the image quality in the preview is not that important, and the resolution only needs to match the display of the device. The main thing that matters is that the user gets real time information about what is going to be included in the captured image. Therefore, low latency and adequate frame rate is required.

With the actual still image capture, however, image quality is one of the most important factors. The captured image will be stored in the device memory in addition to showing it on the screen, and the image quality should be as good as possible. Resolution is preferably the highest available so that the images will look good on a larger screen also. Frame rate is not that important as long as an image can be acquired within a reasonable time frame. Traditionally, shutter lag is more important than the processing latency.

While the preview and capture represent the two basic use cases in a mobile camera that have different requirements, video can be considered being somewhere in the middle. In video recording, the frame rate is a constant value that needs to be acquired. Latency needs to be kept within boundaries or frames will be dropped. Image quality is made as good as is possible with the performance requirements.

Apart from the three basic use cases in mobile cameras, the potential amount of different configurations even in a mobile device is huge. Camera can be used in a wide variety of different ways. Some users may be satisfied with a recognizable face in video calls while others try to replace a professional digital camera with it. On the other hand, what humans view as a high quality image may not be true for certain algorithms trying to interpret information from it. Augmented reality provides additional overlay to the environment captured by the camera requiring recognition of the surrounding objects from the image. User recognition during login or other secure access using face detection or iris scanner needs to obtain an image from which the algorithm can recognize the user.

While there are certainly more than enough use cases in mobile cameras to make them complex, the matter is further complicated once the same or similar platforms are used in different products in the imaging context. Surveillance cameras for one may include additional intelligence to track persons or vehicles. Robots require knowledge about their environment and may need specialized cameras to obtain it. Night vision devices concentrate on providing a viewable image from an otherwise dark environment while the traditional quality of the image does not matter.

All these use cases combined with the configurability of image resolutions and other parameters provide the reason for using graphs. Vertices can be reused, replaced or reorganized. Implementations can be done on any compute resource available.

Essentially, the complexity of the underlying processing system can be handled in a single graph and be hidden from the above software layers.

4. VERIFICATION OF IMAGE PROCESSING GRAPHS

Each image processing graph needs to be verified in order to ensure that the graphs are working according to the specification. Verification is not a simple task, however. What to verify from the graphs and how the data interfaces can be measured in a way that supports the verification? Which methods should be used to ensure that the image quality and performance meet the requirements?

Verification in general can be split into two different categories: static and dynamic verification. Static verification (also called formal verification) means statically verifying the implementations using mathematical formulations [18]. In static verification, either two implementations are compared with each other or the implementation is otherwise showed to satisfy the specification. Static verification can theoretically show that the implementation satisfies the specification under all inputs. However, in practice it can be infeasible to convert the implementations into the required mathematical form.

Dynamic verification (also called testing) on the other hand means verifying the implementations while the system is actually running using a finite set of test cases [18, 50]. Input is provided to the implementations and the outcome is examined. As dynamic verification only uses the input and output interfaces of the implementations, there is no need to know the internal details of the implementations, which in turn allows black box testing. Dynamic verification can only be used to verify an implementation to a certain point, as in practice a complete set of tests can be considered infinite [50]. The coverage of the test cases is limited by resources available and the test cases need to be created based on risk and prioritization.

There are several aspects to the graph verification. The functionality of the graph needs to match that of the graph specification. Performance of the graph has to be greater than or equal of what is required by the specification. The graph needs to also be stable, or at least stable enough to exceed the required criteria. Since the implementations behind the vertices are unrestricted, static verification that requires details of the implementations is infeasible from a graph point of view. Therefore, dynamic verification suits the needs of graph verification more adequately. Naturally, this does not restrict the use of static verification separately in the development of each implementation if deemed suitable and useful.

Fundamentally verification of image processing graphs is creating and using test cases that verify the properties of the graph against the specification. As the development of the project goes forward, new features will be gradually available and integrated. Tests are implemented along the integration of the new features or even before this utilizing the test driven approach [50]. After a test is passing, it can always be executed with the other tests whenever tests are run, potentially revealing regressions in the future.

In addition for the tests being used to catch any regressions, they should be considered as a tool for the developer. When modifying features or adding new ones, tests can be executed and used to drive the inputs of the system under development. This substantially reduces the amount of dependencies when compared to for example running the whole camera application. Changes to a small subsystem can be more efficiently developed with a more concentrated test.

4.1 Utilizing the modularity of graphs

Traditionally, especially in the imaging context where implementations often use hardware to perform the required algorithms, a use case would result in a single functional unit that can be executed from software as required. Exposing only a single big black box to the software causes testing having to be performed on top of the whole use case. Any finer testing needs to be done at the level of firmware or even hardware. There is a long gap between testing the blocks at such a low level, compared to being able to test them at the actual software level where they will eventually be used. Especially in the early stages of development where features are not yet ready, this is inconvenient and risky. An example of testing complete use cases with a traditional software stack utilizing hardware can be seen in figure 4.1.

Instead of exposing a use case as a whole to the software, smaller portions of the system needs to be exposed. In software development, unit testing is a common practice that has been used to isolate parts of the code and to show they are working correctly [11]. Naturally, unit testing requires small portions of the system to be available for such isolation. In order to perform unit testing for implementations that utilize different processing elements, for example dedicated hardware, also the hardware needs to be exposed in a way that allows the isolation to happen. This

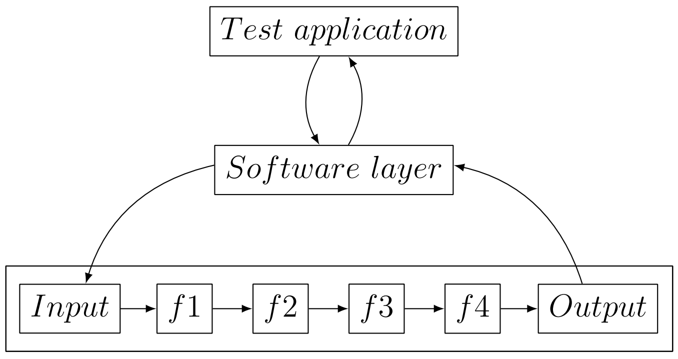


Figure 4.1 *Traditional testing of a processing system*

can easily result in complicated interfaces that lack consistency from each other and are hard to test.

With graphs, the modularity required for isolated testing of small parts of the system already exists. Each vertex is exposed to the software level. In addition to being able to test the graph as a whole, subgraphs or even single vertices can be tested. As was described in section 3.3, the only interactions between vertices can happen through arcs connecting them and no side effects are allowed. This means that the vertices are always guaranteed to have a compatible interface and they can be isolated no matter how complicated the actual implementation is. An example of testing the system using graphs and its subgraphs can be seen in figure 4.2. In this graph, vertices v_1 and v_2 are tested in isolation, while the vertices v_3 and v_4 are tested as a subgraph consisting of both vertices.

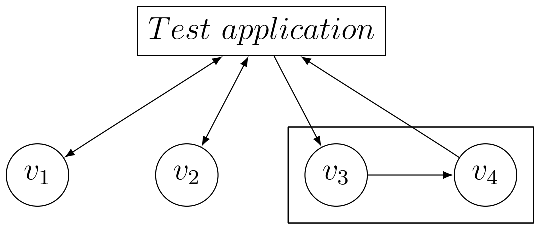


Figure 4.2 *Example testing of subgraphs in isolation*

Vertex or subgraph testing in isolation is especially useful in the early development stages when new features are being integrated and need to be properly tested, but while the complete graph is not fully implemented or integrated yet. Vertices can be tested properly using the exact same interfaces and software layer as will be in the final graph. This eases the final integration of the whole graph and lowers the risk of unpredictable issues later.

Another benefit of the modularity is the ability to use test related vertices in the graphs in addition to testing subgraphs in isolation. For example, if some of the vertices are not yet implemented, unimplemented vertices can be replaced by a reference implementation, or a mock-implementation that does not perform any actual computation but serves as a placeholder and an API for the software layers controlling the execution of the graph. If the hardware does not yet support the features requested or the hardware does not yet exist, simulation or emulation can be used in some vertices. Also, if some implementations are not completely ready, additional vertices can be added to create subgraphs that fulfill the missing computations of a vertex using software. An example configuration utilizing these can be seen in figure 4.3.

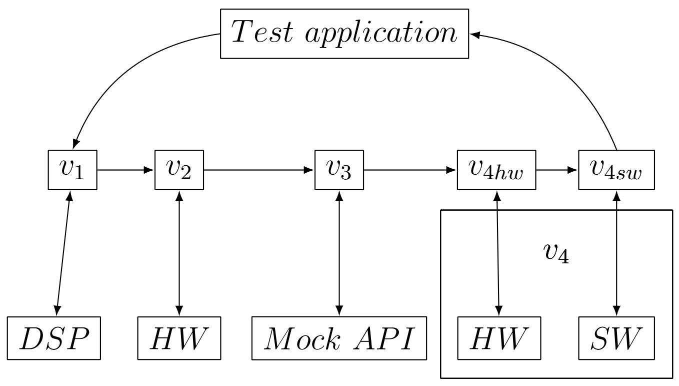


Figure 4.3 *Usage of different development platforms in testing*

During the development and integration of new features it is often convenient to be able to efficiently test the graphs with different inputs and configurations. Because of the modularity of the graphs, there are virtually no limits on what kind of infrastructure can be added to the graph. Vertices can be added anywhere in the graph and the outputs of the vertices can be flexibly connected into the surrounding infrastructure.

4.2 Graph verification

Verifying the graphs provide two types of approaches. The graph can be either verified as a complete or split the into subgraphs or even single vertices, as described in the previous section, and verify those independently. The best manner of approach depends on the granularity of the graph but both approaches also supplement each other.

Because the interfaces of the graphs and the vertices are compatible with each other and specified in beforehand, software created for verifying the graphs can be easily reused. Even complex implementations can be tested with little changes to the test code. Once a basic test bench exists for the graphs, new tests can be easily created.

4.2.1 Functionality

Functional verification of a graph refers to verifying that the graph processes the data according to the specification. For a known input, the output can be examined using various methods. The data can be evaluated either objectively or subjectively.

Since the algorithms behind the vertices are known, a golden model can be used to produce reference input and output data. With this input data, the output data should match the reference no matter what the implementation behind the graph is. This is called bit exactness and can be used to verify the graph implementation. An example bit exactness test is shown in figure 4.4.

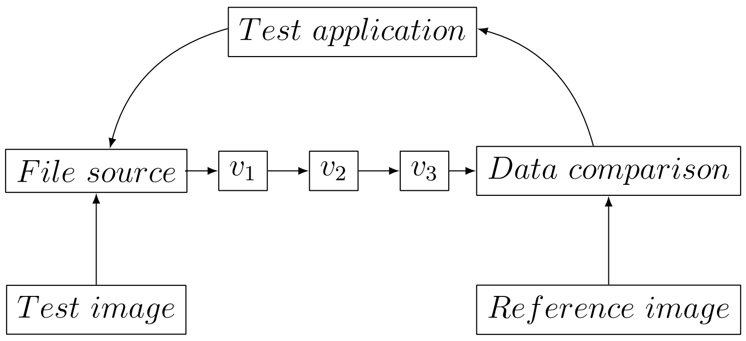


Figure 4.4 *Testing the graph bit exactness against reference data*

In this test additional test vertices are included to the graph to support loading the input data and comparing the output data against the reference. These vertices replace the original source and sink in the graph.

Modularity of the graphs can be utilized for bit exactness testing also. This is especially useful in pinpointing the location of the problems in case the output from the graph does not match the reference data. The graph failing the test can be divided into smaller blocks, each of which is tested independently in isolation, revealing the broken vertex by failing the isolated test. The finer the graph granularity is, the more detailed is the test outcome.

Verifying the bit exactness of a graph against a golden model expects that the image quality of the algorithms is already validated. Image quality can also be measured using the graphs and the metrics introduced in section 2.1.

In subjective assessment, functional validation means that the output image is visually inspected to identify whether the graph is executing according to the requirements. This requires that the output image is converted into a representation that is viewable by the user. Depending on the output format of the graph and the stage of the processing, extra reference implementation vertices like color converters can be added to the graph to make this convenient. A sink can then be used to either save the converted image to file to be viewed by an external application or to directly display it on the screen.

With a human being seeing the output image, an analysis of the image quality can be done. As described in section 2.1, subjective assessment is the most reliable way to evaluate the image quality. It is also easy to implement and requires only a small test infrastructure on top of the graph.

Subjective assessment can also be used to inspect the effects of subgraphs or single vertices to the image quality. This can be especially useful when the algorithms are being still developed or their parameters are configured so that the best outcome can be obtained. In the best case, the developer can isolate a single algorithm and use a highly algorithm specific input data to see the effects of that specific algorithm.

As was described in section 2.1, the human visual system has its characteristics on how it sees the effects of distortions in the image. Objective assessment can be used to support the subjective evaluation to measure the image quality. Verifying the output image against a reference using objective evaluation results in metrics that can be easily interpreted. This can be especially useful if changes in the algorithms result in small changes in the image quality that can be more easily measured objectively than by the naked eye. From these metrics it can be determined if the image quality is within the requirements. While not as reliable as subjective evaluation, these metrics can be processed automatically without any human intervening. Objective assessment allows for faster verification with more coverage.

Being able to perform the objective assessment algorithms, some additional vertices need to be added in a similar way as in figure 4.4. Instead of a bit exactness comparison, the algorithm comparing the output and the reference data is run. The resulting metric is then outputted and compared against the minimum requirements.

Objective assessment can also be utilized with subgraphs. In a similar way as the subjective assessment can be used in development, objective evaluation provides another way to determine the effects of a certain algorithm to the image. For example, in case of a lossy compression vertex, calculating the difference between the output and the input of that vertex provides good information about the quality of the compression.

4.2.2 Performance

Performance is another important criteria of verifying the graphs. Depending on the use case, the requirements may vary significantly from one graph to another. In image processing, there are several different aspects to performance. The frame rate at which the processing is executed needs to be high enough to support the use case, but also the latency of the graph needs to be inside the specification.

Frame rate, as described in section 3.6, is the average number of frames in a given amount of time. Frame rate can be verified by running the graphs for a long enough period and calculating the average FPS from the amount of time the test lasted and the number of frames that were processed. This measurement gives a general overview of the processing performance of the pipeline.

However, average frame rate does not tell the whole truth about the performance. In addition to being able to process frames fast enough in average, the processing time of each individual frame is of importance. There may be variation in the time of separate frames for example because of limited hardware resources and their shared usage blocking the execution. Too long processing time for a single frame and the frame may need to be skipped or lag may be otherwise visible in the output. Some correction algorithms require the images to be captured within strict time limits or otherwise the corrections may fail.

The variation in the times, also called jitter, can be verified by tracking the arrival of the frames at key points in the graph. Jitter in the image capture can be calculated based on the timestamps from the source, while deviations in the rest of the pipeline can be obtained from the difference between the source timestamp and the sink timestamp. Jitter for separate vertices can be calculated in a similar fashion. All these values can then be compared against the requirements and used to help in pinpointing the bottlenecks in automated testing.

Also the latency of the pipeline can be significant. For example, in case of heavy internal parallelization in the pipelines, the frame rate of the pipeline and the variation in the processing times can be within the specification, but the delay for one frame to reach the output from the input may be too high. In some use cases it is important to receive the processed image as soon as possible, perhaps even more important than the ability to provide a high average frame rate. The average latency can be calculated from the timestamps already introduced with jitter.

In graph performance measurements the performance of the complete graph is what matters in the end as that is the true performance seen by the user. However, when investigating performance issues, it is extremely useful to know as much details as possible about what exactly is causing the low performance numbers. For this, the ability of being able to test the vertices separately is of great help. The performance tests can be used as a tool for high level profiling. Depending on the graph granularity, areas that need optimizing can be identified.

4.2.3 Stability

While having the graph function as is defined by the specification and being able to provide the performance required by the use case are fundamental aspects of verifying pipelines, they are of no use if the stability of the graphs is poor. Instability may manifest itself in several ways. Output from the graphs may be sometimes corrupted, some of the processing elements may fail to execute successfully, or in case of privileged access, the whole system may crash.

One way to catch instability problems is to simply run the graph for long periods and observe for any anomalies in the output or errors in the execution of the graph. Bit exactness comparison can be performed for each frame against a reference image. Since stability issues usually occur quite randomly, this method will not catch all stability related issues, but is rather a crude way to spot the most major issues.

A common cause for instability is related to concurrency. While graphs with SDF already improve the situation for writing thread safe software, concurrency is still a typical source of instability. In those graphs that support concurrent execution, it must be made sure that there are no issues related to the parallelism. For example, separate processing elements may depend on the same hardware resources, which if not handled properly may cause stability issues. Concurrency can be tested in a similar way as the other stability issues by utilizing multiple concurrent graphs at the same time or by modeling concurrent vertices in the graph. As was the case with the non-parallel stability testing, issues may not reproduce all the time since the concurrent issues are timing dependent.

While stability issues are not easy to catch, they are still an important part of the graph verification. A common metric called MTBF (Mean Time Between Failures) describes the predicted time between failures for a system. Based on the required MTBF, appropriate stability testing is needed to ensure the value can be reached.

4.3 Automating the test execution

A part of efficiently utilizing the power of modularity of the graphs and the implemented tests is to frequently execute the tests, preferably after each change or release impacting any component in the domain. The burden of running the tests should be as low as possible, while still being able to achieve substantial coverage of the graphs and the specified use cases. Ideally there is one test set that can be automatically executed and which reports to the user about any occurred failures.

Some of the tests introduced in the previous sections can be made automatic straightforwardly while others are more complicated. Automation of subjective testing is infeasible, so only objective testing can be used. Bit exactness test automation is straightforward. For each test case there needs to be a set of input and output data against which the graph output is verified. Objective image quality assessment can be automated in the same way as the bit-exactness testing. Performance test results can be compared against the specified requirements. For each graph there can be limits for frame rate, latency and jitter.

Stability testing is more complicated to automate for several reasons. First of all, the methods described in section 4.2.3 require long periods of execution which takes a substantial amount of time. Secondly, problems with stability may cause issues that render the platform unusable, resulting in a total crash or an unknown state of the device. Many different methods like hardware watchdog timers for recovering from total system lockups can be used to provide valuable information over long periods of testing. If stability related problems are noticed, then testing can be manually continued to further investigate the problem based on the information received from the automated test. The most important thing of automated stability testing is to be aware of any problems so that they can be acted upon.

5. USAGE IN A PRODUCT DEVELOPMENT PROJECT

Image processing graphs were used from the beginning of the software development of a mobile camera product all the way until the completion of the project. Graph verification was started alongside the first graph implementations as a set of tests that were continuously developed and improved alongside the graphs. The graphs that were used were not full end-to-end graphs but only contained most of the image processing part, leaving the image capture and some amount of processing outside the graphs.

Graphs were used as a part of a complex camera software stack. Vertices were implemented using a processing unit consisting of fixed imaging function hardware and multiple VLIW (Very Long Instruction Word) processors. The vertices and utilities to assist graph verification were implemented for a CPU. A proprietary C-language based cross-OS graph framework was used to implement the required functionality to construct and execute graphs. The same graph implementations were used in multiple operating systems, but the main development and verification was done in an Android environment.

Graphs were used directly on top of the kernel driver interfacing with the hardware similarly as was shown in figure 3.9b. The graphs were isolated so that they can be constructed and executed without dependencies throughout the whole software stack, requiring only the input data and some resources that are requested in a generic way. This allowed graph verification to happen in isolation with only a light test framework around it, removing the need for supporting the rest of the complex stack above it for verification purposes.

Shift left was utilized in the development. Two separate pre-silicon development paths were used from the beginning of the project, one to enable the development of the software stack above the graphs, and the other to enable the verification of the graphs and the implementations behind them. Both paths utilized the properties introduced by the graphs. The following sections describe the details of the product development.

5.1 Test framework

For a software based graph framework, a software based test framework is a logical choice. Using an already existing framework to implement the tests instead of creating everything from scratch introduces a great advantage: all the required tools are already available. Implementing tests is only about writing the tests, thus not having to use resources to create the plumbing below. This saves both effort and time.

Since the framework used for the graphs was cross-OS capable and used in multiple operating systems, this was also a requirement for the tests. There should be no need to replicate similar tests between different platforms if the software interface below the tests does not change. Naturally, the test framework must also support the programming language of the software to be tested, which was in this project C.

The requirements for the test framework led to the choice of Google Test. Google Test [19] is an open-source cross-platform C++ test framework maintained by Google and distributed using a BSD-style (Berkeley Software Distribution) license making the use of the test framework quite unrestricted. Since C++ supports calling C code, the programming language used for Google Test is an excellent choice, allowing the test developers to use more modern tools included in C++ to support testing, and also to provide the possibility to test any other related code components possibly implemented in C++.

5.2 Test implementation

Tests were implemented during the product development starting from a few cases running in emulation and resulting in a comprehensive set of tests in the final product. For each new graph and vertex, tests were made when the features were integrated. After the integration, the tests were used for regression testing for every change affecting graphs or implementations below them that were introduced. A model for test driven vertex development was made available but was not fully put into use during the project.

5.2.1 Functionality

Two types of functional tests were implemented. The first type was a basic functional test that uses arbitrary data as a frame input for each available graph, constructs the graphs according to the graph description on the target device and tries to execute the graph for multiple cycles. The testing was done without any meaningful input data, but was rather to verify that the graph can be created according to the description and that the graph is actually executable on the target device.

The second type was a frame content test that operates on the actual frame data providing additional features to the verification. The inclusion of frame content data allowed the users to provide input frame data to the graphs under testing and to verify the output data by using subjective and objective assessment. Additional file source and file sink or display sink were added to the graphs under test to support the frame content verification. The verification of graphs using image quality metrics can be seen in figure 5.1. A reference algorithm is used to provide a reference output for implementations. Both the graph implementation and the reference algorithm is configured using same reference parameters. Also, the same reference input is used in both. Vertex outputs are then compared against the reference outputs using image quality metrics.

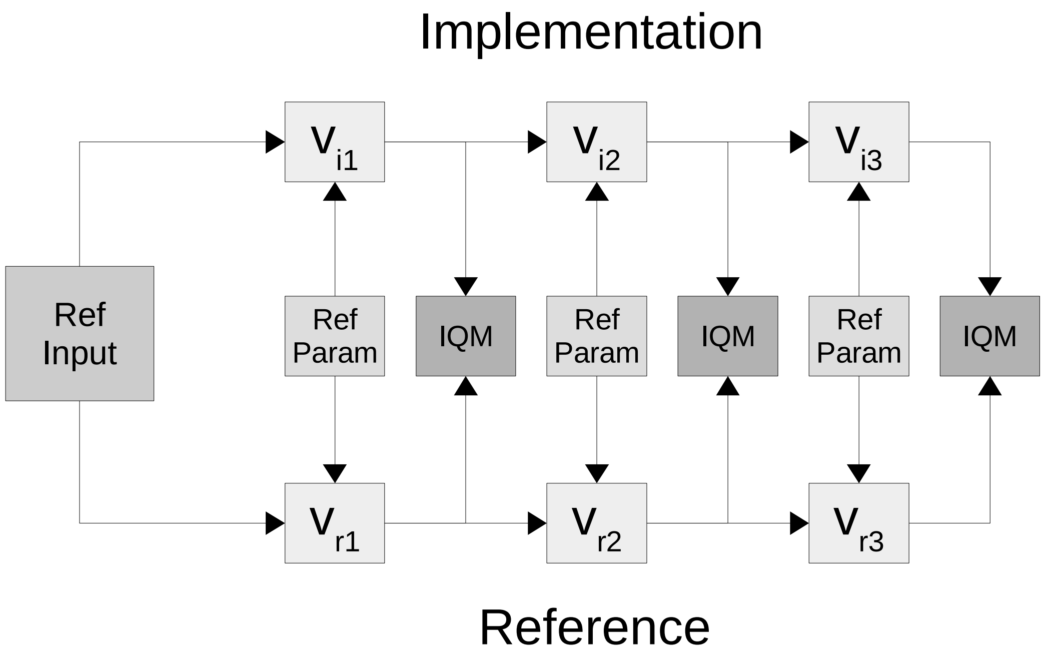


Figure 5.1 *Graph functional verification. Implemented graph vertices are verified using IQM (Image Quality Metric) against a reference.*

The functional tests were used to test the functionality of all the required graphs and use cases. With a list of input and output resolution configurations for a given graph that were needed to be supported, tests could be created for each of these configurations. With the test set it was possible to verify that all the required configurations could be executed and that the output image was meeting the requirements.

5.2.2 Performance

Performance tests were implemented for the graphs that were specified to have performance requirements. Tests were made so that for each graph under test, the average frame rate and the minimum and maximum execution times are reported. Tests were also implemented for each vertex in the graphs so that performance profiling could be done with a finer granularity, allowing more focused benchmarking and performance bottleneck pinpointing.

The performance test executes the graph under test for several times in a similar way as the basic functional tests do using arbitrary input frame data. The average frame rate is then calculated based on the amount of frames processed and the time the execution took in total. Latency is reported based on timestamps that are captured before and after the graph execution.

The purpose of the performance testing was to verify that the graphs reach the specified performance requirements. The processing speed capabilities of the graphs in relation to the clock frequency were verified to make sure they fulfill the requirements. Another purpose of the performance testing was to verify that the bandwidth of the graphs exceed the requirements.

5.2.3 Stability

Stability testing was enabled by allowing the amount of iterations in the basic functional tests to be increased. Graphs could be then executed for long periods in order to spot any instability. Support for detecting any corruption in the output was also added by comparing the output data against the output of the first iteration.

Concurrent testing was also introduced. Multiple instances of the same graphs supporting concurrency were created and then executed in separate threads for long periods of time. Tests using combinations of different graphs reflecting real use cases were implemented. The amount of threads for each test was controllable to vary the amount of parallelism.

5.3 Test environments

Several different test environments were used during the development of the product. Development began with low level software stubs providing only the APIs against which higher level components were initially developed. Gradually development was moved through several different development platforms eventually resulting as a layer in a full software stack in the real device.

Testing was a part of the development starting already in the very earliest phase. Functionality for the tests was added piece by piece as the overall complexity of the software increased and more features became available.

5.3.1 Software stub

The initial platform for the beginning of the software development was to use low level software stubs. These stubs were made in the kernel mode driver level to provide the low level API for the vertices used in the graphs. These stubs did not have any real functionality, but they were merely implemented to allow early development of the components above the kernel driver interface in the stack. Software stubs were used in a previous platform running Android and were executing solely on CPU. A figure of the software stubs and their relation to the graph framework and the application can be seen in figure 5.2.

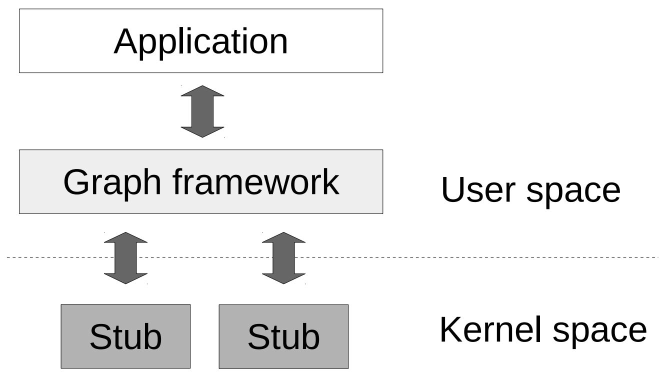


Figure 5.2 *Software stubs in early development*

Software stubs were of great help when the initial implementations of the graph framework and the firmware dependent vertices were implemented. Basic tests running graphs in isolation could be performed for the software, allowing the core of the flow to be implemented and tested before any real functionality was available. Running the tests with the stubs also served as a proof of concept for the graph verification. It was showed that it is possible to isolate the graphs and their vertices and to verify them without any additional software components unrelated to the testing on top of the graphs.

5.3.2 Software API emulation

An earlier product was used to support development before any actual hardware was available. Low level APIs were implemented using the old hardware to behave similarly as the APIs would in the real future hardware. This software had the functionality required to build coarse versions of the basic graphs. The emulation setup can be seen in figure 5.3.

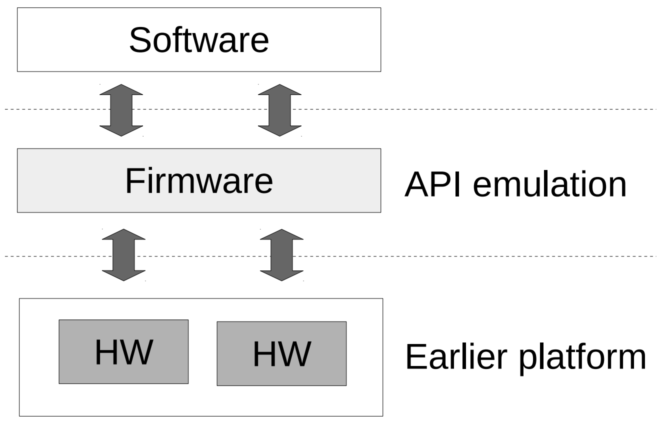


Figure 5.3 *Software API emulation using an earlier product. Firmware performs the emulation by mapping old hardware to mimic future hardware behavior.*

In this environment, the basic use cases could be tested using real image data. This provided the means to actually test the whole graph from end to end, and to develop many features that would have otherwise been postponed by the lack of hardware. Even though the actual implementations and hardware were not part of this testing, the data flows could be implemented and tested.

The functionality of this development platform also enabled the developing of the software layers above graphs long before than what could have been done in a traditional schedule. Since all the graphs and the vertices within the graphs are exposed via the same API, the implementation details do not considerably alter the way software higher in the stack uses the graphs. Software API emulation was the first platform in which the graphs were used all the way from the user application, proving that the graph approach works in practice.

Software API emulation was used as the primary platform to further develop the higher level software on top of the graphs. Even though the available graphs were very limited from the feature perspective, they provided a real-time platform with working use cases on top of which a substantial amount of the generic software implementation could be done. This allowed the verification of the actual hardware in the pre-silicon environments to be left out for the more focused graph verification.

A working environment with usable graphs also allowed the development of the verification tests. All types of tests were developed for this environment and utilized as a proof-of-concept and a reference for the future environments. Because of the generic handling of the graphs, the tests could then be utilized with little changes for the future real use cases.

5.3.3 Hybrid system emulation

Both software stubs and the software API emulation were very useful to implement the early data flows with the graphs. This allowed early development of the higher level software stack. However, they provided very little value to the verification of the actual graphs running on the target hardware.

Hybrid system emulation was the first environment that represented the actual hardware. The environment consisted of a combined set of virtual platforms implemented in software and actual hardware RTL (Register-Transfer Level) implementations running in emulation. Hybrid emulation provided accuracy in the components where it was needed. At the same time hybrid emulation allowed faster execution through the more inaccurate software implementations for the less important blocks. Figure 5.4 shows the hybrid system emulation setup.

In this environment, the main area of focus was to verify that the hardware and the firmware running on the hardware was working as was expected in a system-wide environment. Real hardware, firmware and software could be executed all the way up to the graphs. Because of the software API emulation that allowed the higher layer software development, it was not considered necessary to introduce those to the emulation. Functional verification of the graphs was deemed sufficient instead.

The major drawback of the system emulation environment is that in order to be able to emulate the complex hardware, large amounts of computing power is required. Even with the presence of dedicated servers and different accelerators to carry out this job, the time it takes to perform even small tasks is long. On the other hand, if

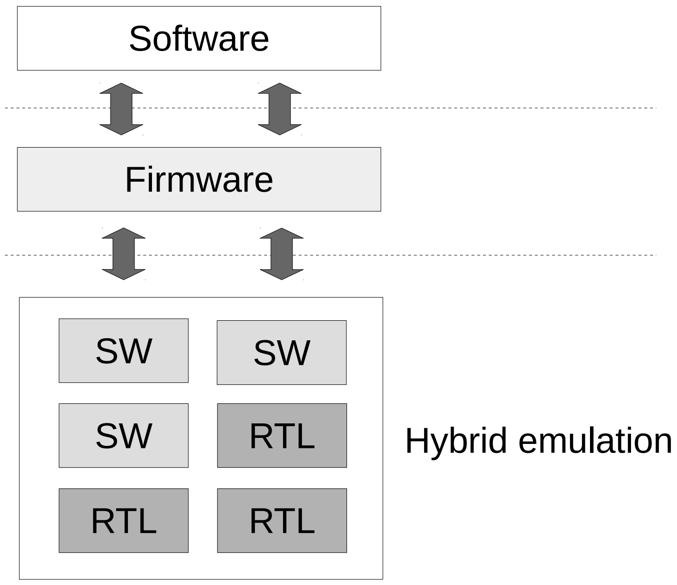


Figure 5.4 *Hybrid system emulation using both software and RTL to emulate the behavior of the hardware.*

the accuracy would be lowered and the speed increased, emulation would loose its point and the verification results would be indecisive.

Due to the time limitations, it made sense to run as small tasks as possible with the hybrid emulation environment. Being able to execute only a single vertex provided the means to do this, while being still able to verify that the output of the vertex is what was expected. Both subjective assessment and bit-exactness comparison were performed successfully. Performance and stability tests were out of the question in this environment.

5.3.4 Host PC and FPGA

While the aim of the hybrid system level emulation is to provide a whole emulated system that can be used in low level software development and verification, also a more focused approach can be used. One device that can be used in hardware emulation is an FPGA. FPGAs can be programmed to resemble the target silicon and used in the development similarly as the real hardware.

FPGAs are faster than software based simulation. They are, however, slower than the real hardware. Even if the speed of the FPGAs used in prototyping is not suitable for real time applications, they can be used for development in a reasonable way without hours of waiting for each iteration of software execution to complete.

In this project, FPGAs with the relevant blocks of the hardware being present were used for prototyping. The FPGAs were used in an environment where the device was connected to a host PC running a standard Linux distribution. Software was executed in the PC while the FPGA was used to provide the hardware implementations required by the vertices making it a hybrid emulation environment. Figure 5.5 describes the FPGA emulation setup.

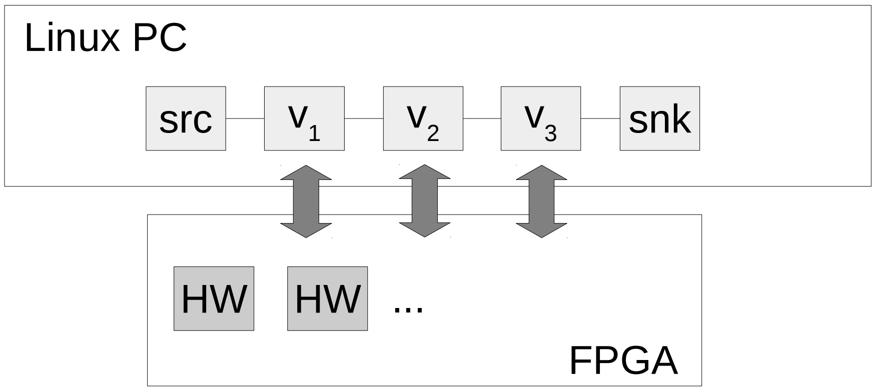


Figure 5.5 *Hybrid FPGA emulation. Graph framework and test related vertices are running on a generic Linux PC. FPGA is used to emulate hardware that the graph vertices are implemented for.*

Since the framework used was cross-OS capable, it was possible to use the graph implementations on the host PC's generic Linux distribution. All the available tools and utilities from the distribution were available, making it an efficient platform to do development and verification on. Since the higher level software development was still effectively ongoing on the software API emulation layer, only graph verification was done with the FPGA setup.

The FPGA environment allowed us to run all the required use cases on a near-realtime device. Many different tests could be enabled on the FPGA. Functional tests with subjective image quality assessment and bit-exactness testing could be performed, initial performance tests could be executed and the performance of different parts of the graphs profiled, and even some stability testing could be done. In fact, the FPGA setup was an environment where the true powers of the graphs really started to show.

5.3.5 Actual device

While all the previous environments provide the means to develop and test parts of the stack, eventually it all comes together in the actual device. This is where the whole software stack is put together, and everything is supposed to work together. Even if the core functionality has been tested in the earlier environments, there are always differences in comparison to the real device. The speed of the hardware is now much faster, potentially revealing timing related issues, or there may be hardware changes or bugs when compared to earlier revisions which are not known in advance but manifest themselves now.

Enabling the features on the actual real device starts from the low level software, gradually progressing step by step higher in the stack until the complete software stack is enabled and functional. The testing done in the earlier development environments has already proven the functionality of the different vertices and the graphs consisting of these elements. This provides the developers a known baseline that should be achievable.

The same tests can obviously be executed on the real device as well. Using the tests working in the previous environments to begin with, any issues related to the untested parts of the software stack can be ruled out. Also, since the graphs are an integral part of the rest of the software stack, no functionality can be achieved without those working.

The integration began from the low level software advancing step by step to the graphs. The verification that was done in the previous environments had been already used to solve the issues that were found during the development phase. All the tests that were able to pass in the FPGA environment were automatically enabled and passing on the real device. This in turn enabled the final adaptation of the use cases.

After the software stack was properly running on the device, the tests were used to drive the development of the current features and the integration of new ones. Any tests that were already passing were included in regression testing for all the later changes. Tests were implemented in a test-driven way to integrate new features.

With the real silicon, also the performance testing properly began. Estimations based on incomplete development platform results were no longer required. On silicon, it could be measured how the graphs were performing, and separate vertices could be benchmarked to find any bottlenecks in the graph. Stability testing could at this point also be performed properly. Longer time periods of testing could be performed than for example in the FPGA environment, as the silicon was significantly faster and more stable.

6. FUTURE POSSIBILITIES

Graph usage in the image processing system was a new concept for us that saw its first use in this project. Because of the new concept, there was no earlier foundation of verification available in this level. The usage of the test framework and the implementation of all the tests were done during the project. With the test assets that were built during the project and the knowledge obtained from graph usage and verification, more advanced utilization of graphs and their verification could be performed in future projects starting already in the early phases of the product development.

One useful feature would be the automation of the frame content tests using objective image quality assessment. A set of test images could be used and the output from the graph would be automatically compared to a reference image using one or multiple objective assessment algorithms. In addition to being very useful in developing the vertex implementations, this would be especially useful in continuous integration, where changes to either software or firmware are tested. For each change, these tests could be automatically executed, and no human interfering would be required to inspect the image quality as long as no threshold value is exceeded.

Another improvement would be to use a finer granularity in the graphs. In this project, the granularity of the graphs was rather coarse. The coarse granularity was used in order to achieve performance improvements by allowing implementation's internal communication and dataflow to be platform specific without the need of exposing the intermediate data to software. However, it would be possible to combine the use of a finer granularity graphs and the platform specific performance optimizations by allowing the vertices to communicate and transfer data in an implementation defined way. This would still retain the modularity and the possibility of reorganizing the vertices. Also, single vertex verification could still be performed by exposing the data in a generic way when the testing requires it.

The algorithm development could also benefit from the graphs. In this project, algorithms and the implementations based on those algorithms were both developed separately in different environments. Algorithms were verified using different tools that were not part of the graph verification. With the groundwork for graph verification now done, the algorithm development could start utilizing the graphs. This would allow the algorithm development to benefit from the test assets introduced with the graph verification. Since the implementation behind the vertices is not restricted, algorithms using any development platform could be developed and tested as a part of the rest of the graph. This could allow for example objective image quality assessment to happen automatically with the addition of subjective evaluation by the developer. Also, it would bring the algorithms closer to the actual product, easing the integration and lowering the risk in the future.

7. CONCLUSIONS

This thesis presented an approach to handle the ever-increasing issues with modern image processing systems. Time to market demands placed on the products along with the ever-increasing complexity of the systems put strain on the development process. Graphs can be used to handle and hide the complexity while at the same time being modular to allow efficient shift left development and verification of separate processing elements. Graph verification provides the means to verify the functionality, performance and stability of the graphs and their vertices in isolation.

Graphs were used in a real life product development project with success. Graph verification began in the very beginning of the project continuing all the way until the finalized product. Tests were implemented when new features were integrated and used for regression testing after that. Shift left was exercised by testing the graphs in many different pre-silicon environments without the burden of the whole complex software stack. Modularity of the graphs was used to test vertices in isolation. Functional, performance and stability tests were implemented and used.

Graphs and their verification provided several advantages to the product development in comparison to a traditional system that directly accesses the underlying implementations. With graphs, the complex functionality could be isolated into an individual entity. This allowed the verification of the graph as a whole in isolation without the rest of the software stack in an early phase of the product development. On the other hand, the modularity of the graphs allowed access to individual vertices in contrast to a single black box. This permitted finer granularity verification which was especially useful in the integration of new features and graph troubleshooting. Modularity was also used to allow test specific features to be integrated to the graphs. This allowed more advanced verification tightly integrated with the graphs.

Functional tests were successfully used to identify issues with new features that were to be integrated and to fix those in the early stages of the development. Functional tests were also used for regression testing. This protected efficiently against faulty changes being introduced to the implementations. Performance bottlenecks in the graphs were located and improved with the help of the performance tests. Performance tests were also used to monitor the overall graph performance during the development process. Stability testing was performed and unstable vertices were fixed whenever those were found.

In the future, finer granularity graphs could be used to allow more fine-grained development and verification of the vertices. This would further increase the possibilities of utilizing the graph modularity. Also, more advanced image quality metrics could be integrated. Combined with a fine-grained graph, this would allow more efficient and isolated development of the vertices.

BIBLIOGRAPHY

[1] *3A Modes and State Transition*, Android Open Source Project. [Online]. Available: https://source.android.com/devices/camera/camera3_3Amodes.html

[2] *H.264 video compression standard. New possibilities within video surveillance.*, AXIS Communications. [Online]. Available: http://www.axis.com/files/whitepaper/wp_h264_31669_en_0803_lo.pdf

[3] S. Battiato, A. Bruna, G. Messina, and G. Puglisi, *Image Processing for Embedded Devices*, ser. Applied digital imaging. Bentham Science Publishers, 2010.

[4] B. Bayer, "Color imaging array," Patent, July 20, 1976, US Patent 3,971,065. [Online]. Available: http://www.google.com/patents/US3971065

[5] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *Signal Processing, IEEE Transactions on*, vol. 49, no. 10, pp. 2408–2421, Oct 2001.

[6] C. Biber, S. Ellin, E. Shenk, and J. Stempeck, "The polaroid ultrasonic ranging system," in *Audio Engineering Society Convention 67*, Oct 1980. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=3680

[7] J. Bondy and U. Murty, *Graph Theory With Applications.* Elsevier Science Publishing Co., Inc., 1976, 0-444-19451-7.

[8] S. C. Carlson, "graph theory," Encyclopædia Britannica, 2016, Available: http://www.britannica.com/topic/graph-theory.

[9] D. M. Chandler and S. S. Hemami, "VSNR: A wavelet-based visual signal-to-noise ratio for natural images," *IEEE Transactions on Image Processing*, vol. 16, no. 9, pp. 2284–2298, Sept 2007.

[10] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, Dec 2010, pp. 225–236.

[11] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, Nov 2014, pp. 201–211.

[12] “Image sensor architectures for digital cinematography,” DALSA Digital Cinema, Available: http://www.teledynedalsa.com/public/corp/PDFs/papers/Image_sensor_Architecture_Whitepaper_Digital_Cinema_00218-00_03-70.pdf.

[13] R. Diestel, *Graph Theory.* New York: Springer-Verlag, 2000, 0-387-95014-1.

[14] E. Dumic, M. Mustra, S. Grgic, and G. Gvozden, “Image quality of 4:2:2 and 4:2:0 chroma subsampling formats,” in *ELMAR, 2009. ELMAR ’09. International Symposium*, Sept 2009, pp. 19–24.

[15] Y. Fang, “Application-specific visual quality assessment: Current status and future trends,” in *Proceedings of the 7th International Conference on Internet Multimedia Computing and Service*, ser. ICIMCS ’15. New York, NY, USA: ACM, 2015, pp. 87:1–87:4. [Online]. Available: http://doi.acm.org/10.1145/2808492.2808579

[16] “YUV pixel formats,” fourcc.org. [Online]. Available: http://fourcc.org/yuv.php

[17] *X3 Technology, Direct Image Sensors*, Foveon. [Online]. Available: http://www.foveon.com/article.php?a=67

[18] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design: Modeling, Synthesis and Verification*, 1st ed. Springer Publishing Company, Incorporated, 2009, pp. 255-285.

[19] *Google Test*, Google. [Online]. Available: https://github.com/google/googletest

[20] K. Gottfried and S. Peter, “A method and apparatus of contrast-dependent sharp focussing,” Sept. 19 1972, US Patent 3,691,922. [Online]. Available: http://www.google.com/patents/US3691922

[21] gstreamer, “open source multimedia framework.” [Online]. Available: http://gstreamer.freedesktop.org/

[22] B. Gunturk *et al.*, “Demosaicking: color filter array interpolation,” *Signal Processing Magazine, IEEE*, vol. 22, no. 1, pp. 44–54, Jan 2005.

[23] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, appendix A.

[24] *HSA Platform System Architecture Specification*, HSA Foundation, 2015, "Version 1.0". [Online]. Available: http://www.hsafoundation.com/standards/

[25] D. Hubel, *Eye, Brain, and Vision.* Scientific American Library, 1995. [Online]. Available: http://hubel.med.harvard.edu/book/bcontex.htm

[26] "Subjective video quality assessment methods for multimedia applications," International Telecommunication Union, Apr 2008, recommendation ITU-T P.910.

[27] K. Jack, *Video Demystified: A Handbook for the Digital Engineer, 5th Edition*, 5th ed. Newton, MA, USA: Newnes, 2007.

[28] A. A. Jerraya, "Long term trends for embedded system design," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, Aug 2004, pp. 20–26.

[29] S. B. Kang, M. Uyttendaele, S. Winder, and R. Szeliski, "High dynamic range video," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 319–325, July 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882270

[30] *The OpenCL Specification*, The Khronos Group Inc., "Version 2.1". [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf

[31] *OpenMAX, The Standard for Media Library Portability*, The Khronos Group Inc.

[32] *OpenMAX Development Layer API Specification*, The Khronos Group Inc., 2007, version 1.0.2.

[33] *OpenMAX Integration Layer Application Programming Interface Specification*, The Khronos Group Inc., 2008, version 1.1.2.0.

[34] *OpenMAX Application Layer Application Programming Interface Specification*, The Khronos Group Inc., 2011, version 1.1.

[35] *The OpenVX Specification*, The Khronos Group Inc., 2014, version 1.0.1.

[36] J. Korneliussen and K. Hirakawa, "Camera processing with chromatic aberration," *Image Processing, IEEE Transactions on*, vol. 23, no. 10, pp. 4539–4552, Oct 2014.

[37] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[38] J.-S. Lee *et al.*, "An advanced video camera system with robust AF, AE, and AWB control," *Consumer Electronics, IEEE Transactions on*, vol. 47, no. 3, pp. 694–699, Aug 2001.

[39] *Motion JPEG 2000 File Format*, Library of Congress, Digital Preservation. [Online]. Available: http://www.digitalpreservation.gov/formats/fdd/fdd000127.shtml

[40] W. Lin and C. C. Jay Kuo, "Perceptual visual quality metrics: A survey," *J. Vis. Comun. Image Represent.*, vol. 22, no. 4, pp. 297–312, May 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jvcir.2011.01.005

[41] *Linux Media Infrastructure API*, LinuxTV Developers. [Online]. Available: https://linuxtv.org/downloads/v4l-dvb-apis/

[42] R. Lukac, *Single-Sensor Imaging: Methods and Applications for Digital Cameras*, ser. Image Processing Series. CRC Press, 2008.

[43] *Introduction to DirectShow Application Programming*, Microsoft Corporation. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms786509(VS.85).aspx

[44] L. Mijatovic, H. Dean, and M. Rozic, "Implementation of algorithm for detection and correction of defective pixels in FPGA," in *MIPRO, 2012 Proceedings of the 35th International Convention*, May 2012, pp. 1731–1735.

[45] P. Mohammadi, A. Ebrahimi-Moghadam, and S. Shirani, "Subjective and objective quality assessment of image: A survey," *CoRR*, vol. abs/1406.7799, 2014. [Online]. Available: http://arxiv.org/abs/1406.7799

[46] "Understanding MPEG-4: Technologies, advantages and markets," The MPEG Industry Forum. [Online]. Available: https://www1.ethz.ch/replay/docs/whitepaper_mpegif.pdf

[47] J. Nakamura, *Image Sensors and Signal Processing for Digital Still Cameras.* Boca Raton, FL, USA: CRC Press, Inc., 2005.

[48] R. Ramanath, W. Snyder, Y. Yoo, and M. Drew, "Color image processing pipeline," *Signal Processing Magazine, IEEE*, vol. 22, no. 1, pp. 34–43, Jan 2005.

[49] H. R. Sheikh and A. C. Bovik, "Image information and visual quality," *IEEE Transactions on Image Processing*, vol. 15, no. 2, pp. 430–444, Feb 2006.

[50] I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014, pp. 4-1 - 4-16.

[51] N. Stauffer, “Range determination system,” Jan. 22 1980, US Patent 4,185,191. [Online]. Available: http://www.google.com/patents/US4185191

[52] G. Sullivan and S. Estrop, *Recommended 8-Bit YUV Formats for Video Rendering*, Microsoft Corporation, 2002, updated 2008. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/dd206750(v=vs.85).aspx

[53] J. Teich, “Hardware/software codesign: The past, the present, and predicting the future,” *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, May 2012.

[54] G. Wallace, “The JPEG still picture compression standard,” *Consumer Electronics, IEEE Transactions on*, vol. 38, no. 1, pp. xviii–xxxiv, Feb 1992.

[55] Z. Wang, “Applications of objective image quality assessment methods [applications corner],” *IEEE Signal Processing Magazine*, vol. 28, no. 6, pp. 137–142, Nov 2011.

[56] Z. Wang and A. C. Bovik, “Mean squared error: Love it or leave it? a new look at signal fidelity measures,” *IEEE Signal Processing Magazine*, vol. 26, no. 1, pp. 98–117, Jan 2009.

[57] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “Image quality assessment: from error visibility to structural similarity,” *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, April 2004.

[58] Z. Wang, H. R. Sheikh, and A. C. Bovik, “Objective video quality assessment,” in *In the Handbook of Video Databases: Design and Applications.* CRC Press, 2003, pp. 1041–1078.

[59] Y. Yoshida, S. Shinohara, H. Ikeda, K. Tada, H. Yoshida, K. Nishide, and M. Sumi, “Control of lens position in auto-focus cameras using semiconductor laser range finder,” in *Circuits and Systems, 1991., Proceedings of the 34th Midwest Symposium on*, May 1991, pp. 395–398 vol.1.

[60] Y. Zheng, S. Lin, C. Kambhamettu, J. Yu, and S. B. Kang, “Single-image vignetting correction,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 12, pp. 2243–2256, Dec 2009.

[61] D. Zorin and A. H. Barr, “Correction of geometric perceptual distortions in pictures,” in *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH ’95. New York, NY,

USA: ACM, 1995, pp. 257–264. [Online]. Available: http://doi.acm.org/10.1145/218380.218449