Image Processing on limited devices

Markus Eriksson

July 7, 2007
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Pedher Johansson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

As cell phones become more of an entertainment device, rather than just a wireless phone, and today holds a camera with several mega pixels resolution, the need of image processing on the handset becomes useful. To be able to run advanced effects on large images on a device with limited memory and CPU power, an efficient image processing engine has been developed. With this engine the user can apply simple per pixel filters and more advanced filters like convolution and transformation filters. This thesis designs and implements such an engine and a set of filters, suited to run on a cell phone. This thesis was done at the Graphics department at Sony Ericsson Mobile Communications (SEMC) in Lund.

Acknowledgements

I would like to thank the following persons for helping me with this thesis:

– Andreas Agvard, my supervisor at SEMC Graphics for the input and advice for development

– Pedher Johansson, my supervisor at Umeå University

– Alexander Klintström, for help getting me this thesis at SEMC

– Srdan Boskovic, for giving me the chance to do the thesis at SEMC

Acronym and abbreviations

NMT	Nordisk Mobil Telefoni, analog cell phone standard
GSM	Global System for Mobile communications, digital cell phone standard
3G	Third generation cell phone standard, current today together with GSM
4G	Future cell phone standard
CIF	Common Intermediate Format, video resolution of 352 x 288 pixels
NVIDIA	NVIDIA Corporation, maker of graphic cards
RAM	Random access memory, memory where program resides during execution
ARM	Advanced RISC Machine, company making CPUs for embedded systems
SEMC	Sony Ericsson Mobile Commuication
GUI	Graphical User Interface

Contents

Chapter 1

Introduction

Today.s cell phones are not just used to talk to other people but to take pictures, listen to music and send images. The cameras are ever-increasing in resolution with today.s 3 mega pixel and tomorrow's 5 mega pixel. If a user wants to alter a photo it is unlikely that the person will transfer the file to a PC, process it, and then transfer it back to the phone to send it. Therefore, a simple and fast image processing on the cell phone is needed.

This thesis was done at Sony Ericsson Mobile Communication (SEMC) at the Graphics department during spring of 2007. This report will focus on cell phones from SEMC.

1.1 From wireless phone to multimedia device

One of the first cell phones was developed at Motorola during the 1970.s and the first cell phone available to the public market in US was the Motorola DynaTAC 8000X in 1983 [29]. The first commercial cell phones in Sweden used the NMT (Nordisk MobilTelefoni) network which was publicly available in 1981. The phones of this time could make and receive voice calls and in some cases transfer data via modem at rates up to 9600 bits per second. NMT is counted as the first generation, 1G, cell phone standard.

NMT was followed by the second generation, 2G, GSM in the beginning of 1990's. GSM phones could send and receive voice calls, send text messages and do data transfers up to 57.6 kbit/s [10].

In early 2000.s extensions to GSM was made making it possible to surf the mobile web via WAP and send multimedia messages with images, sounds and text called MMS, Multimedia Messaging Service. At this point cell phones had begun to transform from just handling voice calls and text messages, to be equipped with color displays, cameras and music playback capabilities. The resolution of the cameras was typically 288×352 pixels (CIF) which is about 0.1 mega pixels.

Today, the third generation networks (commonly known as 3G) with greater transfer rates than previous networks are being used in public. The transfer rates currently peak at 3.6Mbit/s downstream, and 384kbit/s upstream. This gives the end-user the possibility to do video calls and enjoy mobile broadband.

Over the years the cell phone has transformed from just being a wireless phone into becoming a multimedia device. Today a top level cell phone holds a camera, mp3 playback capabilities, games with accelerated graphics, video streaming, and a web browser. A phone of today can take pictures with almost the same resolution as a compact camera, (today 5 mega pixels Nokia N95 [26]).

The technological advances show no signs of slowing down with the 4G planned to be released in 2010 [6] enabling transfer rates of 100 Mbit/s [32]. Also cameras continue to increase in resolution with Samsung.s SCH-B600 shown at Cebit 2006 with 10 mega pixels [7]. A few cell phones even have hardware support for 2D/3D accelerated graphics, e.g., Samsung SPH-G1000 [30] and Sony Ericsson W900i [28], the latter equipped with the NVIDIA GoForce 4800 chip.

1.2 The need of an efficient filter engine

The increasing number of pixels are however, not matched by the performance of the cell phone processor, mainly due to an increased power consumption, resulting in a shorter stand-by time. The amount of RAM available on the phone is also a limitation to typically an amount of 16 MB.

To be able to process large images on a limited platform like a cell phone, an efficient filter engine is necessary where the constraints of the CPU speed and memory space is considered. For example a JAVA program running on a SEMC phone can allocate somewhere around 500kB of heap memory [9]. To be able to process an image of 5 mega pixel (15 MB of data) the engine has to be able to work on smaller parts on the image in sequential pass. At the same time the engine has to be CPU effective. The CPU of today.s phones from SEMC is ARM9 with clock-speed of about 200 MHz [8].

With these prerequisites and the current state of image processing capabilities of cell phones from SEMC it is necessary to develop an efficient filter engine. The current image processing software can not apply any advanced effects, for example, kernel filters. For a more exhaustive list of capabilities of image processing see Chapter 4.

1.3 Goal

The goal of this thesis is to develop an efficient filter engine capable of processing 5 mega pixel images (and possibly even larger) in an efficient way. Memory is the primary concern, but CPU power constitutes a limit too. A set of filters is to be implemented, both simple per pixel filters and more advanced, like kernel filters. The priority is speed and low memory consumption rather than perfect photographic quality.

1.4 Method

The application is to be developed in a desktop environment in the C programming language using Microsoft.s Visual C++ 6. The programming language is chosen because it is memory and CPU effective (if correctly implemented), and is already used for the phone development at SEMC, which makes the integration process easy. The implementation should also be able to run on a cell phone from Sony Ericsson.

1.5 Related work

There are commercial companies doing similar products this thesis is aiming at. One of theses companies is Scalado [31] in Lund with their product CAPS.

Chapter 2

Image Processing on Computers and Mobile Devices

2.1 General image processing

Digital image processing is a subset of the signal processing field. The main difference between signal processing in general and the digital subset is the non-continuous form the data is presented. General signal processing works on continuous and discrete signals where digital image processing only works on discrete ditto.

A digital image is a collection of signals measured in a short time and usually arranged in a two dimensional grid of color elements known as pixels. A pixel is the smallest part of an image and usually represents a color or a value.

2.2 Color representation

The most common way to represent a pixel is RGB: Red, Green and Blue with floating point values from 0 to 1. Sometimes RGBA is used where the A represents the alpha value describing transparency level. The RGB color model is additive, used in computer displays and TV sets. By being additive means that some element emits a certain color, which is the case in a CRT display. All RGB channels set to zero yields black. By mixing the amount of the three colors great number of colors can be described.

The human eye can recognize a wide variety of colors. In 1920s W. David Wright (1928) and John Guild (1931) independently laid the foundation to the CIE XY 1931, a color diagram where all the colors visible to the human [16] is represented. Many devices, (e.g., a CRT display) can not display all colors in the CIE XY 1931. Hence, the reproductable colors on a device are a subset of colors known as the gamut. In 1995, Microsoft and Hewlet-Packard proposed a standard, sRGB, with a gamut suitable for most available devices [23]. The sRGB which also became the internet standard of representing RGB colors. One may notice that the gamut often cover a relatively small part of the complete color space visible to a human eye. Adobe has created another color space, "Adobe RGB Color Space", defining a larger set of colors [2].

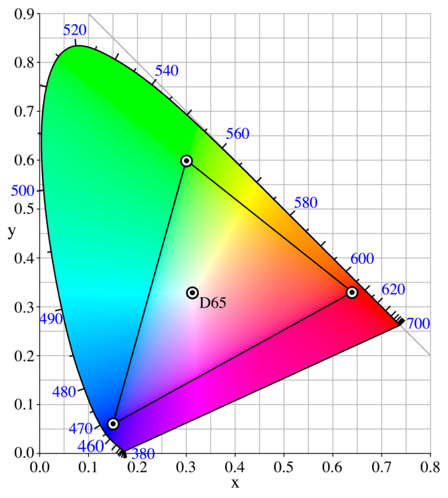

Figure 2.1: CIExy1931 color space projected such that each color has the same brightness (also known as the CIE chromaticity diagram). The projected sRGB gamut is drawn inside. D65 denotes the white point at 6500 K color temperature (not covered in this thesis)

One other common color model is CMY, short for Cyan, Magenta and Yellow. This model is opposed to RGB a subtractive color model, meaning it absorbs some color when exposed to light. Conversion from RGB to CMY is done by:

$[CMY]^T = [111]^T - [RGB]^T$

CMYk is a variant of CMY used for printing where the k stands for key describing the amount of black. The letter k was chosen over b as in black to avoid confusion with RGB where B denotes the blue component. Addition of black contributes to lower consumption of ink when printing black with black ink instead of the CMY at maximum level and does not give perfect black either due to lack of total absorption of light. Each channel in RGB and CMY color model can be seen as orthogonal vectors in a three dimensional space. With this representation the relationship between RGB and CMY becomes clear. Along the vector from black (0,0,0) to white (1,1,1) the gray colors are represented.

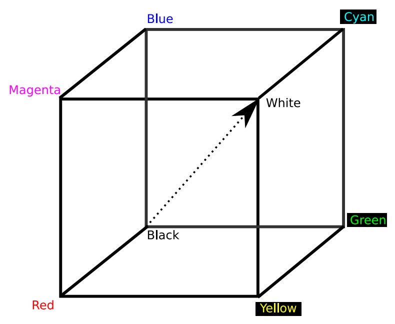

Figure 2.2: Color cube showing the relation RGB vs. CMY

A third way of representing colors is the HSV/HSB color model which is short for Hue, Saturation and Value/Brightness. The hue is represented as degrees $[0, 360]$ where $0°$ is red, $120°$ is green and $240°$ is blue. Saturation the purity of the color, a value of 0 indicates gray. The saturation, or the purity of the colors are represented either as a floating point value between zero and one, or in percent.. The same holds for the brightness. Zero saturation means a gray color. HSV color model can be compared to an artist's palette where hue is the color pigment (at full saturation), saturation the amount of white to bleach the color and brightness the (inverted) amount of black color mixed in to darken the result.

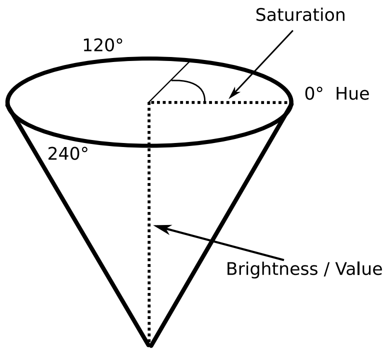

Figure 2.3: HSV color model represented as a 3D cone

The last color space described is the YCbCr color space where Y represents the luminance (black to white) component, Cb blue chrominance (chroma for short) and Cr red chrominance. Chrominance is a relation between luminance and blue Cb, where Cr is the relation between luminance and red. RGB can easily be converted to YCbCr [14] with the following relationship:

$$Y = 0.299R + 0.587G + 0.114B$$
$$Cb = -0.1687R - 0.3313G + 0.5B + 128$$
$$Cr = 0.5R - 0.4187G - 0.0813B + 128$$

This yields YCbCr with channel values [0, 255].

YCbCr was first developed as a standard for transmitting color TV signals. The first color TV experiments used RGB signals but this was not compatible with the current monochrome TV sets. When YCbCr was used the Y channel was sent as the monochrome channel while the color TV sets added the CbCr channels to achieve color display. YCbCr is used today in a different context, the JPEG compression.

2.3 Image representation

The most common way to represent a binary image is by a 2-dimensional grid of pixels. Each pixel is usually made up of [0, 255] levels (8 bits) in three channels (R, G, and B), i.e., 24 bits in total. Greater resolution per channel is used in other formats as OpenEXR, Photoshop PSD, and JPEG-2000 with up to 31 bits per channel in the latter. An image with 24 (or more) bits can consume a lot of storage space. Therefore it is usually compressed with either a lossy or lossless algorithm. A lossless format restores exactly the same image as the original whereas the lossy format discards information (hopefully) not perceived by the human eye to save space. The most common format for lossy compression is the JPEG format. The PNG (Portable Network Graphics) is an example of an open format where lossless compression is possible[13].

2.3.1 JPEG file format

To compress an image with the JPEG standard, three major steps are taken

– Block splitting, color space conversion, and down-sampling.

– Discrete Cosine Transform (DCT) and quantization.

– Huffman encoding.

The algorithm works on a smaller part of an image at a time. First the image is split into squares of 16 x 16 pixels (when 16 rows or columns can not be retrieved it is filled with zeroes). Depending on which down-sampling method used (here the YUV420) this area is again spilt, into four squares and converted to YCbCr as described above giving four 8x8 squares. In YUV420 the Y-channel is kept for every pixel but every second CbCr values (along X- and Y-axis) are skipped. Because the human eye is more sensitive to change in brightness compared to colors this compression technique produce images with almost no notable difference.

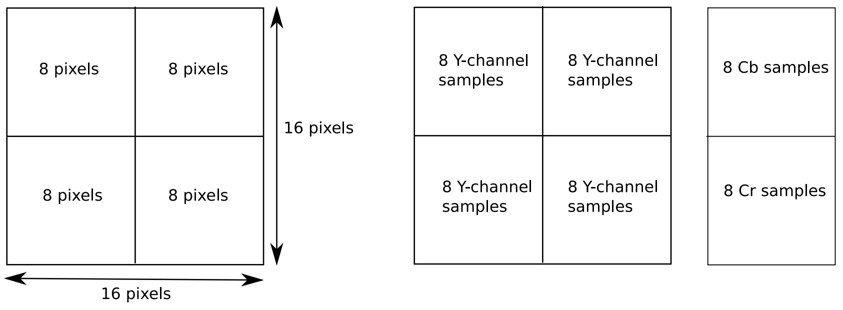

Figure 2.4: YUV420 down-sampling: Left is 64 x 3 bytes of RGB data which becomes the middle and right parts after YCbCr conversion and YUV420 down-sampling, 64 x 1 bytes for intensity, 16 x 1 bytes for Cb and Cr channels resulting in 192 bytes compressed to 96 bytes.

To compress further each 8x8 down sampled channel is first centered on zero by subtracting a scalar of 128, and then run through the DCT (Discrete Cosine Transform type II) giving an 8x8 coefficient matrix. This 8x8 grid is quantized by subtracting a quantization matrix developed to keep the first values (low x- and y-indices) and removing the terms at higher indices.

After that step the resulting matrix is very well suited to compress with Run Length Encoding, RLE with the well known Huffman coding (see Figure 2.8). Because of many repeating zeroes the compression ratio is great. To optimize the RLE encoding even further the sub image is not processed line wise but in a zigzag pattern. It is clear that using this method to traverse the matrix, the number of consecutive zeroes is greater than line wise, hence better compression.

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix} \begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix}$$

Figure 2.5: Left: Y-channel data for an 8x8 sub-image. Right: 128 subtracted from Y-channel.

$$\begin{bmatrix} -415 & -30 & -61 & 27 & 56 & -20 & -2 & 0 \\ 4 & -22 & -61 & 10 & 13 & -7 & -9 & 5 \\ -47 & 7 & 77 & -25 & -29 & 10 & 5 & -6 \\ -49 & 12 & 34 & -15 & -10 & 6 & 2 & 2 \\ 12 & -7 & -13 & -4 & -2 & 2 & -3 & 3 \\ -8 & 3 & 2 & -6 & -2 & 1 & 4 & 2 \\ -1 & 0 & 0 & -2 & -1 & -3 & 4 & -1 \\ 0 & 0 & -1 & -4 & -1 & 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Figure 2.6: Left: the sub-image after DCT. Right: Typical quantization matrix

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.7: Sub-image after division with quantization matrix. (All matrix images courtesy of Wikipedia [22])

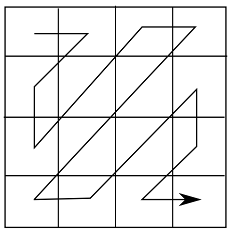

Figure 2.8: Zigzag pattern

To decode a JPEG image the whole chain is reversed: decode Huffman data, multiply with the quantization matrix, run the inverted DCT (Discrete Cosine Transform type III), re-center around 128 and build RGB data from the 16x16 blocks by interpolating the CbCr values.

For more in depth on JPEG see "The JPEG Still Picture Compression Standard. Summary by Gregory K. Wallace. [35]"

2.4 Image filters

2.4.1 Per pixel filters

The most obvious way to alter an image, is with a per pixel filter. An input pixel becomes an output pixel after some altering function is applied, i.e., $filterfunction(input(x, y)) = output(x, y)$. In spite of their simplicity, these filters can help enhance an image greatly; e.g., per pixel filters are used to change the brightness, get the negative, grayscale and change the contrast or the gamma of an image.

To visualize this we compare the input to the output in the following images. The left column describes the linear relationship, input and output data is the same. The right column describes the input versus the output and how the values change.

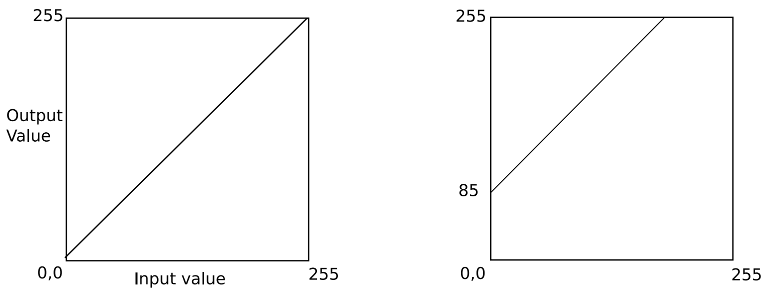

Figure 2.9: Example of brightness increase: brightness range [0, 255] becomes [85, 255]

To adjust the global brightness of an image, each pixel value (the triplet of RGB) is increased (or decreased) with a scalar. This changes the brightness. Values exceeding the maximum are clamped to 255.

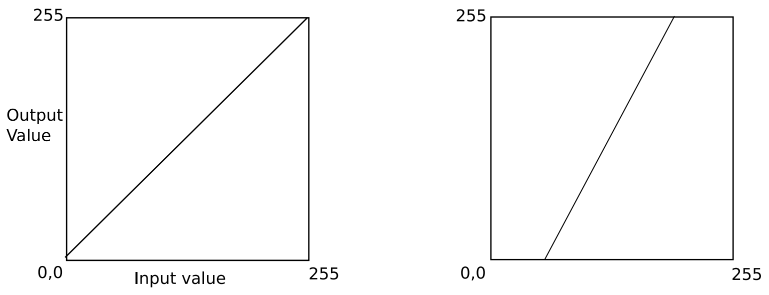

Figure 2.10: Contrast = 100 vs. contrast . 150%

Changing a pixel.s contrast is done by increase or decrease the gradient. By changing the contrast relationship in an image the detail level can be increased. Decreasing the contrast will make the image look grayish.

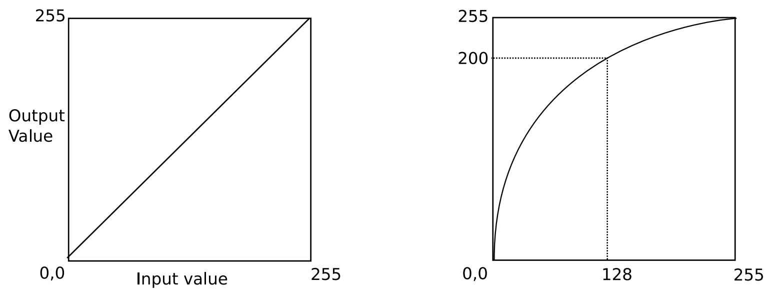

Figure 2.11: γ (gamma) 1.0 vs. increase, γ 2.2

To change the gamma of an image a non linear transformation is used: $componentvalue^{\gamma} = componentvalue'$ where γ typically range from 0.5 to 3.5. Gamma can be compared to brightness adjustment, but with the non linear relationship. The gamma control operation is useful to compensate for different attributes of brightness computer displays can have, which is called gamma correction, not covered by this thesis. To learn more about gamma correction see [5].

Other common per pixel filters are invert and grayscale. To invert a pixel, the current value (one per channel) is subtracted from the channel maximum. As seen in the following example: ($[129, 57, 95]$ is a RGB pixel in vector representation).

$$Invert[129, 57, 95] = (255 - 129, 255 - 57, 255 - 95) = [126, 198, 160]$$

To grayscale a pixel the average is calculated as in example below. However, the different channels are often weighted due to how the human eye percept the luminance of different colors.

$$Grayscale[129, 57, 95] = \frac{(129 + 57 + 95)}{3} = [93, 93, 93]$$

2.4.2 Kernel filters

While per pixel filters depend on only one pixel, the kernel filters depend also on the pixel's neighbors. Kernel filters can be used for a number of different effects like sharpening, blurring, and edge detection.

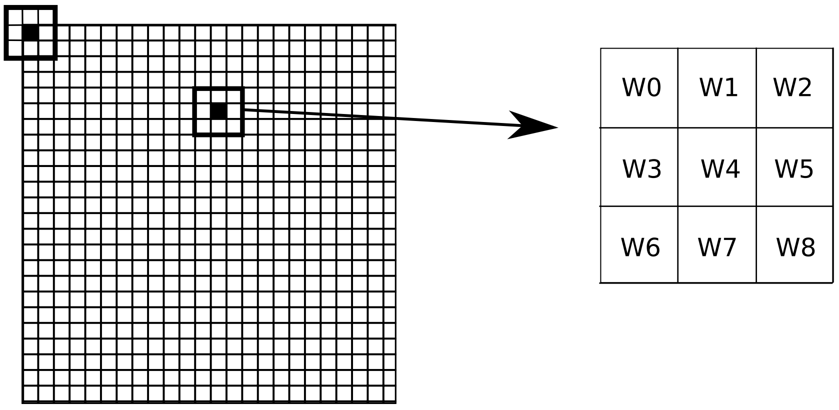

Figure 2.12: Left: An image with two positions of a 3x3 kernel. Right: A 3x3 kernel with weights (index 0 to 8)

1	1	1
1	-8	1
1	1	1

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Figure 2.13: Typical kernels left to right: Laplace edge detection and Gaussian blur

To calculate a resulting pixel, surrounding pixels are weighted and summed together by:

$$\sum_{i=1}^{k} w[i] * pixel[i]$$

While this may overflow the representation of a pixel value (255) or lead to a "white out" picture it is often normalized with the sum of all weights giving:

$$(1/\sum(w[i])) * \sum_{i=1}^{k}(w[i] * pixel[i])$$

One problem with kernel filters is that they produce a smaller image than the input,

$$\frac{kernelwidth}{2} - 1$$

and

$$\frac{kernelheight}{2} - 1$$

This problem emerges when the kernel shall be placed along an outer edge or a corner (see Figure 2.12) and there is no data to fill the kernel with for these positions. To solve this problem, data is usually repeated with the closest data, filled with zeroes or output the image is shrunken.

The underlying mathematical idea of kernel filters is to combine two functions (the image and the kernel) and to produce one answer, the pixel. This is sometimes referred to as convolution or convolution filter from the mathematical convolution operator which takes two functions and produce a third ditto, in this case the resulting image when applied.

In the case of edge detection the change in derivative (the second derivative) is used to color the pixel. High difference yields a brighter pixel compared to low difference. To illustrate this, the current pixel is denoted $f(x)$ and the next pixel on the same row is $f(x+1)$. The difference in derivative is:

$$\frac{df}{dx} = f(x+1) - f(x)$$

Expanding this to two dimensions (the image represented as a two dimensional grid) giving

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

the Laplacian derivative operator. This operator is isotropic meaning it does not depend on which direction the change of derivative is. To apply this to the discrete case (pixels are not continuous) gives us two partial second order derivatives:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1,y) + f(x-1,y) - 2f(x,y)$$

and

$$\frac{\partial^2 f}{\partial y^2} = f(x,y+1) + f(x,y-1) - 2f(x,y)$$

and these two combined:

$$\nabla^2 f = (f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)) - 4f(x,y)$$

While this covers the next, previous, above and below pixel the diagonals are not covered. Adding the diagonals:

$$f(x+1,y+1) + f(x+1,y-1) + f(x-1,y+1) + f(x-1,y-1) - 4f(x,y)$$

results in:

$\nabla^2 f = (f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) +$ $f(x+1,y+1) + f(x+1,y-1) + f(x-1,y+1) + f(x-1,y-1)) - 8f(x,y)$

This yields the left kernel in Figure 2.13.

Gaussian Kernel

The Gaussian kernel weights are based on the Gaussian distribution:

$$G(u, v) = \frac{1}{(2 * \pi * \sigma^2)} * e^{\frac{-(u2+v2)}{2}}$$

where r is the kernel radius $r^2 = u^2 + v^2$ and σ is the standard deviation of the Gaussian distribution. The right kernel in Figure 2.13 is a result of the Gaussian distribution in two dimensions.

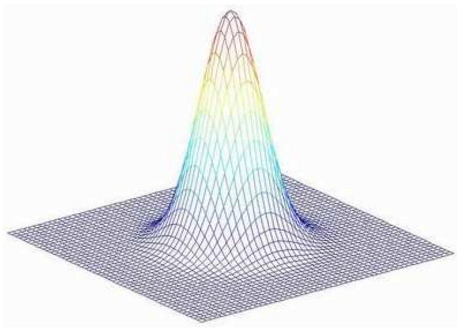

Figure 2.14: Gaussian kernel plotted in 3 dimensions with weights as the height of the cone (image courtesy of Paolo Favaro [11])

Because the Gaussian kernel does not include any subtraction of pixels, the result will be a blurry image as it sums, with weights, its surrounding neighbors. Blurring an image is necessary when resampling it to a lower resolution to avoid noise in the result, or to give a fake out-of-focus effect to an image.

Kernel filters are generally expensive to apply due to the number of arithmetic operations and the many data accesses needed. A 5x5 kernel needs 25 multiplications and 24 additions (with normalization one more multiplication) resulting in 49 (50) instructions per pixel-channel. A RGB pixel would then need around 150 instructions just to apply the kernel. It is clear that this is a very time complex filter compared to a simple per pixel filter needing four. Because of this complexity, a large kernel is not feasible on a limited device such as a cell phone. It has been shown [19] that kernel filters of size 5x5 or larger can be optimized when the kernel is symmetric along x- and y-axis. However, even after optimization, it is still a time-consuming filter.

2.4.3 HDR (High Dynamic Range) and Tone map

As mentioned before, the color representation of a pixel is made up of 8 bit values per RGB channel. While this is enough to display text and drawings, the real world.s dynamic in contrast can not be fully represented due to limitations in many steps of producing the final image on a monitor. These steps include the camera sensors, the internal data representation, the graphics card, and finally the monitor.

As an illustration the human eye has a luminance sensitivity range from 10^{-4} to 10^8 cd/m^2 (candela per square meter), a $1 : 10^{12}$ ratio, compared to an LCD panel of today that has a $1 : 1000$ ratio. This is a factor of 10^9 in difference. As mentioned earlier, the internal representation of 8 bits per channel can easily be changed to other formats with a sufficient range (e.g., OpenEXR, JPEG-2000 or PSD).

However, the standard digital camera has too low dynamic, the best around 1:8000 [20]. To resolve this, many images of the same scene are taken with either same aperture and different shutter speed, or the other way around. When these images are put on top of each other and added together they produce a much higher dynamic range than just having one image, i.e., :

$$\sum_{i=1}^{k} dynamic[i]$$

where k is the number of images. This leaves the monitor and the graphic card as limits. Today there is a monitor on the market able to display a contrast ratio of 1:200 000, the BrightSide DR37-P display from BrightSide Technologies, now acquired by Dolby. Still these monitors are not as common as the other techniques. But there is a shortcut: to display high dynamic range images on a low dynamic range (LDR) display a tone map operator is used to adapt the range to a common display. This will fit a wide contrast ratio to the narrow one of a common LCD panel of today. This will not give the same result as a HDR display but it will enhance the image.

The simplest way would be to just divide the maximum of the representation with the maximum of the HDR image, then scale all pixels with the result. This would fit the image in the displayable range but it would render a very dull and grayish result because the linear division (same denominator for all scalars) results in contrast loss.

Instead of a linear scaling operator Stockham [34] proposed a logarithmic operator where the maximum luminosity can be represented on a LDR display.

$$L_d = \frac{log(L_w + 1)}{log(L_{max} + 1)}$$

Where L_d is the display luminosity, L_w the current element.s luminosity and L_{max} the global maximum luminosity.

While this tone mapping operator greatly compresses the luminosity ratio, some of the contrast is lost in drastic changes. To mend this F. Drago et al [3] purposed a solution that locally analyzes the contrast change with a global pass of kernel filters and adopted the denominator in the above formula to keep the contrast locally. This method is based on E. Reinhard et al[33] algorithm for local adoption. The F. Drago paper also compared implementations on CPU vs. GPU. N. Goodnight et al[17] went further and implemented a real time solution for tone mapping utilizing the programmable hardware of modern graphics card.

Because of the expensive operations in these algorithms, floating point representation to keep accuracy, log and power of functions, and big kernel filters it.s not feasible to apply these algorithms using a cell phone today. A global algorithm such as the one Stockham purposed could be done.

While today's cell phones lacks programmable graphics hardware (also called shaders and fragment programs) they are likely to be equipped with this in the future [27]. Therefore, the idea of doing HDR images from a cell phone.s camera is not "off the wall". The idea would be to acquire a set of images from the cell phone's camera during very short time, hopefully short enough to avoid the need of a tripod. These images would have different exposure time to distinguish details in levels of different light.

2.4.4 None enhancing filters

Image processing on the whole works towards enhancing images to make them easier to interpret. There is however a field interested in non photorealistic processing / rendering, for short: NPR. The use of non photorealistic images can for example be an image of mechanical parts in an engine where correct colors and shadows are of low or no interest.

Other more artistic filters is called painterly-, or, stroke based-rendering where painterly comes from the artistic community describing a painting technique where the artist paints with patches instead of hard lines, compare to an oil paintings. This is possible to do in post processing of an image shown by L. Kovács and T. Sziáni [21]and in an other implementation by Nehab [25]. These two algorithms work by:

- extracting the edges to keep some of the sharp edges in the image,
- calculate a reduced set of colors to paint with,
- cluster areas where colors are approximately the same, and
- paint the result image with strokes, smaller strokes where there is sharp edges, bigger strokes where larger areas of low contrast is located.

Other methods to achieve sketched-like images are the "pen and ink" method, tracing the curves and lines in an image. While this is possible from a 3D model as input, where lighting and shadows can be easily calculated this is not yet easily done from a 2D image, as pointed out by Hertzmann [15].

Other types of NPR algorithms is toon shading where 3D models are rendered with a limited color map and hard drawn black edges, yielding a cartoon-like result. This can be done with 2D images as well where the most straight forward way is the posterizing effect. Posterizing includes clamping the current colors to a much reduced number of colors, for example an image with 40.000 colors results in having 13 distinct colors. To expand the sketch/cartoon-effect for 2D images black and white edge detection combined with a posterized color layer can be done.

2.4.5 Color map transformations

To give another artistic look to an image, the color balance can be changed. One common example is to grayscale an image to give it another feeling, e.g., an older look.

One other transformation is to mimic an old color space of the Technicolor two-strip film. This film technique was developed in the 1920's and was based on two primary colors, red and green. An frame captured by a camera was split with a prism onto two stripes of film (hence the name) via a filter, projecting red on one strip and green on the other [24].

Due to the lack of blue component and this being an additative color system the result was biased to green and red. While this technique evolved to being subtractive, and later on including three strips including blue channel, a few films were shot with this technique.

In current times, films has tried to mimic this way of displaying colors, one of them "The Aviator" [12].

This effect can be achieved in the RGB color space where a volume in the cube maps to some other volume in the resulting image, either with a one to one mapping, or three to one mapping. By one to one means that each channel can independently be translated, a one dimensional mapping. In the three to one mapping, the red output channel depends on all three input RGB channels. Same goes for green and blue channel.

2.4.6 Warp

Warp is a form of transformation, where parts of the image is translated to new positions and/or stretched/shrunken to new areas. Warping is used when applying a 2D texture to a non axis aligned plane in 3D. In the context of this thesis warping occurs in the non enhancing case, for example the face warp. This image warp can of course be applied to any part of the image, not just faces. Face warping is a way of giving faces a new look, for example adjusting the smile of one person, giving the person a happier face.

The warp can be defined either by a function:

$$f(x,y) = cos(\frac{\pi}{2} * |1-x|), sin(\frac{\pi}{2} * |1-y|), x, y \in [0,1]$$

The total area of the warp is [0,1] in both directions. The warp can also be described by a matrix:

$$\begin{vmatrix} 1,1 & 2,-1 & -2,1 \\ 0,1 & 0,0 & -1,-1 \\ 0,-1 & -1,-1, & 0,0 \end{vmatrix}$$

which describes 9 pairs of control points (vectors (x,y)). This matrix can be interpolated across a greater area than the 3x3 pixels it currently describes. An illustraded displacement matrix can be seen in Figure 2.15.

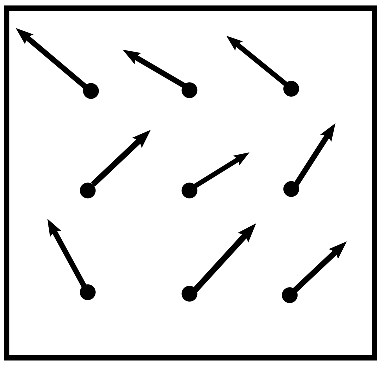

Figure 2.15: Control points visualized, not according to matrix above

Chapter 3

Image Processing on Cell Phones Today

Today the camera phones have resolution of 3 to 5 mega pixels. While focusing on a greater number of pixels, the market has not so much to offer when it comes to processing the pictures in a cell phone. The Sony Ericsson K800i with 3 mega pixel has a solution called *PhotoDJ*$^{\text{TM}}$where the user can fine-tune their photos by a number of effects,

- changing the contrast, brightness, auto levels, light balance,
- red eye removal,
- invert colors,
- add frames or clip art images,
- rotate, and
- effects like cartoon, frosted glass and painting

The competitor in the same class from Nokia, the N93 with a 3 mega pixel camera has the following image processing abilities (simply called "Edit"):

- Contrast and brightness control
- Red eye removal
- Inverted colors, sepia, black and white
- Add frames or clip art images
- Rotate, crop, resize
- Cartoon effect
- Sharpen or blur the image

The effects are almost the same but the Nokia cell phone having one more advanced effect, sharpen/blur. As complement the K800i has a Java application for warping faces.

Chapter 4

Engine and Filter Design

As previously stated, this thesis designs and implements a filter engine with a set of filters. The design was done by first implementing a simple engine with one per-pixel filter, tying the decoder, filter and encoder together. It was then extended with more advanced filters and different abilities for caching image data in the engine.

4.1 Overview

The engine design was based on the idea of a filter chain, given in the specification of this thesis. A filter chain can be seen as an assembly line in a car factory, where the decoder supplies the raw material. Each filter is a station on the line, modifying the image in a specified way according to the filter rules. At the end of the line the encoder stores the result on permanent memory.

To extract and store image data, a decoder and an encoder was given prior the design phase. The encoder and decoder do not work with JPEG as their source and target, but with the RAW file format. Their APIs are very similar to the JPEG ditto. The engine is intended to run with a JPEG decoder/encoder in the future, and is therefore designed and optimized with the JPEG decoder/encoder in mind.

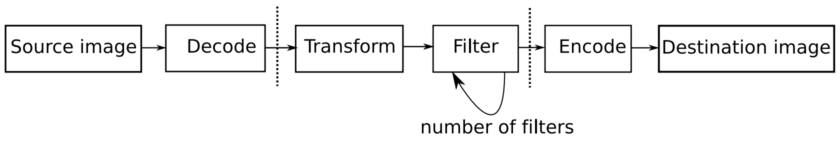

Figure 4.1: An overview of the data flow through the engine.

4.1.1 Specification

A number of requirements and restriction was set on the design from the specification at first.

– RAM (heap) usage must not exceed 500kB.

– The engine should be result driven meaning that the position of the output pixels decide which input pixels to process.

- A possibility to chain (almost) any number of filters.
- Efficiency of code should be taken into the design, e.g., no floating point data should be used.
- Data should be fetched and stored in an optimized way, from the JPEG encoder/decoder point of view.

4.1.2 Hardware limitations and restrictions

The ARM9 CPU does not support floating point operations in hardware [4] other than software emulated floating point support, which is slow. Fixed point math is used to achieve floating point precision.

4.2 Engine design

4.2.1 Memory concerns

To be able to process images sizing 15 MB (and possibly larger), the engine has to work on smaller parts at a time. Due to the construction of JPEG, storing data in 16 x 16 pixel blocks, the size was set to the whole width of the image and height of 16 rows. In the case of a typical 5 mega pixel image this would result in: $2500 \times 3 \times 16$ bytes, (width $\times$ number of color channels $\times$ height), in total 97.5 kB. To be able to chain any number of filters, and at the same time be memory efficient, the engine needs to have at least two internal buffers of this size. To reuse the allocated space the two buffers are used as source and target depending on the number of filters in the chain.

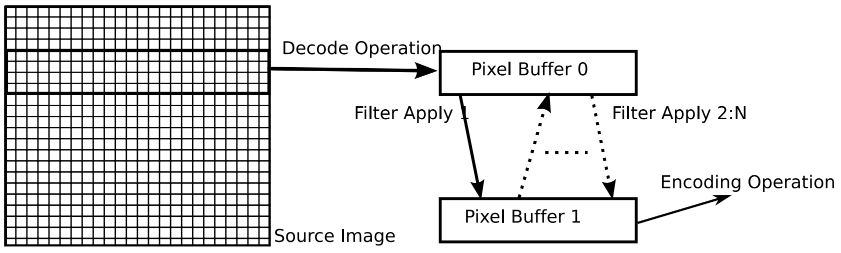

Figure 4.2: Pixel buffers changing role between source and target, depending on which filter in chain that is applied.

4.2.2 Decoder and Encoder API

The decoder is used to extract specific parts of an image, and load it to the engine buffers. The encoder stores the result buffers to a target file. The essential functions for the decoder and encoder are:

- The decoders Render function that renders a specific area of the source image to a buffer.
- The encoders Encode function that encodes the result buffer to a file for permanent storage.

4.2.3 Engine API

The engine has a small API because most of the complexity lies in the filters. The engine is responsible for managing the filters and their order. For example a transformation filter has to be run first in the chain (explained further down). The engine stores the set of filters in a list, by holding a reference to the filter.

Engine API

1. Create engine: Allocates the pixel buffers in Figure 4.2, creates the filter list, and ties the decoder and encoder to the engine.
2. Destroy engine: Deallocates the resources, leaving the decoder and encoder intact.
3. Add filter: Adds a filter to the engine. The filters are added in FIFO order, the first added filter is run first.
4. Delete filter: Deletes a filter from the engine, keeps the filter intact.
5. Run engine: Processes the filter chain and applies them to the input image. A deeper look is given in the next section.
6. Swap buffers: To reuse the pixel buffers in an optimized way this function switches the pointers internally, giving the filters the possibility to use the same reference to output independently where they are in the chain.

4.2.4 In depth: Run Engine

This being the heart of the engine, it is given a more through out explanation. Run engine does the following:

1. Queries the decoder for the size of the source image.
2. Queries the whole filter list for the smallest common output size given the height of the input buffer. Kernel filters will for example have a smaller output than input.
3. Calculate the number of iterations needed to process the image.
4. Start the loop for the sliding window over the image:
   (a) Render a part of the image as source data into one of the pixel buffers.
   (b) If there is a transformation filter in the chain, calculate the size of its buffer needed to process output in this pass, and if needed, render it to a separate buffer.
   (c) Start applying the filters in the chain (described by a list) on the current buffer:
   (d) If this is the first or last pass and there is a kernel filter in the chain, the filter is told to "invent" the pixel rows needed to process the first or last line in the source image.
   (e) Swap the source and target buffers pointers.
   (f) Encode the output buffer for this pass.
   (g) Done sliding window loop.
5. Done.

4.2.5 Engine Setup

To setup an engine and process an image with filters, a setup is done in by the following steps:

- Create handles and call decoder, encoder and engine create functions with the handle as argument.
- Create the desired filters by calling their constructors.
- Add filters to the engine via the engine API.
- Call the run engine function.
- Destroy filters using their handles.
- Destroy decoder, encoder and engine using their handles.
- Free other resources.

For a code sample of this see Appendix A.

4.3 Filter design

4.3.1 Filter API

All filters follow a common interface where the essential functions are: apply, query and set parameters.

Apply

Apply is called once for each pass of the sliding window over the image. This function processes the source image data and produces output data to be either written to permanent or as input to the next filter in the chain.

Query

Query is used by the engine to calculate the total area of an engine pass. For a per-pixel based filter the input size is the same as the output size. This is not the case for kernel filters, depending on the size of the kernel.

Set parameters

One filter can have different effects collected under the same filter. For example a color conversion filter can use either a lookup table or a constant function to alter the colors. This behavior is controlled by the set parameters function. An additional data pointer can supply arbitrary data to the filter, for example a kernel.

4.3.2 Filter classes

The image processing filters can be divided into three general types.

– Spatially independent: per pixel filter.

– Spatially dependent, constant: kernel filter.

– Spatially dependent, varying: transformation/warp filter.

Spatially independent

The per-pixel filters are not dependent on where they are applied which makes them spatially independent. This is ideal for processing large amounts of data because no overhead in pixel positioning has to be done.

Spatially dependent

There are two subclasses of this filter, constant or varying.

Kernel filter belongs to the *constant* class when it applies its convolution to the whole image. It is spatially dependent because it needs to know the surrounding pixels to produce an output pixel. This introduces a problem to the design when the image is processed by a sliding window. For example, when given for example 16 rows of data, a 3x3 kernel will produce 14 rows of output (see explanation in previous chapters). To mend this problem, the engine queries the filter chain before applying itself. The filters are given an area and responds with an output area. This is done through the whole chain and the result will tell the engine how much to increase the sliding window position for each pass. This will lead to some duplication of processed data in the case of a kernel filter applied after a per-pixel filter; this because of the input data to the kernel filter will be repeated. To mend this, kernel filters should be applied before per pixel filters.

The transformation filter belongs to the *varying* class where the area needed depends on the position of the sliding window. The transformation filter uses a separate buffer for its source data; this because the area covered by the sliding window does not always hold the needed source pixels.

Transformations are always run first of all filters, see conceptual design Figure 4.1. This is due to the complex process of backtracking output pixels when run later in the chain, for example a blur followed by a transformation. If this was to be applied, the transformation had first been forced to blur its source if it not was covered by the sliding window.

This result in the following descending apply-priority: Transformation, Kernel and lastly Per-Pixel filters.

4.3.3 Non trivial filter functions

The *apply* function of the warp filter has a helper function to assist the calculations of new pixel positions, defining the transformation. This function works independently of the size of the warping area, to be able to transform any arbitrary resolution of the source image. This done by converting the pixel positions to a relative coordinate system, $x, y \in [0, 1]$, and calculate an offset depending on the position in the warp area. For example, the area to be warped is 100 by 100 pixels in size and starts at $x = 400$ and $y = 150$. The position $x, y = 412, 192$ is translated to a position in the

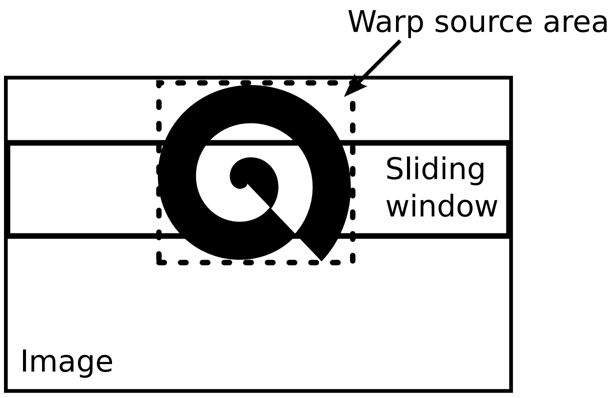

Figure 4.3: Image with warp transform covering a larger area than the sliding window

warp rectangle, $x', y' = 0.12, 0.42$. This position could for example results in an offset $x_{\text{offset}}, y_{\text{offset}} = -0.09, 0.12$, relative to size of the warp area. When translated back to image coordinates: $x_{\text{warped}} = 403$, $y_{\text{warped}} = 204$.

The warp filter needs to calculate the size of its source buffer depending on the size and position of the sliding window, as illustrated in Figure 4.3. The function *QueryRectangle* was designed to solve this problem. It calculates the maximum and minimum of the source pixels position, by running the warping function on the source positions, resulting in a rectangle. The function lets the warp filter request a minimal source area and keep the memory consumption on a low.

The kernel filter needs for example a method when fetching pixels outside its source buffer, a problem with a solution presented earlier. To solve this, the *GetPixelAt* function is designed. The function will emulate an extension to the buffer area by copying pixel data to a surrounding frame around the buffer, making it possible to fetch data outside the buffer for the kernel filter.

4.3.4 Optimization

As stated earlier, memory is limited, and the CPU does not do floating point in hardware. The CPU supports nor SIMD instructions (single-instruction, multiple-data, [18].

Fixed point math

Because the ARM CPU lacks FPU (Floating Point Unit), fixed point math can be used to calculate floating point operations with integer arithmetic. For example:

$$\frac{34.852}{5.556} = 6.27285817$$

(rounded to 8 decimals) is equivalent to

$$\frac{34852}{5556} = 6.27285817$$

where the terms are scaled with a factor of 1000. A CPU works with the base of 2 rather than base 10 as we humans often do. When the input is given as in this case as integers in the range of 1 to 256, the terms are first enlarged by a factor, for example

512, 2^9. This would result in $[1, 256]$ becoming $[1 \times 512, 256 \times 512] = [512, 131072]$. This operation can be done very quickly using the bit shift operator, in C denoted with $<<$ for left shift. Left shifting all scalars 9 steps would give the interval $[512, 131072]$. Scaling with left shift is very fast, on ARM done in one clock cycle. When computing filter operations, the data is held in fixed point representation until it is to be stored. The right shift operator is used to scale back to integer representation, in C denoted with $>>$. Conversion between floating point representation to fixed point is also possible via (in C):

```
int fixedPointInteger = (int)(float_value * 512);
```

When conversion is done, all calculations can be done in the following way, $F0$ and $F1$ denoting the terms, left shifted with 9, shamt denotes shift amount:
Addition: $F0 + F1$
Subtraction: $F0 - F1$
Multiplication: $(F0 \times F1) >> shamt$
Division: $\frac{(F0<<shamt)}{F1}$
Division with 2^n is done by right shifting, for example: $\frac{f0}{2}$: $F0 >> 1$
$\frac{f0}{4}$: $F0 >> 3$.

ARM optimization

To optimize loops, which are used for traversing the image data, branch tests in for and while-loops should be done by comparing with zero rather than a constant, counting backwards. This is not unique to ARM, but on the ARM processor, when an arithmetic operation is done, a flag is set if the answer is zero, hence, the comparison with zero takes no extra operations at all.

An other optimization is the *byte alignment*, where data fetches from memory should be done in sizes modulus eight, for example instead of fetching only one pixel, 3 bytes, it is more efficient to fetch 8 pixels, 24 bytes ($24 \mod 8 = 0$), because of the way ARM fetches data.

Addressing data in arrays should be done by a temporary index pointer rather than an offset to start of data, this because the ARM holds an auto increment register for the data pointer, saving 2 clock cycles per iteration.

The fixed point operations is very well supported by the ARM CPU because of the ability to do an arithmetic operation plus an arbitrary amount of bit shift single clock cycle.

The ARM optimizations have been taken from "ARM System Developer's Guide", [1].

Chapter 5

Results

As a result of the design, an engine with a set of filters has been implemented. The engine has been tested mostly on the PC platform, but for speed benchmarks a set of tests were done on a cell phone, as described in the benchmark section. Because the implementation does not include a GUI to control the engine, no actual screen shots of a run can be presented. The output log from the engine, in two test runs on the cell phone can be found in Appendix B.

5.1 Implemented filters

The following filters were implemented:

– Per pixel filters like gray scale, invert, saturation, color mapping, and posterizing.

– Blend between pixel buffers (currently limited to two sources).

– Kernel filter able of running arbitrary kernels of M x N size and.

– Warp filter.

These filters can be combined in a numerous ways giving the user the possibility to make new effects with these simple tools. Even while for example a kernel filter is of low or no interest to an end-user, a combination of filters can be done to achieve an effect. This is where the design idea of chaining filters is being used. For example a kernel filter, posterizing and a blend operation results in a sketch-like image.

For examples of filter effects and their output images see Appendix C.

5.2 Benchmarks

The goal section stated that speed and memory consumption was crucial to the implementation. To present the efficiency of the implementation, two types of benchmarks have been done, one with memory consumption aspect and one with running time ditto.

The test image measured $2048 * 1536$ pixels in RGB, the same size as a captured image from a Sony Ericsson k800i. The test was run on the same phone model, and compared with the internal *Photo DJ* in the CPU-time benchmark. The times can not be compared directly because of the time needed for *Photo DJ* to decode and encode the image from and to JPEG data. Both test used the memory card as storage of the source and target image. The thesis implementation works in raw RGB data that needs very little processing time when fetched from permanent memory compared to JPEG. Because of this the time was estimated for the encoding and decoding to give a fair comparison.

5.2.1 Memory consumption

A 3.2 mega pixel image was loaded to the engine. All filters were run first alone then combined. The resolution of 3.2 mega pixel was chosen to be able to compare it with the existing implementation on the Sony Ericsson k800i cell phone.

Filter	Maximum heap allocated memory at a time
Invert	192kB
3x3 Kernel	192kB
Invert + 3x3 kernel	192kB
200x200 Warp	309kB
All combined	309kB

While the two first filters combined does not yield an increased amount of memory, the warp filter needs its own buffer, therefore a higher consumption. Note that the stack allocated amount is not included in these numbers above. This benchmark was carried out in a PC environment.

5.2.2 CPU time

Filter	Processing time
Invert	0.95 sec
3x3 Kernel	11.9 sec

These numbers represent the time taken by the engine, not including the decoding and encoding process. Encode and decode would take about 4 seconds each with the JPEG encoder/decoder. This would result in about 9 seconds for the invert filter and 20 seconds for the kernel ditto. This benchmark was carried out on the Sony Ericsson k800i. The warp filter is not part of the results because of its floating point implementation.

To compare with already existing effects the Sony Ericsson k800i takes about 8 seconds to apply an invert filter on an equally sized image. This shows that the design and implementation is not far away from the optimized implemented one in the "Photo DJ" in the phone.

Chapter 6

Conclusion and Future Work

In this thesis the design and implementation of an image processing engine and a set of filters, able of running in an environment of a cell phone has been presented. The result is a working filter engine able to do a lot of different effects. This implementation shows that it is possible and not too complicated to do image processing with limited amount of memory.

While memory limits has been conquered the CPU still constitutes the current limit of how advanced effects can be.

The result is not a finished product, ready to be launched in the next series of phones, rather a proof-of-concept. The implementation does meet the requirements stated in the goal section.

For me as a student the work with this thesis has given me a tremendous knowledge in image processing, which was a new subject to me prior the start of this thesis.

6.1 Future work

The future phones will have faster hardware, most certainly a co-processor for graphics, GPU, which will give the phone the possibility of doing advanced effects in a short period of time. Knowing this, it would have been interesting to implement some of the filters in Open GL shading language, and compare benchmarks between the two implementations.

For the current implementation there is some important work to be done, most concerning calculation speed.

– For greater speed, the use of fixed-point operations, especially in the warp filter should be implemented (which currently uses floats).

– More exploiting of optimized ARM assembler instructions, for example the multiple load instruction when fetching pixel data to the CPUs internal registers.

– The warp filter's query function calculates all the offset points to extract the extreme points for the buffer. These results could have been cached to speed up the apply process, to some limit where the stored results would not allocate to much heap memory.

– Faster (though less accurate) log, cosine/sine, and square-root functions should be implemented and used.

On the filter side a lot more interesting filters would be interesting to implement, especially an advanced painterly algorithm-based stroke filter as referenced earlier.

In a wider aspect it would be necessary to implement GUI to run the engine on the phone. Today the call to *run engine* was connected to a certain key combination, supplying no possibility of interaction with the user. The GUI was on the other hand outside of the scope of the thesis from the beginning.

References

[1] D. Symes A. Sloss and C. Wright. *ARM System Developers Guide*. Morgan Kaufman, 2004.

[2] Adobe. Adobe rgb (1998) color image encoding. http://www.adobe.com/digitalimag/adobergb.html (visited 2007-05-09).

[3] F. Drago K. Myskowski T. Annen and N. Chiba. Adaptive logarithmic mapping for displaying high contrast scenes. Technical report, Iwate University, Morioka, Japan, and MPI Informatik, Saarbucken, Germany, The Eurographics Association and Blackwell Publishers, 2003.

[4] ARM. Arm9 family. http://www.arm.com/products/CPUs/ARM920T.html (visited 2007-05-09).

[5] CGSD. Gamma correction explained. http://www.cgsd.com/papers/gamma_intro.html (visited 2007-05-09).

[6] 3GPP NTT DoCoMo. View on 3g evolution and requirements- 3g longterm volution scenario: Super 3g. http://www.3gpp.org/ftp/workshop/Archive/2004_11_RAN_Future_Evo/Report/%REV_WS_Abstracts.pdf (visited 2007-05-14).

[7] Engadget. Live from cebit. http://www.engadget.com/2006/03/09/live-from-cebit-samsungs-sch-b600-10%-megapixel-cameraphone (visited 2007-05-09).

[8] Ericsson. The emp story. http://www.ericsson.com/ericsson/corpinfo/publications/review/2005_01/f%iles/2005013.pdf (visited 2007-05-09).

[9] Sony Ericsson. Developers' guidelines java me cldc (midp 2). http://developer.sonyericsson.com/getDocument.do?docId=65067 (visited 2007-05-09).

[10] Tech FAQ. What is hscsd? http://www.tech-faq.com/hscsd-high-speed-circuit-switched-data.shtml (visited 2007-05-09).

[11] Paolo Favaro. Depth from focus - defocus. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FAVARO1/dfdtuto%rial.html (visited 2007-05-09).

[12] Aviator FX. The aviator special effects, behind the scenes. http://www.aviatorvfx.com (visited 2007-05-10).

[13] Portable Network Graphics. An open, extensible image format with lossless compression. http://www.libpng.org/pub/png/ (visited 2007-05-25).

[14] E. Hamilton. Jpeg file interchange format. Technical report, C-Cube Microsystems, September 1992.

[15] Aaron Hertzmann. A survey of stroke-based rendering 2003. `http://www.dgp.toronto.edu/~hertzman/sbr02`.

[16] Gernot Hoffmann. Cie color space. `http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf` (visited 2007-05-09).

[17] N. Goodnight R. Wang C. Woolley G. Humpherys. Interactive time-dependent tone mapping using programmable graphics hardware. Technical report, Dept. Comp. Sci., University of Virginia, 2003.

[18] Intel. Glossary. `http://www.intel.com/software/products/documentation/vlin/vtglossary_hh%/vtglossary.html` (visited 2007-05-14).

[19] M. Kopp. Efficient filter computation with symmetric matrix kernels. Technical report, Institute of Computer Graphics, Technical University of Vienna, 1994.

[20] Norman Koren. Tonal quality and dynamic range in digital cameras. `http://www.normankoren.com/digital_tonality.html` (visited 2007-05-15).

[21] L. Kovács and T. Sziáni. 2d multilayer painterly rendering with automatic focus extraction. Technical report, Dept. of Image Processing and Neurocomp, and Comp. and Automation Research Institute, University of Veszpre´m and Hungarian Academy of Sciences, 2006.

[22] You Me and everybody. Wikipedia.org. `http://www.wikipedia.org` (visited 2007-05-09).

[23] M. Stokes M. Anderson S. Chandrasekar R. Motta. A standard default color space for the internet - srgb. `http://www.w3.org/Graphics/Color/sRGB` (visited 2007-05-09).

[24] Widescreen Museum. Widescreen museum. `http://www.widescreenmuseum.com/oldcolor/technicolor2.htm` (visited 2007-05-10).

[25] Diego Nehab. Moment based painterly rendering 2006. `http://www.cs.princeton.edu/~diego/academic/phd/526/final.pdf`.

[26] Nokia. Nokia n95. `http://www.nokia.se/phones/n95/` (visited 2007-05-14).

[27] Nvidia. Nvidia goforce family product comparison. `http://www.nvidia.com/object/hh_compare.html` (visited 2007-05-08).

[28] Nvidia. Reviews & editorials. `http://www.nvidia.in/page/goforce_3d_4500_reviews.html` (visited 2007-05-09).

[29] Retrobrick. Motorola dynatac 8000x. `http://www.retrobrick.com/moto8000.html` (visited 2007-05-09).

[30] Samsung. Samsung introduces 3d game phones with vibration. `http://www.samsung.com/PressCenter/PressRelease/PressRelease.asp?seq=20%050406_0000110729` (visited 2007-05-09).

[31] Scalado. Scalado. `http://www.scalado.com` (visited 2007-05-08).

[32] New Scientist. 4g prototypes reach blistering speeds. `http://www.newscientist.com/article.ns?id=dn7943` (visited 2007-05-09).

[33] E. Reinhard M. Stark P. Shirley and J. Ferwerda. Photographic tone reproduction for digital images. *ACM Transactions on Graphics*, 21:267–276, 2002.

[34] Jr. T.G. Stockham. Image processing in processing in the context of a visual model. *Proc. IEEE*, 60:828–842, 1972.

[35] Gregory K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, April 1991, 1991.

Appendix A

Engine setup

This shows an example of an engine setup. Note that the final decoder could extract information of size and bytes per pixel from the file header instead of the user supplying them. The raw file format used does not support this feature.

```
//Function header skipped

// Initiate handles for encoder, decoder, engine, filters and kernel
MFX_Decoder_t *pDecoder;
MFX_Encoder_t *pEncoder;
MFX_Engine_t *pEngine = NULL;
MFX_filter_t *pF0 = NULL, *pF1 = NULL;
MFX_Rect_t srcRect;
MFX_Area_t dstArea;
MFX_Kernel_t *pKern = NULL;

int kernel[9] = { -1,  -1, -1,
                  -1,   8, -1,
                  -1,  -1, -1};
// Open a source file, UTF8 coded filename.
if (MFX_CreateDecoder(MFXCARDPATH L"test.raw", 1024, 768, 3, &pDecoder)
== MFX_FILE_ERROR)
{ // error handling omitted }
// Get information of soruce file
MFX_GetImageDimensions(pDecoder, &width, &height);

// The setup of the structs srcRect and destArea omitted here.

// Tell the decoder what size the source has.
MFX_SetupDecoder(pDecoder, srcRect, dstArea);

// Open an output file
if(MFX_CreateEncoder(MFXCARDPATH L"output.raw", width, height, 3, &pEncoder)
== MFX_FILE_ERROR)
{ // error handling omitted}
```

```
// Create some filters
pF0 = MFX_KernelFilter_Create();
pF1 = MFX_ColorFilter_Create();

// Use the set parameter function for the kernel filter
pKern = MFX_CreateKernel(kernel,3,3);
pF0->setParameters(pF0, "setkernel", pKern);
pF0->setParameters(pF0, "onechannel", NULL);

// Apply filters
MFX_RunEngine(pEngine);

// Done with engine, deregister the filters.
MFX_DeleteFilter(pF0, pEngine);
MFX_DeleteFilter(pF1, pEngine);

// Free resources held by filters
MFX_KernelFilter_Destroy( (MFX_KernelFilter_t*)pF0);
MFX_ColorFilter_Destroy( (MFX_ColorFilter_t*)pF1);

// Free rest of resources
MFX_DestroyEngine(&pEngine);

MFX_DestroyEncoder(&pEncoder);

MFX_DestroyDecoder(&pDecoder);

// All done!
```

Appendix B

Engine test run

Test run of two different setups of the engine on phone. First a 3x3 kernel filter is run; second run includes only an invert filter. Output from phone log (other phone messages omitted):

```
9657       [  MFXdebug ] Begin function: CreateDecoder
9657       [  MFXdebug ] End function:   CreateDecoder
9658       [  MFXdebug ] Begin function: GetImageDimensions
9658       [  MFXdebug ] End function:   GetImageDimensions
9658       [  MFXdebug ] Begin function: SetupDecoder
9658       [  MFXdebug ] End function:   SetupDecoder
9658       [  MFXdebug ] Begin function: CreateEncoder
9661       [  MFXdebug ] End function:   CreateEncoder
9661       [  MFXdebug ] Begin function: MFX_CreateEngine
9661       [  MFXdebug ] Begin function: GetImageDimensions
9661       [  MFXdebug ] End function:   GetImageDimensions
9662       [  MFXdebug ] End function:   MFX_CreateEngine
9662       [MFX_kernelf] Normalize with : 8
9662       [  MFXdebug ] Begin function: MFX_RunEngine
9662       [MFXRnEngine] Common minimal area is 2048 x 14 [w x h]
9696       [MFX_KF_setp] Invent startpixels
.
18770      [MFX_KF_setp] Invent stoppixels
18797      [  MFXdebug ] End function:   MFX_RunEngine
18797      [MFX_RunEng.] Total runningtime 36.537s.
18797             Time spent in decoder 22.390s time spent in encoder 2.295s
18797                   Time spent in filters 11.852s
18797      [  MFXdebug ] Begin function: MFX_DestroyEngine
18798      [  MFXdebug ] End function:   MFX_DestroyEngine
18798      [  MFXdebug ] Begin function: DestroyEncoder
18798      [  MFXdebug ] End function:   DestroyEncoder
18798      [  MFXdebug ] Begin function: DestroyDecoder
18798      [  MFXdebug ] End function:   DestroyDecoder
.
18798      [  MFXdebug ] Begin function: CreateDecoder
18799      [  MFXdebug ] End function:   CreateDecoder
```

```
18799     [  MFXdebug ] Begin function: GetImageDimensions
18799     [  MFXdebug ] End function:   GetImageDimensions
18799     [  MFXdebug ] Begin function: SetupDecoder
18799     [  MFXdebug ] End function:   SetupDecoder
18799     [  MFXdebug ] Begin function: CreateEncoder
18806     [  MFXdebug ] End function:   CreateEncoder
18806     [  MFXdebug ] Begin function: MFX_CreateEngine
18806     [  MFXdebug ] Begin function: GetImageDimensions
18806     [  MFXdebug ] End function:   GetImageDimensions
18807     [  MFXdebug ] End function:   MFX_CreateEngine
18807     [  MFXdebug ] Begin function: MFX_RunEngine
18807     [MFXRnEngine] Common minimal area is 2048 x 16 [w x h]
.
24602     [  MFXdebug ] End function:   MFX_RunEngine
24603     [MFX_RunEng.] Total runningtime 23.182s.
24603             Time spent in decoder 19.557s time spent in encoder 2.677s
24603                   Time spent in filters 0.948s
24603     [  MFXdebug ] Begin function: MFX_DestroyEngine
24604     [  MFXdebug ] End function:   MFX_DestroyEngine
24604     [  MFXdebug ] Begin function: DestroyEncoder
24604     [  MFXdebug ] End function:   DestroyEncoder
24604     [  MFXdebug ] Begin function: DestroyDecoder
24604     [  MFXdebug ] End function:   DestroyDecoder
```

Appendix C

Example images

Some example images with the filters applied:

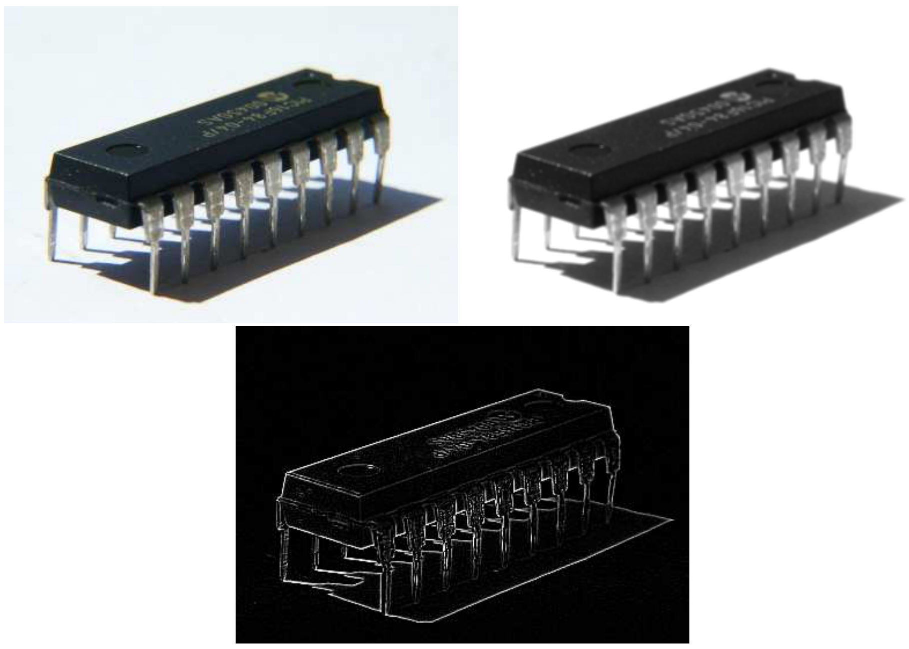

Figure C.1: Kernel Filters: Original, gray scaled plus 3x3 blurred, and edge detected (Laplacian).

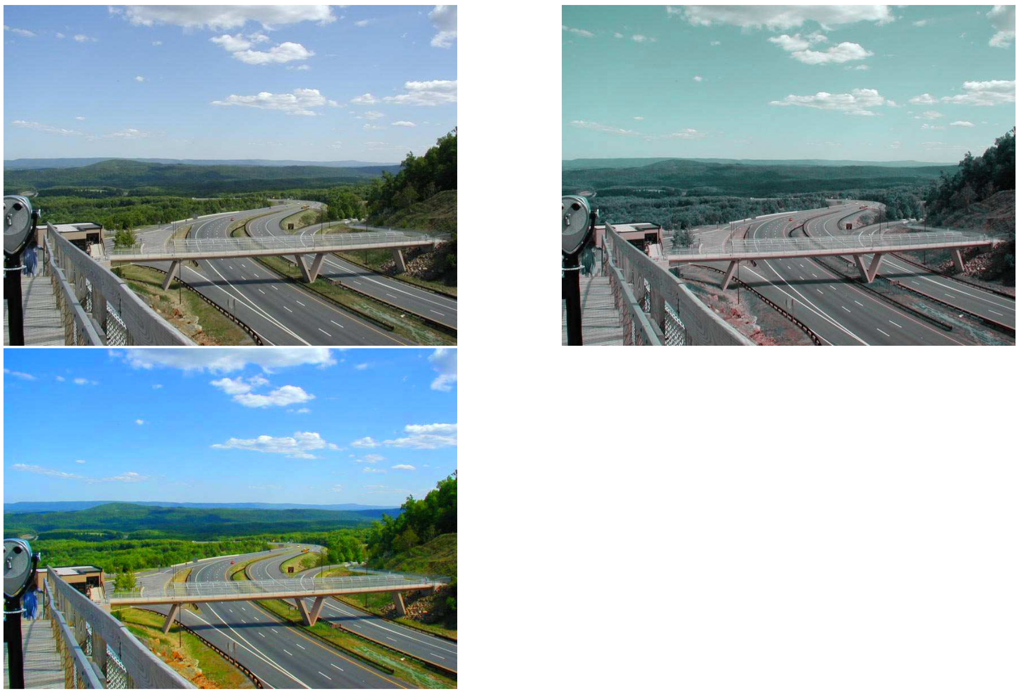

Figure C.2: Color Transformation: Original, Technicolor 2-strip (a like) and Technicolor 3-strip.

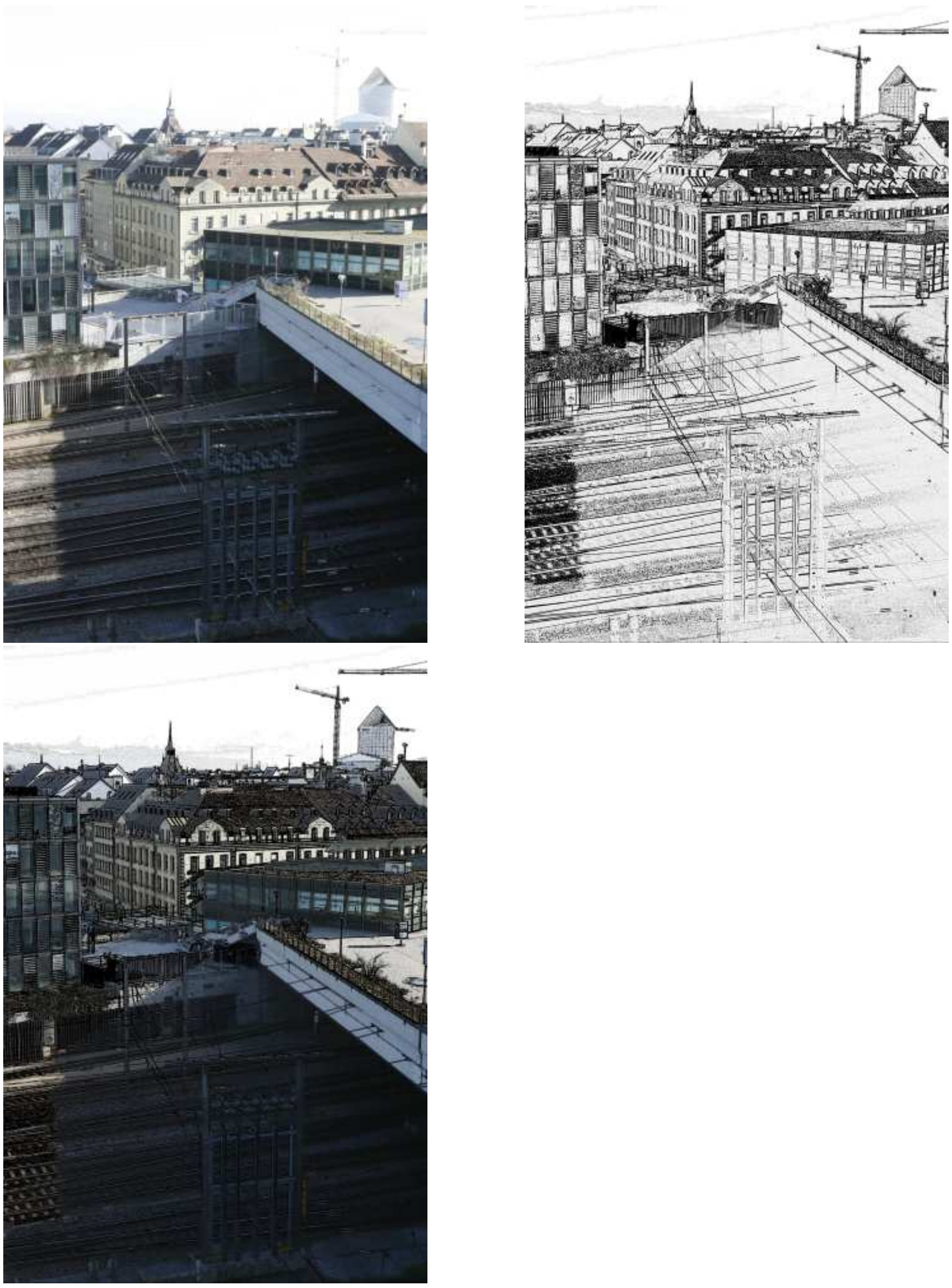

Figure C.3: Sketch Filter: Original, black and white version, original plus black and white.