The Hague University of Applied Sciences

Image acquisition and processing
Lab

Final Report

Author/Student:
Luca van Straaten (18073611)
Roderik Leijssen (15060292)

Instructor:
F. Theinert

November 20, 2022

Document version 2.0

Contents

0 Introduction

we work from the lab assignment [1]. They state:

> The students will work in groups of two and have to attend all lab-sessions in order to pass the course. During the lab-sessions, the students are asked to take images and write software to process them. Students are asked to bring their own laptops with an USB3.0 port. All assignments can be worked out on school-computers with the cameras supplied, but working independently on your own laptop is recommended. Students will receive a virtual machine (Virtual Box) with all software preinstalled on Ubuntu 22.04 LTS.

This report describes the exercises and how they were solved by the students.
The students used the virtual machine and project tamplate provided by the teacher.

1 Assignment 1
Setup

For this assignment we connected the Camera to the Virtual Machine. And to the project tempalte a case was addet to the "switch (key)" statement (see Listing 1). This case was used to take a picture with the camera. The picture was then saved in the folder from wich the program was run.

```
    case 's':
        cout << "Saving..." << endl;
        // save image using openCV API
        imwrite("blahai.png", image);
```

Listing 1: save image to file

Figure 1: Image taken with the camera of a blahai

We pointed the camera at an object and adjusted the apature and focus to get a good looking picture with a shutter time of 100ms. See image 1

2 Assignment 2
Object in front of Dark Background

For this assignment, we will be capturing a image wich we will also use for assignment 3 and 4. It will be of a model car in front of a dark background. The goal is to make a useful setup to acquire the image and solely adjust the exposuretime to come to a well exposed image [1].

The code in Listing 2 was used to take the image, or adjust the exposure time. The image was then saved in the folder from wich the program was run. for the full code see appendix A.

```
    case ' ':
        if (imgSave(image, "output.png")) {
            cout << "Image saved succesfully!" << endl;
        } else {
            cout << "Error saving file." << endl;
        }
        break;
        case ',':
        cam0.setExpoMs(--cfg.exposureMS);
        cout << "Exposure adjusted to " << cfg.exposureMS << endl;
        break;
        case '.':
        cam0.setExpoMs(++cfg.exposureMS);
        cout << "Exposure adjusted to " << cfg.exposureMS << endl;
        break;
        case '[':
        cam0.setExpoMs(cfg.exposureMS -= 10);
        cout << "Exposure adjusted to " << cfg.exposureMS << endl;
        break;
        case ']':
        cam0.setExpoMs(cfg.exposureMS += 10);
        cout << "Exposure adjusted to " << cfg.exposureMS << endl;
        break;
```

Listing 2: save image to file

2.a Object with dark background

Figure 2: Object in front of Dark Background

The image above is the image we took of the model car, it has a matelic gold paint with black stripes across. It was challenging to get the car well exposed, because the metallic paint is reflective. So we needed even light positioned in a way that the reflections would not go into the camera. Enough light was needed for the black parts of the car to not be ender exposed. We placed the object away from the background so we could make shine the light only on the car and not the background. The layout of the setup is shown in section 2.c.

2.b Optimal exposure

We got the best result using a 420 ms exposure time. This is the time the camera takes to gather light on the sensor. The image is shown in figure 2. The image is well exposed and the background is dark. The car is well visible and the details are clear. The image is not overexposed and the background is black but not saturated.

2.c Sketch of setup

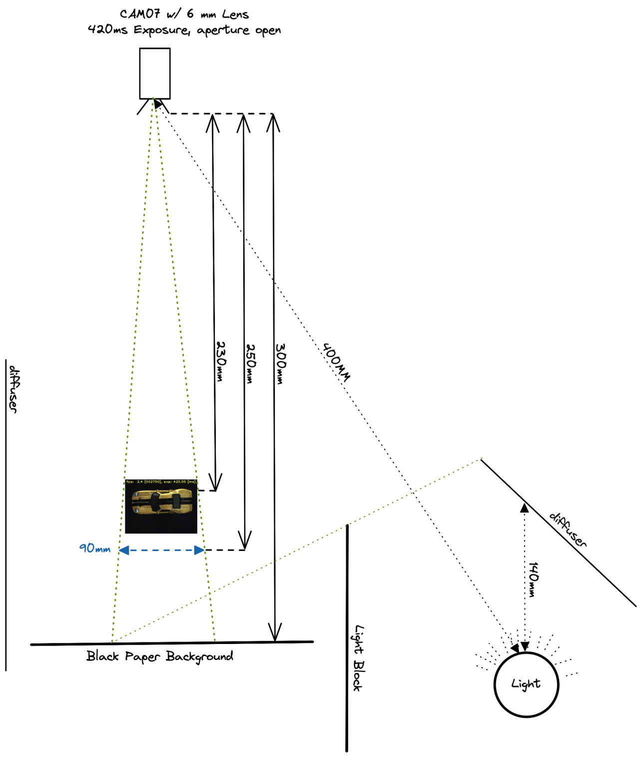

Figure 3: Sketch of setup

2.d Calculation of 'angle of view'

To calculate the angle of view α we need the following information:

- Sensor size [mm] $d = 8.89mm$
- focal length $f = 6mm$

And we used the following formula:

$$\alpha = \frac{180}{\pi} \cdot 2 \arctan \frac{d}{2 \cdot f} = \frac{180}{\pi} \cdot 2 \arctan \frac{8.89}{2 \cdot 6} = 73.1 \text{ degrees}$$

3 Assignment 3
Moving Object

Take an image of a considerably fast moving object (rotating disk) without any motion-blur and without reflection from any light-sources. You will not be able to synchronize the camera, so find a solution which will not need any synchronization. Make a sketch of the required setup first, discuss multiple solutions in the group. [1]

```
// capture 50 frames
if (capt50 == true) {
    capt50 = false;
    for(int i = 0; i < 50; i++) {
        cam0.captureFrame(&image);
        // save image with frame number
        imwrite("../capt/img" + to_string(i) + ".png", image);
    }
}
```

Listing 3: save image to file

3.a Optimal exposure

We use a strobe light to expose this image. The strobe frequency is not important but should be sufficiently slow to make it impossible for two exposures to occur in a single frame, and it should also be slow so that the capacitors inside the strobe enough time to charge to give the lights its maximum brightness. We took 50 images, and saved them to the file system. The first image which looks like 4 was handpicked and the other images ware ignored.

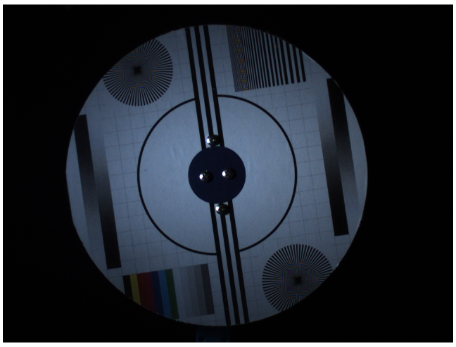

Figure 4: Image of moving object

3.b Sketch of setup

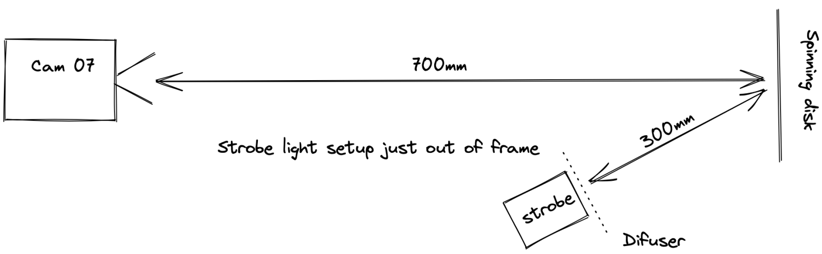

Figure 5: Sketch of setup

3.c Calculation of ‘angle of view’

We use the same camera and lens as assignment two, So the angle of view will be identical as calculated in section 2.d.

4 Assignment 4
Salt and Pepper Noise

4.a Manipulate own image from assignment 2

We applyed a salt and pepper noise filter to the image from assignment 2. The image is shown in figure 6.

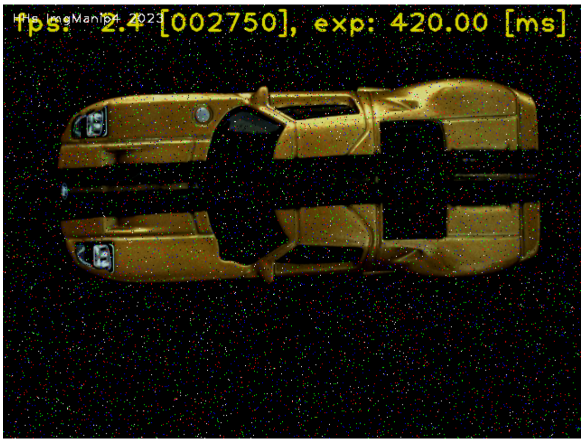

Figure 6: Image with salt and pepper noise

4.b Write C/C++ code for Brightness correction

A Brightness correction filter was written and can be found in appendix B. The filter function is shown in listing 4.

```

void gammaCorrection(Mat* input, Mat* output, float gamma) {
    Size s = input->size();
    long h, w;
    float sum;

    std::cout << "Correcting gamma" << std::endl;
    uint8_t out[s.height][s.width];
    uint8_t lut[256];

    for (int i = 0; i < 256; i++) { //create a lookup table, with the gamma correction curve.
        lut[i] = saturate_cast<uint8_t>(pow((float)(i / 255.0), gamma) * 255.0f); //saturate cast negative values to 0, and higher values to 255 (uint8_t or unsigned char)
        //std::cout << unsigned(lut[i]) << " "; //print the function for testing. cout prints uint8_t as chars so we cast it.
    }


    for (w=0; w<s.width; w++){ //loop over the image,
        for(h=0; h<s.height; h++){
            sum = lut[(input->at<uint8_t>(h,w))]; //the original output value will be scaled to the value in the LUT.
            out[h][w]=(uint8_t)sum;
        }
    }
    std::memcpy(output->data, out, s.height*s.width*sizeof(uint8_t)); //copy our standard 2D array to a new buffer that OpenCV understands
}
```

Listing 4: Brightness correction

4.c Write C/C++ code for ‘Salt and Pepper Noise’ correction

A ‘Salt and Pepper Noise’ correction filter was written and can be found in appendix B. The filter function is shown in listing 5.

```
void filter(Mat* input, Mat* result) {
    Size s = input->size();
    long h, w;
    long sum;

    //std::cout << "Input type was : " << input->type() << std::endl;
    uint8_t out[s.height][s.width];

    std::cout << s.height << " " << s.width << std::endl;

    for (w=0; w<s.width; w++){ //loop over the image,
        for(h=0; h<s.height; h++){
            sum = 0;
            std::vector<uint8_t> median;

            for (int _x = -1; _x < 2; ++_x) //for every pixel, loop over every pixel in a 3x3 kernel
            {
                for (int _y = -1; _y < 2; ++_y)
                {
                    int idx_y = h + _y; //sum the incrementor with the kernel's, so we can identify borders
                    int idx_x = w + _x;

                    if (idx_x < 0 || idx_x > s.width) //the kernel goes outside of the image, therefore break and ignore that pixel.
                        break;

                    if (idx_y < 0 || idx_y > s.height)
                        break;

                    median.push_back(input->at<uint8_t>(idx_y,idx_x)); // Add all the pixels from the kernel into a vector
                }
            }
            std::sort(std::begin(median), std::end(median)); //sort all pixel values from high to low

            for (auto it = median.begin(); it != median.end(); ++it) {
                sum = median.at(median.size()/2); //The pixel at the y,x coordinate is now the median from our 3x3 sliding window
            }
            out[h][w]=(uint8_t)sum;
        }
    }
    // this did not work, since the out array was never copied to memory, causing image data pointing to nowhere!
    //result = Mat(s.height, s.width, CV_8U, out);
    // Instead, the data from out needs to be copied directly to Mat result with the correct size
    std::memcpy(result->data, out, s.height*s.width*sizeof(uint8_t));
}
```

Listing 5: Noise correction filter

The result of output image is shown in figure 7.

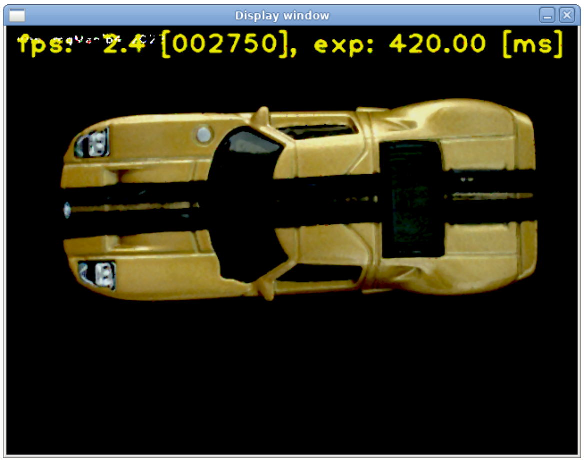

Figure 7: Filtered image

5 Assignment 5
Convolution

We wrote a convolutie 5x5 kernel with all 1's. The kernel put over the image from lab 2 is shown in figure 8.

Figure 8: 5x5 kernel over image from lab 2

The code we used to make the kernel is shown in listing 6. and the full code can be found in appendix C.

```
//int* from original assignment replaced with pointer to the kernel instead, because that makes more sense
void convolve5 (Mat* inputImg, Mat* outImg, int (*kernel5)[5][5]) {
    //I assume the mat is CV_8UC1 since I want to process BGR channels individually
    Size s = inputImg -> size(); // get size of image
    const uint8_t kernel_size = 5; // todo: replace with sizeof
    uint8_t out[s.height][s.width]; // create output array
    int x ,y, h, w, i, j, sum; // declare variables

    std::cout << "Input type was : " << inputImg->type() << std::endl; //debug

    for (h=0; h<s.height; h++) { // for each row
        for(w=0; w<s.width; w++){ // for each column
            sum = 0; // reset sum
            for(i=0 ;i<kernel_size; i++ ){ // for each kernel row
                for(j=0; j<kernel_size; j++){ // for each kernel column
                    y=h-i+1; x=w-j+1; // calculate the position of the pixel in the image
                    // if the pixel is outside the image, set it to the border
                    if(y<0) y=0;
                    if(y>s.height-2) y=s.height-2;
                    if(x<0) x=0;
                    if(x>s.width-2) x=s.width-2;
                    sum += ((*kernel5)[i][j]) * inputImg->at<uint8_t>(y,x); // add the product of the kernel and the pixel to the sum
                }
            }
            sum /= kernel_size*kernel_size; //divide the result of the pixel by 5^2
            if(sum<0) sum=0; // if the result is negative, set it to 0
            if(sum>255) sum=255; // if the result is greater than 255, set it to 255
            std::memcpy(outImg->data, out, s.height*s.width*sizeof(uint8_t)); // copy the result to the output array
            out[h][w]=(uint8_t)sum; // set the result to the output array
        }
    }
}
```

Listing 6: 5x5 kernel

6 Assignment 6
Demosaicing Filter

6.a Write C/C++ code for capturing a raw image

```
// capture raw image
imshow(camName, image);
if (captRAW == true) {
    captRAW = false;
    // save raw image
    cfg.camMode = CAM_MODE_RAW;
    cam0.captureFrame(&image);
    imwrite("../capt/RAW.png", image);

    // save color image
    cfg.camMode = CAM_MODE_COL;
    cam0.captureFrame(&image);
    imwrite("../capt/COL.png", image);
}
```

Listing 7: save image to file

the output of the code is shown in figure 9.

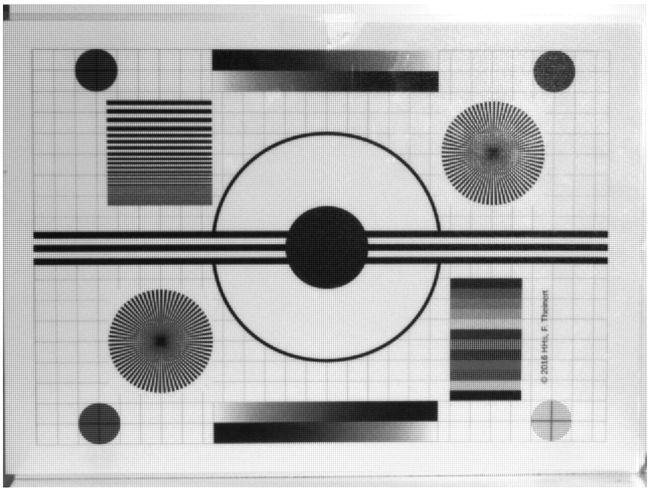

Figure 9: test image

6.b Convert this grayscale image to a color image by implementing a function to recover the actual colors

The function prototype should look as follows:

```
void deBayer(Mat *rawImg, Mat *outImg);
```

Listing 8: function prototype

To convert the RAW image, we first splitt the image into 3 channels, one for each color. Then we interpolate the missing values in the channels. The result is a color image. The function for splitting is shown in listing 9.

```

for(row=0; row<rawImg->rows; row++){  //todo: make this evaluation smaller to increase speed
    for(col=0; col<rawImg->cols; col++){
        if (row % 2 == 0 && col % 2 == 0) //odd row, odd column
            outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] = rawImg->at<uint8_t>(row,col);
        if (row % 2 == 0 && col % 2 == 1) //odd row, even column
            outImgMID->at<Vec3b>(row, col).val[BGR_RED] = rawImg->at<uint8_t>(row,col);
        if (row % 2 == 1 && col % 2 == 0) //even row, odd column
            outImgMID->at<Vec3b>(row, col).val[BGR_BLUE] = rawImg->at<uint8_t>(row,col);
        if (row % 2 == 1 && col % 2 == 1) //even row, even column
            outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] = rawImg->at<uint8_t>(row,col);
```

```
    }
}
```

Listing 9: splitt image

And the function for interpolating is shown in listing 10, ware as a exaple the green channel is interpolated. This function interpolates the missing values in the channels, by using the values of the neighboring pixels. The result is a color image.

```

// Interpolate green channel by taking the value of the nearest neighbour that is green
std::cout << "Interpolating green channel" << std::endl;

for(row=0; row<outImgMID->rows; row++){
    for(col=0; col<outImgMID->cols; col++){
        if (outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] == 0){
            if (col > 0 && outImgMID->at<Vec3b>(row, col-1).val[BGR_GREEN] != 0)
                outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row, col-1).val[BGR_GREEN];
            else if (col < outImgMID->cols-1 && outImgMID->at<Vec3b>(row, col+1).val[BGR_GREEN] != 0)
                outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row, col+1).val[BGR_GREEN];
            else if (row > 0 && outImgMID->at<Vec3b>(row-1, col).val[BGR_GREEN] != 0)
                outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row-1, col).val[BGR_GREEN];
            else if (row < outImgMID->rows-1 && outImgMID->at<Vec3b>(row+1, col).val[BGR_GREEN] != 0)
                outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row+1, col).val[BGR_GREEN];
        }
    }
}
```

Listing 10: interpolate missing values

This gives the output image shown in figure 10. Which is different from the color image shown in figure 11.

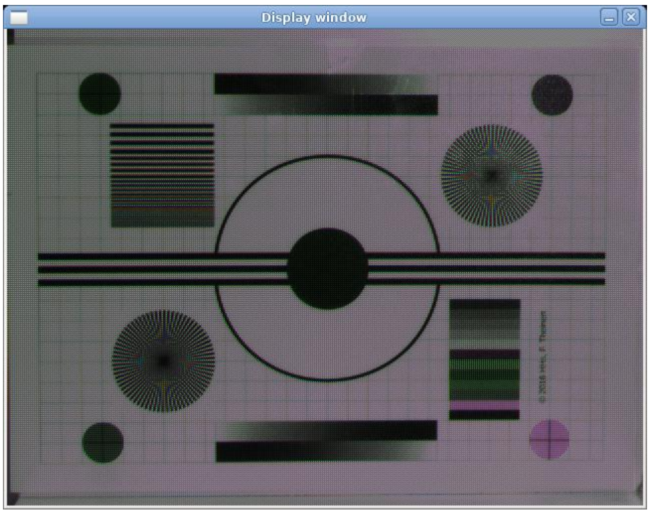

Figure 10: Output image

For the full code see appendix D.

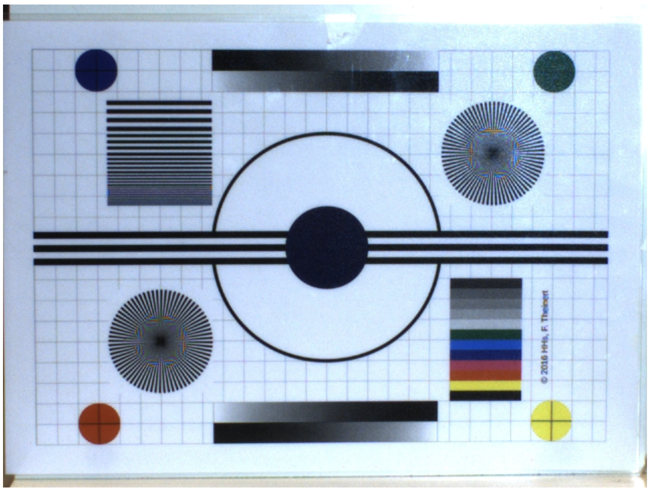

Figure 11: Original image

References

Theinert, F. (n.d.). *Reader image acquisition and processing.* The Hague University of Applied Sciences.

A Appendix A

```
/*
 hhs_cam.cpp

 Get frames from DaHeng USB3.0 camera and
 display them in window
 Created on: 2022 / 07
 Author: Fidelis Theinert
 Reading DaHeng cameras with OpenCV 4.5
 Version 1.0
 */

#include <iostream>
#include <string>
#include <stdio.h>

#include <opencv.hpp>
#include <highgui.hpp>

#include "dh0.h"

// Namespace for using cout.
using namespace std;

// Namespace for OpenCV
using namespace cv;


/******************************************************************************
 DEFINITIONS AND MACROS

 ****************************************************************************
*/

// blue green red is order used in openCV
#define COL_BLUE                    0
#define COL_GREEN                   1
#define COL_RED                     2

#define COLMODE_COL                 0
#define COLMODE_GREY                1

/******************************************************************************
 PROTOTYPES OF NOT EXPORTED FUNCTIONS

 ****************************************************************************
*/

//int imgSave(int cnt, Mat img, string fname);
int Config(int argc, char **argv, struct ImgConf *camCfg);
int PrintHelp(void);
void InitWindows(string);
void ConvertGrey(Mat*);

/****************************************************************************
```

```
**

 PROTOTYPES OF EXPORTED FUNCTIONS

 ***************************************************************************
*/

/***************************************************************************
**

 DEFINITIONS OF GLOBALS

 ***************************************************************************
*/

int ShowFPS = true;
int DisplayMode = COLMODE_COL;

struct ImgConf {
  int resolution;
  int camMode;
  double exposureMS;
};

//
***************************************************************************
int main(int argc, char *argv[]) {
//
***************************************************************************
  double t;
  int key;
  int cntframe = 0;

  string camName;
  char countxt[90];

  struct ImgConf cfg;

  // set default values
  cfg.resolution = CAM_RES_640_480;
  cfg.camMode = CAM_MODE_COL;
  cfg.exposureMS = 12.34; // setting default exposure time in milliseconds

  //  set the configuration according to commandline-parameters
  Config(argc, argv, &cfg);

  //  call the constructor and open default camera
  //  if this does NOT succeed the program will abort here (see: constructor)
  dh cam0(0);

  //  declare the matrix where our image is stored
  Mat image;

  //  set camera-mode and exposure-time
  cam0.setMode(cfg.resolution, cfg.camMode);
  cam0.setExpoMs(cfg.exposureMS);

  //  get camera name
  cam0.getName(&camName);
```

```
  cout << "using device '" << camName << "' " << endl;

  //  initialize our OpenCV display window
  InitWindows(camName);

  // discard first image to let camera settle
  cam0.captureFrame(&image);

  //  get systemtime to calculate frame-rate later on
  t = (double) getTickCount();

  // get actual exposuretime
  cam0.getExpoMs (&cfg.exposureMS);
  cout << "Using resolution: " << image.cols << " by " << image.rows
      << ", exposuretime: " << cfg.exposureMS << " ms" << endl;

  //  here the main-loop starts, read one frame
  while (cam0.captureFrame(&image) == CAM_OK) {
    //  increment frame counter
    cntframe++;

    //  check if retrieving image was successful
    if (!image.empty()) {

      //  check is we have to convert the image to grey-scale
      switch (DisplayMode) {
      case COLMODE_COL: // normal color
        break;

      case COLMODE_GREY: // grey-scale
        ConvertGrey(&image);
        break;
      }

      //  check if we have to display frame-rate
      if (ShowFPS == true) {
        //  define location where to display the frame-rate
        Point org;
        org.x = 10;
        org.y = 30;

        //  calculate the expired time since last acquisition of frame
        t = ((double) getTickCount() - t) / getTickFrequency();

        sprintf(countxt, "fps: %4.1f [%06d], exp: %6.2f [ms]",
            (1.0 / t), cntframe, cfg.exposureMS);
//        sprintf(countxt, "fps: %4.1f [%06d], exp: %6.2f [ms]",
//            (1.0 / t), cntframe, cam0.getExpoMs());

        //  get new time
        t = (double) getTickCount();

        //  print string to image-buffer
        putText(image, countxt, org, 1, 2, Scalar(0, 255, 255), 2, 16,
            false);
      }

      // display frame in standard window
```

```
      imshow(camName, image);
    }

    // make frame visible
    key = waitKey(1);

//    if (key != -1)
//      cout << "key: '" << key << "'  " << endl;

// check for 'Esc' (or 'backspace' or 'enter') to stop
    if ((key == 0x1b) || (key == 0x08) || (key == 0x0d)) {
      cout << "Stopping Cam!" << endl;
      cam0.close();
      break;
    } else {
      // check for keyboard commands
      switch (key) {

      case '?':
        cout << "ROI width = " << image.cols << ", height = "
            << image.rows << endl;
        break;

      case 'e':
        cout << "Exposure time set to: " << cfg.exposureMS << " ms"
            << endl;
        break;

      case ' ':
        // save image using openCV API
        break;
      }
    }
  }

  return 0;
}

//
*****************************************************************************
void ConvertGrey(Mat *image) {
//
*****************************************************************************
  //  go through all cols and rows and convert each pixel to gray value
  //  grey = 0.299 * red + 0.587 * green + 0.114 * blue
  for (int r = 0; r < image->rows; r++) {
    for (int c = 0; c < image->cols; c++) {
      Vec3b &rgb = image->at<Vec3b>(r, c);

      rgb[COL_RED] = (unsigned char) (0.299 * (float) rgb[COL_RED]
          + 0.587 * (float) rgb[COL_GREEN]
          + 0.114 * (float) rgb[COL_BLUE]);
      rgb[COL_GREEN] = rgb[COL_RED];
      rgb[COL_BLUE] = rgb[COL_RED];
    }
  }
}

//
```

```
//*****************************************************************************
int Config(int argc, char **argv, struct ImgConf *camCfg) {
//*****************************************************************************
  //  read commandline-parameters one by one
  if (argc > 1) {
    for (int i = 1; i < argc; i++) {
      if (argv[i][0] == '-') {

        //  check for help
        if (argv[i][1] == '?') {
          PrintHelp();
        }

        //  check for frames per second display
        if (argv[i][1] == 'F') {
          cout << "show FPS!" << endl;
          ShowFPS = true;
        }

        //  check for grey-scale display
        if (argv[i][1] == 'G') {
          cout << "show grey-scale image" << endl;
          DisplayMode = COLMODE_GREY;
        }
      }
    }
  } else {
    PrintHelp();
  }

  return 0;
}

//*****************************************************************************
int PrintHelp(void) {
//*****************************************************************************
  cout << "DaHeng USB3 Camera-Framework, V1.0" << endl;
  cout << "(c) F. Theinert 2022" << endl;
  cout << "Commandline options: -F -G -?" << endl;
  cout << "  -F show frames per second" << endl;
  cout << "  -G grey-scale image" << endl;
  cout << "  -? this help-screen" << endl;

  return 0;
}

//****************************************************************************
void InitWindows(string camName) {
//****************************************************************************

  // make HighGui OpenCV window for display
  namedWindow(camName, WINDOW_AUTOSIZE | WINDOW_GUI_NORMAL);
}
```

```
///* EOF hhs_cam.cpp  */
```

B Appendix B

```
#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>

using namespace cv;

void filter(Mat* input, Mat* result) {
  Size s = input->size();
  long h, w;
  long sum;

  //std::cout << "Input type was : " << input->type() << std::endl;
  uint8_t out[s.height][s.width];

  std::cout << s.height << " " << s.width << std::endl;

  for (w=0; w<s.width; w++){ //loop over the image,
    for(h=0; h<s.height; h++){
      sum = 0;
      std::vector<uint8_t> median;

      for (int _x = -1; _x < 2; ++_x) //for every pixel, loop over every pixel in a 3x3 kernel
      {
        for (int _y = -1; _y < 2; ++_y)
        {
          int idx_y = h + _y; //sum the incrementor with the kernel's, so we can identify borders
          int idx_x = w + _x;

          if (idx_x < 0 || idx_x > s.width) //the kernel goes outside of the image, therefore break and ignore that pixel.
            break;

          if (idx_y < 0 || idx_y > s.height)
            break;

          median.push_back(input->at<uint8_t>(idx_y,idx_x)); // Add all the pixels from the kernel into a vector
        }
      }
      std::sort(std::begin(median), std::end(median)); //sort all pixel values from high to low

      for (auto it = median.begin(); it != median.end(); ++it) {
        sum = median.at(median.size()/2); //The pixel at the y,x coordinate is now the median from our 3x3 sliding window
      }
      out[h][w]=(uint8_t)sum;
    }
  }
  // this did not work, since the out array was never copied to memory, causing image data pointing to nowhere!
  //result = Mat(s.height, s.width, CV_8U, out);
  // Instead, the data from out needs to be copied directly to Mat result
```

```
with the correct size
  std::memcpy(result->data, out, s.height*s.width*sizeof(uint8_t));
}

void gammaCorrection(Mat* input, Mat* output, float gamma) {
  Size s = input->size();
  long h, w;
  float sum;

  std::cout << "Correcting gamma" << std::endl;
  uint8_t out[s.height][s.width];
  uint8_t lut[256];

  for (int i = 0; i < 256; i++) { //create a lookup table, with the gamma correction curve.
    lut[i] = saturate_cast<uint8_t>(pow((float)(i / 255.0), gamma) * 255.0f); //saturate cast negative values to 0, and higher values to 255 (uint8_t or unsigned char)
    //std::cout << unsigned(lut[i]) << " "; //print the function for testing. cout prints uint8_t as chars so we cast it.
  }


  for (w=0; w<s.width; w++){ //loop over the image,
    for(h=0; h<s.height; h++){
      sum = lut[(input->at<uint8_t>(h,w))]; //the original output value will be scaled to the value in the LUT.
      out[h][w]=(uint8_t)sum;
    }
  }
  std::memcpy(output->data, out, s.height*s.width*sizeof(uint8_t)); //copy our standard 2D array to a new buffer that OpenCV understands
}


int main() {
  // Read the image (in BGR)
    Mat img = imread("pixerror.png", IMREAD_COLOR);
    if(img.empty())
    {
        std::cout << "Could not read the image: " << std::endl;
        return 1;
    }
    Size imgsize = img.size();

    // Split the image into 3 new images for blue, green and red.
    std::cout << "Splitting channels: " << std::endl;
  Mat bands[3];
  split(img, bands);

  Mat bandsFiltered[3];
  Mat bandsCorrected[3];
  bandsFiltered[0] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsFiltered[1] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsFiltered[2] = Mat(imgsize.height, imgsize.width, CV_8U);

  bandsCorrected[0] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsCorrected[1] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsCorrected[2] = Mat(imgsize.height, imgsize.width, CV_8U);
```

```
  filter(&bands[0],&bandsFiltered[0]); //filter all channels from noise individually
  filter(&bands[1],&bandsFiltered[1]);
  filter(&bands[2],&bandsFiltered[2]);

  gammaCorrection(&bandsFiltered[0],&bandsCorrected[0],0.33);
  gammaCorrection(&bandsFiltered[1],&bandsCorrected[1],0.33);
  gammaCorrection(&bandsFiltered[2],&bandsCorrected[2],0.33);

  Mat merged;
  std::vector<Mat> channels = {bandsCorrected[0],bandsCorrected[1],bandsCorrected[2]};
  merge(channels, merged);

    // Display the image until q is pressed
    std::cout << "Displaying result: " << std::endl;
    imshow("Display window", bands[0]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bands[1]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bands[2]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[0]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[1]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[2]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", merged);
    waitKey(0); // Wait for a keystroke in the window
    return 0;
}
```

C Appendix C

```
#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>

using namespace cv;

void filter(Mat* input, Mat* result) {
  Size s = input->size();
  long h, w;
  long sum;

  //std::cout << "Input type was : " << input->type() << std::endl;
  uint8_t out[s.height][s.width];

  std::cout << s.height << " " << s.width << std::endl;

  for (w=0; w<s.width; w++){ //loop over the image,
    for(h=0; h<s.height; h++){
      sum = 0;
      std::vector<uint8_t> median;

      for (int _x = -1; _x < 2; ++_x) //for every pixel, loop over every pixel in a 3x3 kernel
      {
        for (int _y = -1; _y < 2; ++_y)
        {
          int idx_y = h + _y; //sum the incrementor with the kernel's, so we can identify borders
          int idx_x = w + _x;

          if (idx_x < 0 || idx_x > s.width) //the kernel goes outside of the image, therefore break and ignore that pixel.
            break;

          if (idx_y < 0 || idx_y > s.height)
            break;

          median.push_back(input->at<uint8_t>(idx_y,idx_x)); // Add all the pixels from the kernel into a vector
        }
      }
      std::sort(std::begin(median), std::end(median)); //sort all pixel values from high to low

      for (auto it = median.begin(); it != median.end(); ++it) {
        sum = median.at(median.size()/2); //The pixel at the y,x coordinate is now the median from our 3x3 sliding window
      }
      out[h][w]=(uint8_t)sum;
    }
  }
  // this did not work, since the out array was never copied to memory, causing image data pointing to nowhere!
  //result = Mat(s.height, s.width, CV_8U, out);
  // Instead, the data from out needs to be copied directly to Mat result
```

```
with the correct size
  std::memcpy(result->data, out, s.height*s.width*sizeof(uint8_t));
}

void gammaCorrection(Mat* input, Mat* output, float gamma) {
  Size s = input->size();
  long h, w;
  float sum;

  std::cout << "Correcting gamma" << std::endl;
  uint8_t out[s.height][s.width];
  uint8_t lut[256];

  for (int i = 0; i < 256; i++) { //create a lookup table, with the gamma correction curve.
    lut[i] = saturate_cast<uint8_t>(pow((float)(i / 255.0), gamma) * 255.0f); //saturate cast negative values to 0, and higher values to 255 (uint8_t or unsigned char)
    //std::cout << unsigned(lut[i]) << " "; //print the function for testing. cout prints uint8_t as chars so we cast it.
  }


  for (w=0; w<s.width; w++){ //loop over the image,
    for(h=0; h<s.height; h++){
      sum = lut[(input->at<uint8_t>(h,w))]; //the original output value will be scaled to the value in the LUT.
      out[h][w]=(uint8_t)sum;
    }
  }
  std::memcpy(output->data, out, s.height*s.width*sizeof(uint8_t)); //copy our standard 2D array to a new buffer that OpenCV understands
}


int main() {
  // Read the image (in BGR)
    Mat img = imread("pixerror.png", IMREAD_COLOR);
    if(img.empty())
    {
        std::cout << "Could not read the image: " << std::endl;
        return 1;
    }
    Size imgsize = img.size();

    // Split the image into 3 new images for blue, green and red.
    std::cout << "Splitting channels: " << std::endl;
  Mat bands[3];
  split(img, bands);

  Mat bandsFiltered[3];
  Mat bandsCorrected[3];
  bandsFiltered[0] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsFiltered[1] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsFiltered[2] = Mat(imgsize.height, imgsize.width, CV_8U);

  bandsCorrected[0] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsCorrected[1] = Mat(imgsize.height, imgsize.width, CV_8U);
  bandsCorrected[2] = Mat(imgsize.height, imgsize.width, CV_8U);
```

```
  filter(&bands[0],&bandsFiltered[0]); //filter all channels from noise individually
  filter(&bands[1],&bandsFiltered[1]);
  filter(&bands[2],&bandsFiltered[2]);

  gammaCorrection(&bandsFiltered[0],&bandsCorrected[0],0.33);
  gammaCorrection(&bandsFiltered[1],&bandsCorrected[1],0.33);
  gammaCorrection(&bandsFiltered[2],&bandsCorrected[2],0.33);

  Mat merged;
  std::vector<Mat> channels = {bandsCorrected[0],bandsCorrected[1],bandsCorrected[2]};
  merge(channels, merged);

    // Display the image until q is pressed
    std::cout << "Displaying result: " << std::endl;
    imshow("Display window", bands[0]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bands[1]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bands[2]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[0]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[1]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", bandsFiltered[2]);
    waitKey(0); // Wait for a keystroke in the window
    imshow("Display window", merged);
    waitKey(0); // Wait for a keystroke in the window
    return 0;
}
```

D Appendix D

```cpp
#include <iostream>

#include <opencv2/core.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <opencv2/opencv.hpp>

#define BGR_BLUE  0
#define BGR_GREEN 1
#define BGR_RED   2


using namespace cv;

void deBayer(Mat *rawImg, Mat *outImg) {
  // the Daheng camera datasheet specifies a GRBG pattern (page 88). Our input values will be 8 bit
  // Split 1 channel image into 3 channels according to bayer pattern
  //   0  1
  //0  G  R
  //1  B  G

  //I will create a new "Mat" with OpenCV that contains three channels. The splitting and processing is done by hand.
  if (rawImg->type() != CV_8UC1)
    throw("Sorry, only 1 8-bit channel should be used");

  (*outImg) = cv::Mat::zeros(rawImg->rows, rawImg->cols, CV_8UC3); // Fill output buffer with zeros with the correct geometry
  MAT outImgMID = cv::Mat::zeros(rawImg->rows, rawImg->cols, CV_8UC3); // Fill output buffer with zeros with the correct geometry
  //Size s = rawImg->size(); I will be using rows and colums rather than height and width
  long row, col;

  std::cout << "Bayer splitting to 3 channels" << std::endl;

  for(row=0; row<rawImg->rows; row++){  //todo: make this evaluation smaller to increase speed
    for(col=0; col<rawImg->cols; col++){
      if (row % 2 == 0 && col % 2 == 0) //odd row, odd column
        outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] = rawImg->at<uint8_t>(row,col);
      if (row % 2 == 0 && col % 2 == 1) //odd row, even column
        outImgMID->at<Vec3b>(row, col).val[BGR_RED] = rawImg->at<uint8_t>(row,col);
      if (row % 2 == 1 && col % 2 == 0) //even row, odd column
        outImgMID->at<Vec3b>(row, col).val[BGR_BLUE] = rawImg->at<uint8_t>(row,col);
      if (row % 2 == 1 && col % 2 == 1) //even row, even column
        outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] = rawImg->at<uint8_t>(row,col);
    }
  }

  // Interpolate green channel by taking the value of the nearest neighbour that is green
  std::cout << "Interpolating green channel" << std::endl;
```

```
  for(row=0; row<outImgMID->rows; row++){
    for(col=0; col<outImgMID->cols; col++){
      if (outImgMID->at<Vec3b>(row, col).val[BGR_GREEN] == 0){
        if (col > 0 && outImgMID->at<Vec3b>(row, col-1).val[BGR_GREEN] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row, col-1).val[BGR_GREEN];
        else if (col < outImgMID->cols-1 && outImgMID->at<Vec3b>(row, col+1).val[BGR_GREEN] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row, col+1).val[BGR_GREEN];
        else if (row > 0 && outImgMID->at<Vec3b>(row-1, col).val[BGR_GREEN] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row-1, col).val[BGR_GREEN];
        else if (row < outImgMID->rows-1 && outImgMID->at<Vec3b>(row+1, col).val[BGR_GREEN] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_GREEN] = outImgMID->at<Vec3b>(row+1, col).val[BGR_GREEN];
      }
    }
  }

  // Interpolate red channel by taking the value of the nearest neighbour that is red
  std::cout << "Interpolating red channels" << std::endl;

  for(row=0; row<outImgMID->rows; row++){
    for(col=0; col<outImgMID->cols; col++){
      if (outImgMID->at<Vec3b>(row, col).val[BGR_RED] == 0){
        if (col > 0 && outImgMID->at<Vec3b>(row, col-1).val[BGR_RED] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_RED] = outImgMID->at<Vec3b>(row, col-1).val[BGR_RED];
        else if (col < outImgMID->cols-1 && outImgMID->at<Vec3b>(row, col+1).val[BGR_RED] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_RED] = outImgMID->at<Vec3b>(row, col+1).val[BGR_RED];
        else if (row > 0 && outImgMID->at<Vec3b>(row-1, col).val[BGR_RED] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_RED] = outImgMID->at<Vec3b>(row-1, col).val[BGR_RED];
        else if (row < outImgMID->rows-1 && outImgMID->at<Vec3b>(row+1, col).val[BGR_RED] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_RED] = outImgMID->at<Vec3b>(row+1, col).val[BGR_RED];
      }
    }
  }

  // Interpolate blue channel by taking the value of the nearest neighbour that is blue
  std::cout << "Interpolating blue channels" << std::endl;

  for(row=0; row<outImgMID->rows; row++){
    for(col=0; col<outImgMID->cols; col++){
      if (outImgMID->at<Vec3b>(row, col).val[BGR_BLUE] == 0){
        if (col > 0 && outImgMID->at<Vec3b>(row, col-1).val[BGR_BLUE] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_BLUE] = outImgMID->at<Vec3b>(row, col-1).val[BGR_BLUE];
```

```
        else if (col < outImgMID->cols-1 && outImgMID->at<Vec3b>(row, col+1).val[BGR_BLUE] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_BLUE] = outImgMID->at<Vec3b>(row, col+1).val[BGR_BLUE];
        else if (row > 0 && outImgMID->at<Vec3b>(row-1, col).val[BGR_BLUE] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_BLUE] = outImgMID->at<Vec3b>(row-1, col).val[BGR_BLUE];
        else if (row < outImgMID->rows-1 && outImgMID->at<Vec3b>(row+1, col).val[BGR_BLUE] != 0)
          outImg->at<Vec3b>(row, col).val[BGR_BLUE] = outImgMID->at<Vec3b>(row+1, col).val[BGR_BLUE];
      }
    }
  }

  // done?
  std::cout << "Done?" << std::endl;

}

int main() {
  // Read the image (in 8bit grayscale)
    Mat img = imread("test_RAW.png", CV_8UC1);
    if(img.empty()) {
        std::cout << "Could not read the image: " << std::endl;
        return 1;
    }

    Mat result;
    deBayer(&img, &result);


    imshow("Display window", result);
    waitKey(0); // Wait for a keystroke in the window
    return 0;
}
```