CSC4140

Final Project

ISP for RGB-IR sensor

Name: **Yuyang LIN**

Student ID：**120090377**

Result Overview

- All the image used in this report can be found in the `./image` folder

- `1_1.RAW` picture with light on
- Total average pipeline time `15.58s` on `Intel i7 12700H`
- Total average pipeline time `9.58s` `on Apple M1 Chip`
- which is basically the same with original pipeline and fulfills the state of the art requirements of `realtime` image processing pipeline for `RGB-IR sensor`
- With `NLM` the longest pipeline (8.375s) and modified `CFA` only took 4s
- With AAF:
-

- Without AAF (Sharpen, Very Close to sample):
-

- `1_1.RAW` picture with no light on
- Total pipeline performance matrix is the same with the previouse one
- With AAF

- 
- Without AAF
- 

- `test.yaml` used in the above sample image

```yaml
module_enable_status:                   # do NOT modify modules order
  dpc: True
  blc: True
  aaf: False
  awb: True
  cnf: True
  cfa: True
  ccm: True
  gac: True
  csc: True
  nlm: True
  bnf: True
  ceh: True
  eeh: True
  fcs: True
  hsc: True
  bcc: True
  scl: False

hardware:
  raw_width: 2592
  raw_height: 1944
  raw_bit_depth: 10
  bayer_pattern: rgb-ir


# -------------------- Module Algorithms Parameters ----------------
----

dpc:
  diff_threshold: 30

blc:
  bl_r: 0                               # a subtractive value, not
additive!
  bl_gr: 0
  bl_gb: 0
  bl_b: 0
  alpha: 0                              # x1024
  beta: 0                               # x1024
```

```yaml
aaf: ~

awb:
  r_gain: &r_gain 2415                  # x1024
  gr_gain: 1024                         # x1024
  gb_gain: 1024                         # x1024
  b_gain: &b_gain 1168                  # x1024

cnf:
  diff_threshold: 0
  r_gain: *r_gain                       # x1024
  b_gain: *b_gain                       # x1024

cfa:
  mode: custom

ccm:
  ccm:
    - [ 1896, -811, -62, 0 ]            # x1024, copied from dcraw.c
    - [ -160, 1687, -503, 0 ]
    - [ 18, -557, 1563, 0 ]

gac:
  gain: 256                             # x256
  gamma: 0.95

csc: ~

nlm:
  search_window_size: 9
  patch_size: 3
  h: 10                                 # larger value has stronger
smoothing effect

bnf:
  intensity_sigma: 0.5                  # larger value has stronger
smoothing effect
  spatial_sigma: 0.4                    # larger value has stronger
smoothing effect
```

```
ceh:
  tiles: [4, 6]
  clip_limit: 0.01

eeh:
  edge_gain: 512                    # x256
  flat_threshold: 2                 # delta <= flat_threshold: set
delta to 0
  edge_threshold: 4                 # delta > edge_threshold:
increase delta by edge_gain
  delta_threshold: 64

fcs:
  delta_min: 8
  delta_max: 32

hsc:
  hue_offset: 0                     # in degree
  saturation_gain: 256              # x256

bcc:
  brightness_offset: 0
  contrast_gain: 256                # x256

scl:
  width: 1536
  height: 1024
```

1. How to run

1.1 Requirements

> - The project is written in python and in the pipeline code structure provided by the instructor. [fast open-ISP]

- The Github repo of this project will be made public after DDL, you may refer to this **link** for more information. FYI, there is also a vulkan based `BDPT` renderer named `LUNA` and a game engine project named `ICARUS` working in progress in my **Github,**
- The `requirements.txt` should be found on the root directory of this project.
- For installing all the requirements in this project, you may choose one of the following.
- **Option 1**

```
> python -m pip install -r requirements.txt
```

- **Option 2**

```
> conda create -n CSC4140_venv python=3.8
> conda activate CSC4140_venv
> python -m pip install -r requirements.txt
```

- `requirements.txt`

```
numpy=1.24.2
opencv-python=4.7.0.72
scikit-image=0.20.0
scipy=1.9.1
```

1.2 Run the project

- cd into the root directory of this project

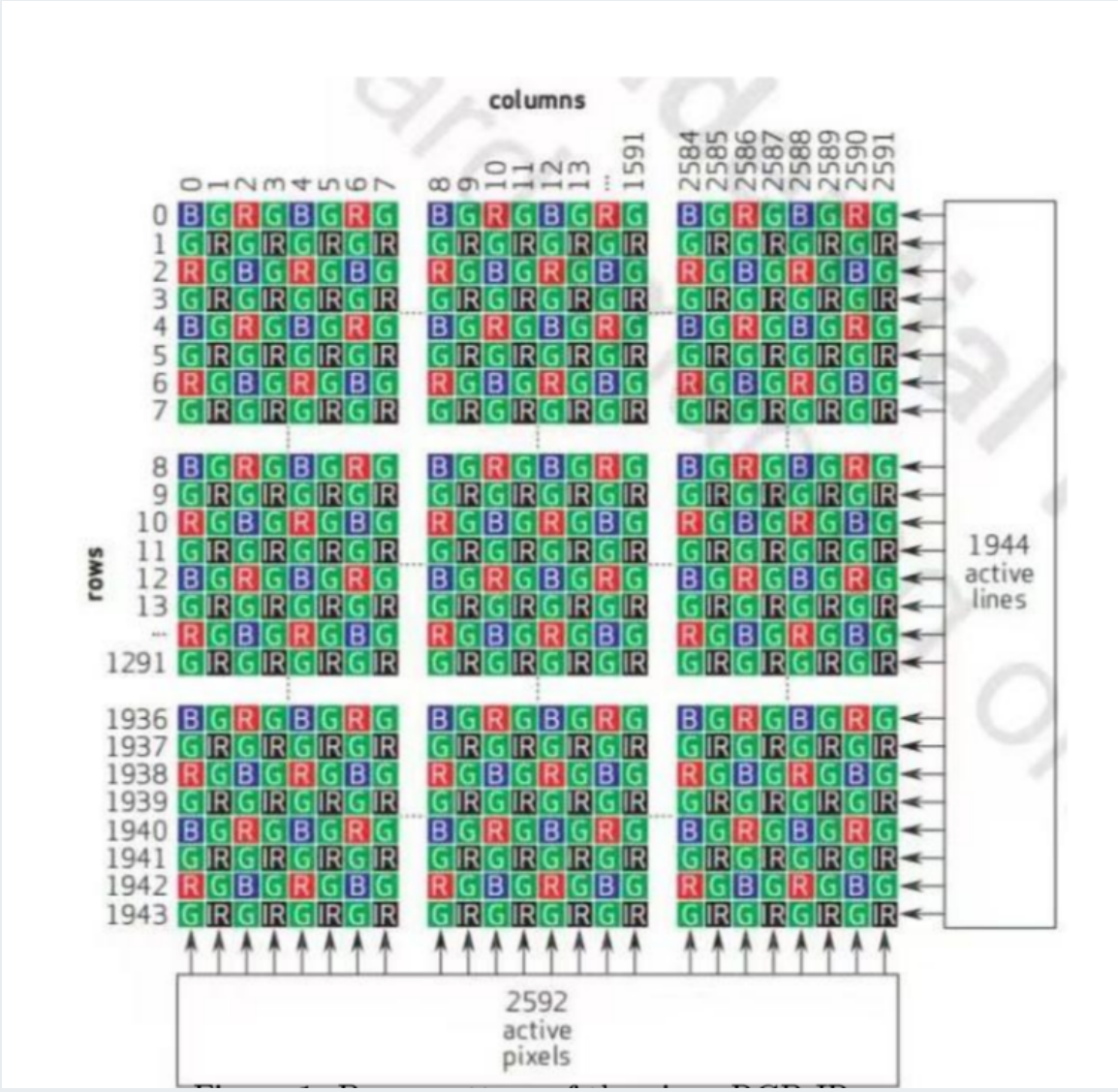
```
> python demo.py
```

- You can check the output image in the `./output` folder

- To change the input test image, go to the `demo.py` and change the corresponding input file.

Overview

- In this project, we are required to extend one of the current Image Signal Processing pipeline proposed by **Jueqin Qiu**.
- The original pipeline is designed for processing `RGGB` raw image with simple Bayer pattern and convert it into a `RGB` image and done some preprocessing steps.
- The modification this project made is to change the pipeline to adapt to the `RGB-IR` Sensor, which has completely different image Bayer pattern compared with the previous one.
-



- The pattern is as followed
-

  - This `RGB-NIR CFA pattern` would support the `RGB` and `IR` in one-shot acquisition. However, the additional `IR` pixels placed in a full pixel active area, it will cause the full pixel resolution to drop out half red and half blue pixels compared to a single RGB picture. This kind of mechanism will cause the `RGB` image performance to be low and requires a complex algorithm in the `ISP` to reproduce the `RGB` image.

- The overall `ISP` provided in the sample code structure consists of several modules;

- **DPC**(Defective Pixel Correction)： This stage identifies and corrects any defective pixels in the image sensor.
- **BLC**(Black Level Correction): This stage adjusts the black level or baseline offset to eliminate any unwanted variations in the sensor's output, ensuring accurate color reproduction.
- **AAF**(Anti-Aliasing Filter): This stage applies an anti-aliasing filter helps reduce the occurrence of moiré patterns and aliasing artifacts by smoothing out high-frequency components in the image.
- **AWB**(Auto White Balance): This stage automatically adjusts the color balance of the image to compensate for different lighting conditions, ensuring accurate white color representation.
- **CNF**(Color Noise Filtering): This stage reduces color noise present in the image, improving overall image quality.
- **CFA**(Color Filter Array): This stage reconstructs the full-color image by interpolating the missing color information from the sensor's Bayer pattern, which is `RGB-IR` in this cases.
- **CCM**(Color Correction Matrix): This stage applies a color correction matrix to adjust the color rendition and ensure accurate color reproduction.
- **GAC**(Gamma and Contrast Matrix): This stage adjusts the image's tonal response, while contrast correction enhances the visual contrast for better image quality.
- **CSC**(Color Space Conversion): This stage converts the image from one color space to another, such as from RGB to YUV or vice versa, to meet the requirements of downstream processing or display devices.
- **NLM**(Non-Local Means Denoising): This stage is a technique used to reduce noise in the image while preserving details and textures.
- **BNF**(Brightness and Saturation Filter): This stage adjusts the image's brightness and saturation levels to enhance its visual appearance.
- **CEH**(Contrast Enhancement and Histogram Equalization): This stage improves image contrast and equalizes the histogram distribution for better visibility and detail.
- **EEH**(Edge Enhancement): This stage enhances the edges in the image to improve sharpness and perceived image quality.
- **FCS**(Fixed Pattern Noise Correction): This stage aims to remove any fixed-pattern noise introduced by the image sensor.
- **HCS**(High Sensitivity Control): It adjusts the sensitivity or ISO level of the image sensor to adapt to different lighting conditions, improving low-light performance.
- **BCC**(Blemish Correction and Clipping): This stage corrects blemishes or artifacts in the image and prevents clipping in high-intensity areas.

    - **SCL**(Sharpening and Clarity): The sharpening and clarity stage enhances the image's sharpness and improves overall clarity for better visual impact.

- These stages collectively form an ISP pipeline, which processes the raw sensor data to produce a high-quality image with accurate colors, improved dynamic range, reduced noise, and enhanced details.

- In this project, generally all pipeline before **CFA** and **CFA** itself must be modified since the old pipeline is designed for `RGB` only. But after **CFA**, We have already got the seperate `RGB` and `IR` file from the raw image, the operation on `RGB` file shoud be the same to the old pipeline, so there is no need to change. However, the `EEH` stage is improved since the old one is not satisfactory.

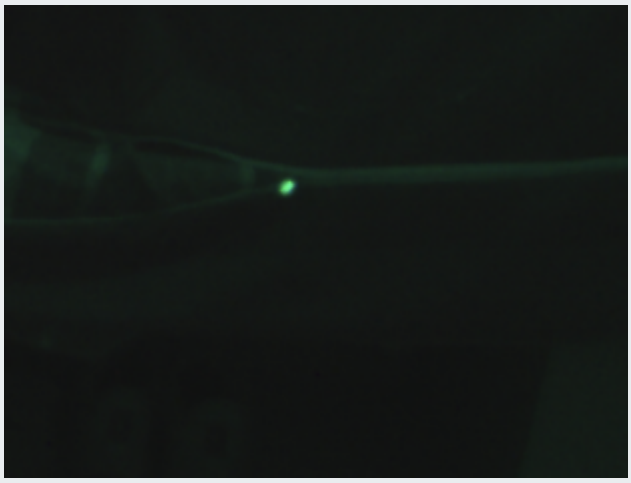
- The main contribution of this project is a fast and state of the art **CFA** including demosacking algorithm and interpolation process. This project has discovered two state of the art algorithm, the first is the residual interpolation proposed in this **paper**, and the second is the current using one, which is based on this **paper**.

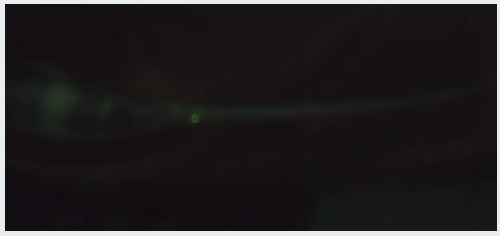
    - The residual interpolation method will have better output and seperate R, G, B, and IR channel naturally. But the time spent on this method is twice as large of the current using one and the method itself is quite complicated, So this is not used.
    - The second method is a simple interpolation but can achieve a well result in significantly short amount of time, which is suitable for real time image signal processing pipiline.
    - The idea behind those method will be discussed in the corresponding section of the report.

ISP for RGB-IR

Defective Pixel Correction

Result

- Before
- 
- After



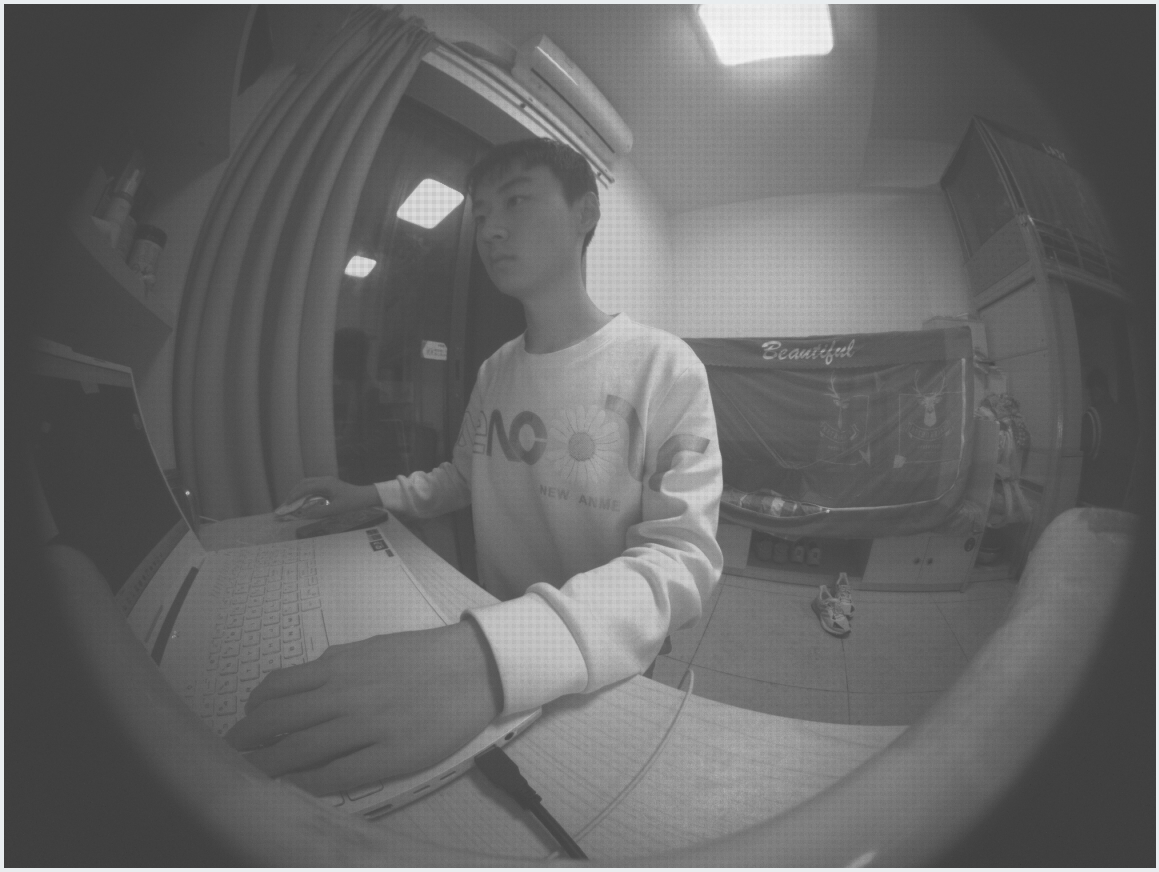
Idea

- Defective pixels can occur in image sensors due to various reasons, such as manufacturing imperfections, physical damage, or aging of the sensor. These pixels may exhibit abnormal behavior compared to the surrounding pixels, resulting in noticeable defects in the captured image. Defective pixels can appear as bright spots (hot pixels) or dark spots (dead pixels) in the image.
- In the before image, you can see there is a hot pixels which is brighter than all the surrounding pixels. And this defective pixel is being detected and fixed in the after image using the algorithm in the pipeline.
- The original pipeline of DPC is actually quite mature and can achieve the state of the art output from the result image provided above, so the algorithm behind it is not modified.
- Hence that one thing need to be modified in the code is the pads used in the padding. The original value is 2 and is suitable for `RGGB` pattern. But, for `rgb-ir`, we need to set `pads = 4`

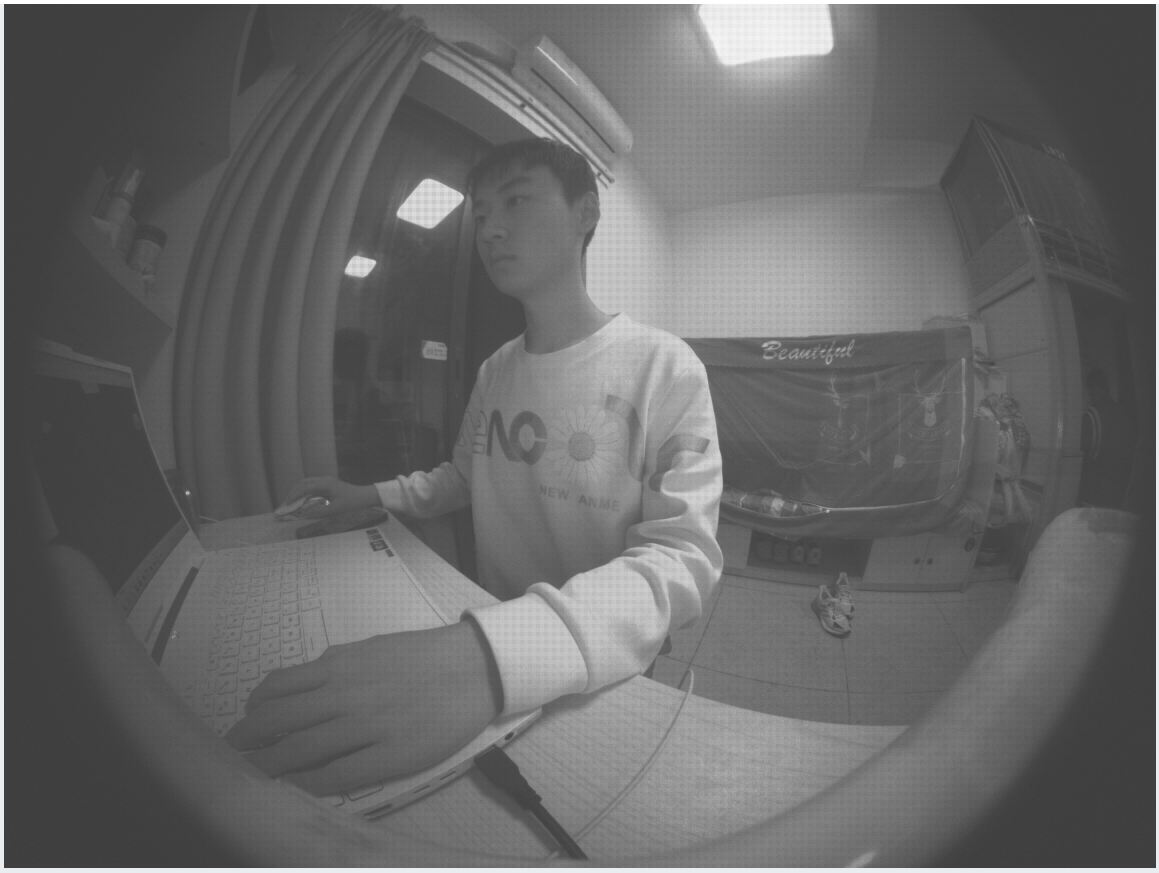
- Also, the helper function used for split Bayer and reconstruct Bayer pattern should be modified for the `RGB-IR` pattern.

Black Level Correction

Result(No light)

- Before
- 

- After
-



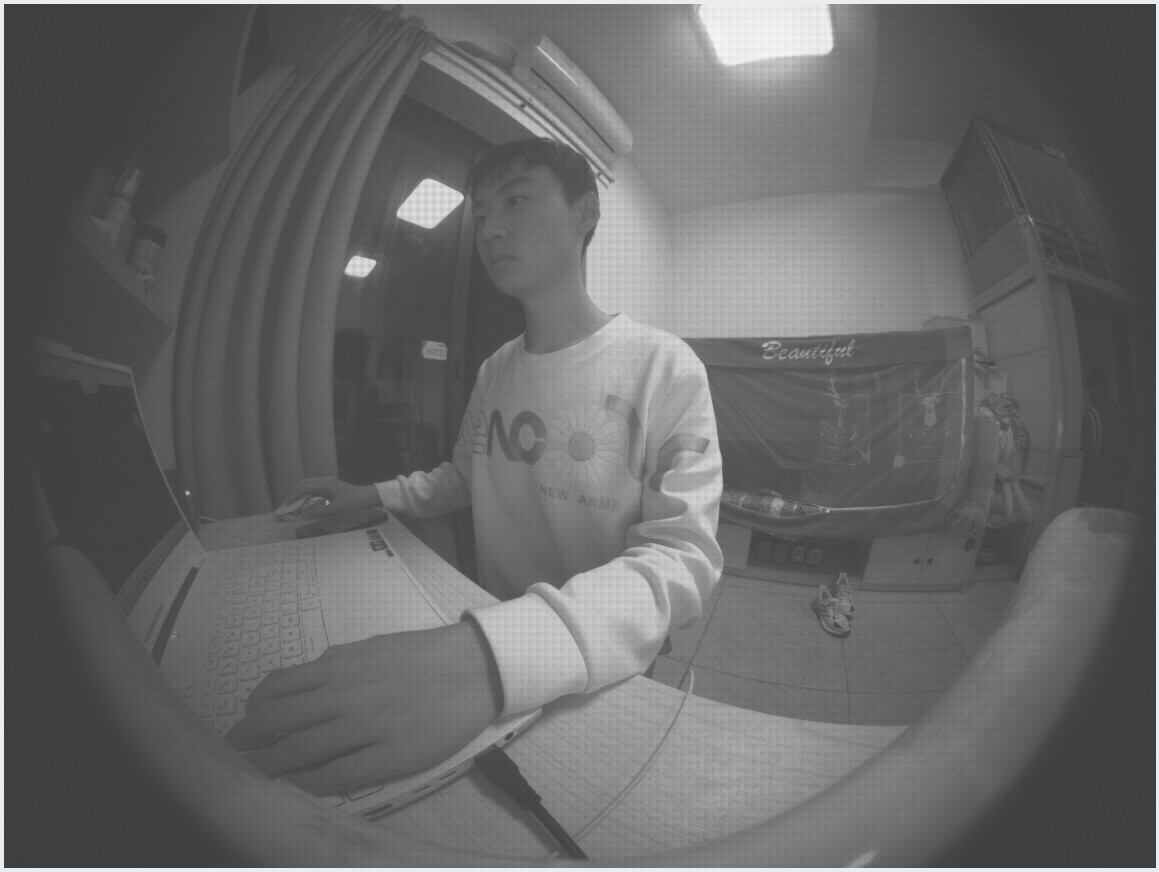
Idea

- Black level refers to the baseline offset or the level of signal produced by the sensor when there is no light input. However, due to various factors like sensor imperfections or electronic noise, the black level may vary across different regions of the sensor or even between individual pixels. These variations can result in an inaccurate representation of black color and affect the overall color balance of the image.
- The purpose of Black Level Correction (BLC) is to compensate for these variations and establish a consistent and accurate black level across the entire image.
- You can observe some slight changes like darkening the whole image from the above output sample image.

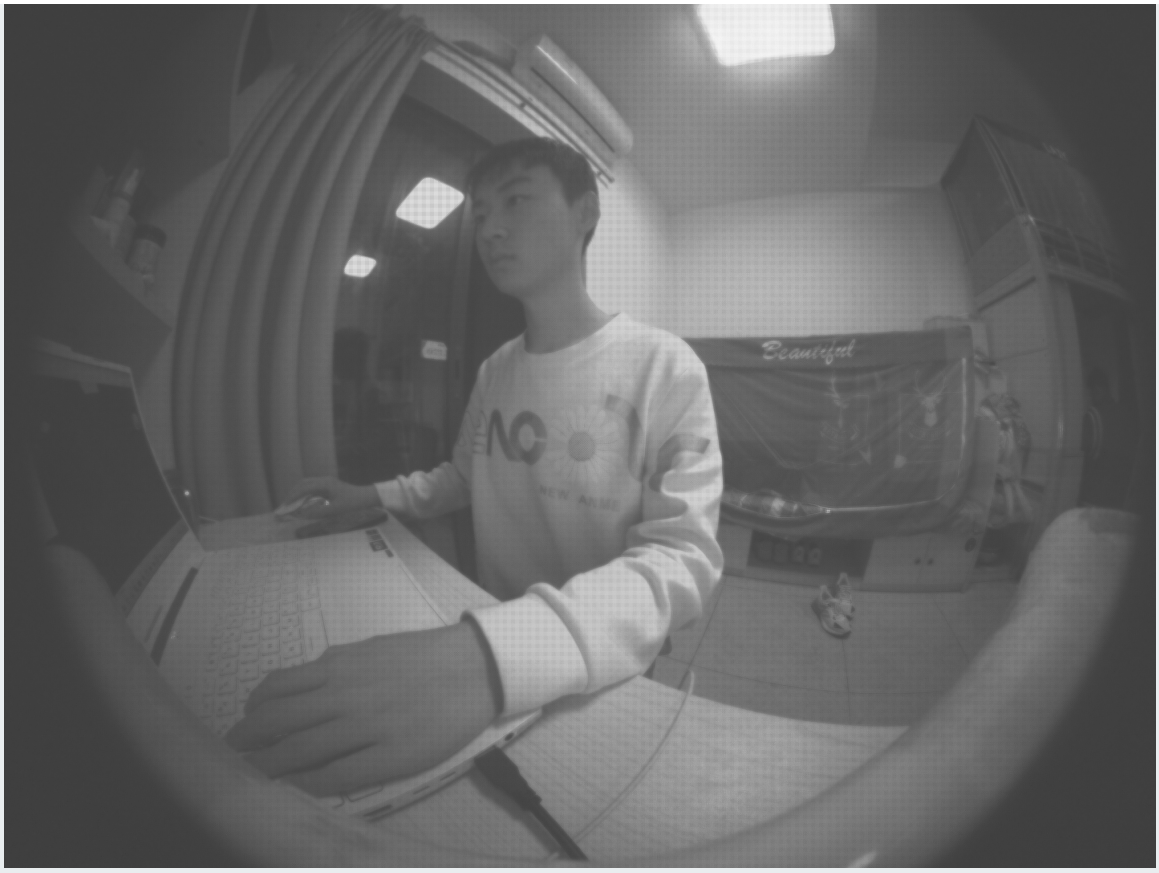
- The basic idea here is about reading the black level value from the `.yaml` file for each channel and clip the range of it. This process is consistent with the old pipeline algorithm but being adpated to fit the need of `RGB-IR` pattern.
- Noted that the Black level adjusment value is now fixed to 0, you may change it by adding additional attributes in the `.yaml` file to use it in the `blc.py`
- But in my implementation, I just used fixed value directly in `blc.py`, you should be aware of such behavior.

Anti-Aliasing Filter

Result(No light)

- Before
- 

- After
-

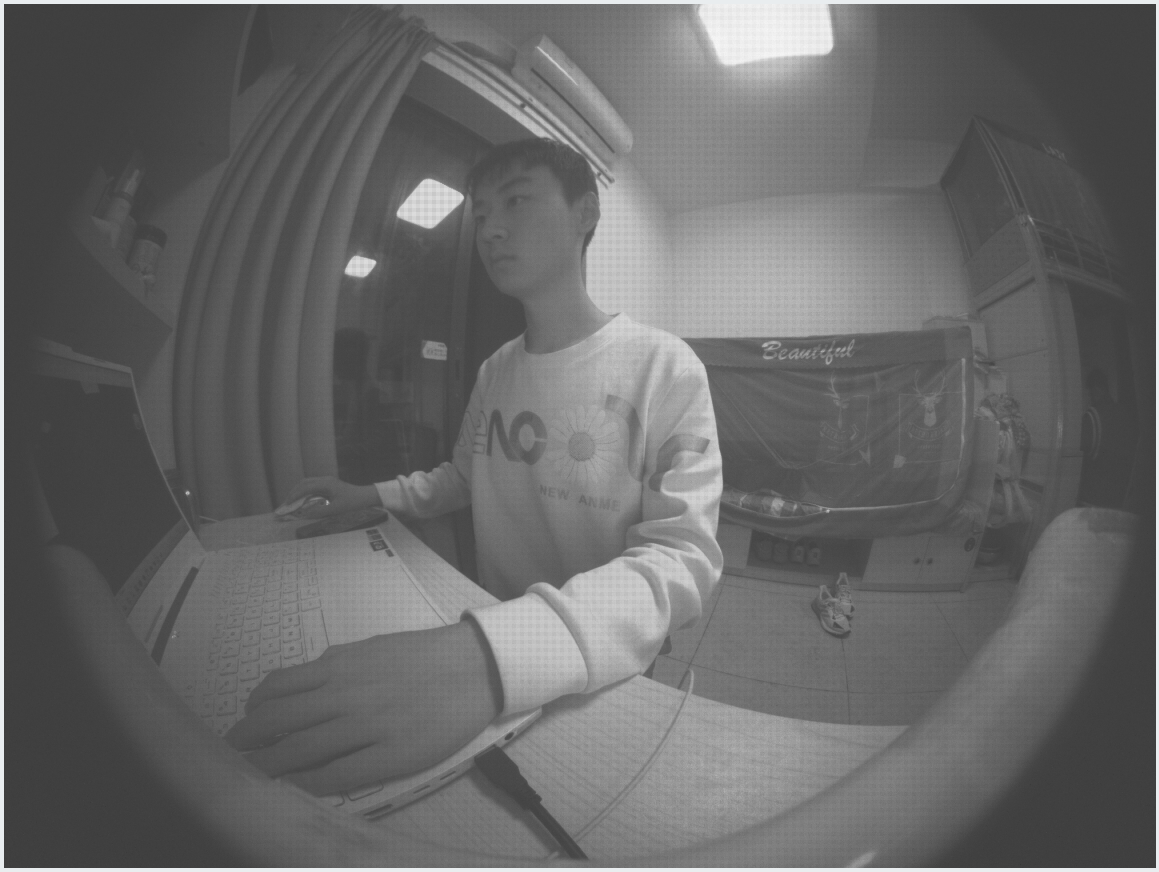


Idea

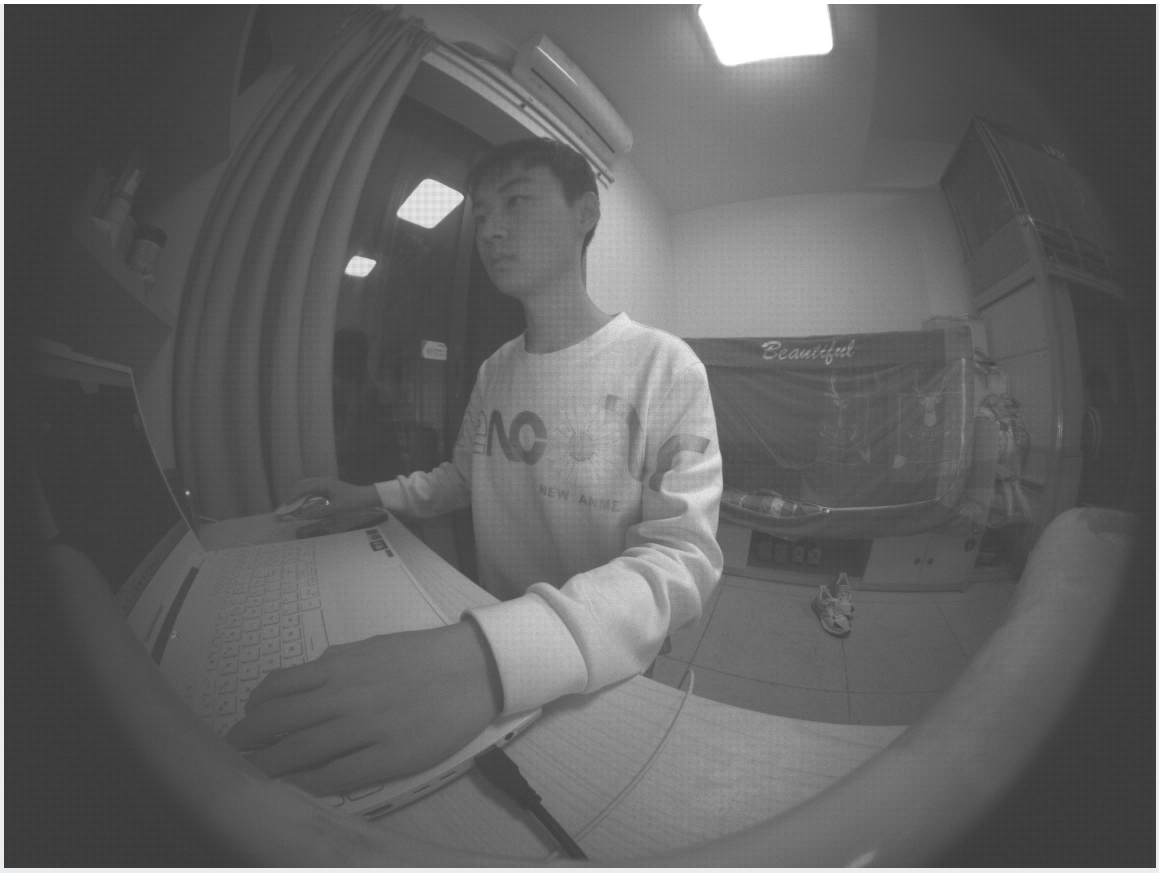
- Aliasing refers to the phenomenon where high-frequency details or patterns in an image appear distorted or produce undesirable artifacts when sampled or digitized at a lower resolution. This occurs due to the interaction between the frequency of the image content and the sampling rate of the sensor. Aliasing artifacts can manifest as moiré patterns, jagged edges, or false colors.
- The purpose of the Anti-Aliasing Filter (AAF) is to reduce or eliminate the occurrence of aliasing artifacts in the captured image. The AAF accomplishes this by attenuating or smoothing out high-frequency components of the image before it is sampled by the image sensor.
- As you can see, the output image is much smoother than the original one but also more blurred.
- I think this is due to the fact that there is no much alias in the original picture, so that the resuslt after AAF will be worse.
- So in the begining of this report, I just include two kinds of output; with and without AAF.
- Specifically in this case the one without AAF will be better and the pipeline with AAF will in general be better.
- The default proposed pipeline is no AAF.

Auto White Balancing

Result(No light, No AAF)

- Before
- 

- After
-



Idea

- Different light sources have different color temperatures, ranging from warm (e.g., incandescent bulbs) to cool (e.g., daylight or fluorescent lighting). The human visual system automatically adjusts to these color temperature variations, perceiving white objects as white regardless of the light source. However, digital cameras need to account for these variations and adjust the color balance to produce accurate and natural-looking images.
- Auto White Balance (AWB) is a feature that automates this process by analyzing the scene and determining the appropriate color balance settings to achieve accurate white color representation.
- You can see from the sample image that the white area of the light is a little bit darken than the original one, which is the appropriate color balance and temperature in this cases.

- The algorithm I used here is called QCGP algorithm, which is a method that combines the perfect reflection and Gray World algorithm. The original pipeline is using fixed value written in the `.yaml` file to do the white balancing, therefore it

is not auto. And the method I'm using here automate the process of White Balancing.

- The Gray World algorithm assumes that the average color of the entire scene should appear gray or neutral, and it adjusts the color gains accordingly. On the other hand, the Perfect Reflection algorithm assumes that some areas in the image should be completely white due to perfect reflection, such as highlights on highly reflective surfaces.
- By combining elements of the Gray World and Perfect Reflection algorithms in a orthogonal way, the "QCGP" algorithm aims to achieve a more accurate and robust Auto White Balance by considering both the overall color balance of the scene and the presence of highly reflective areas.
- You may found reference material about this method in internet.

Code

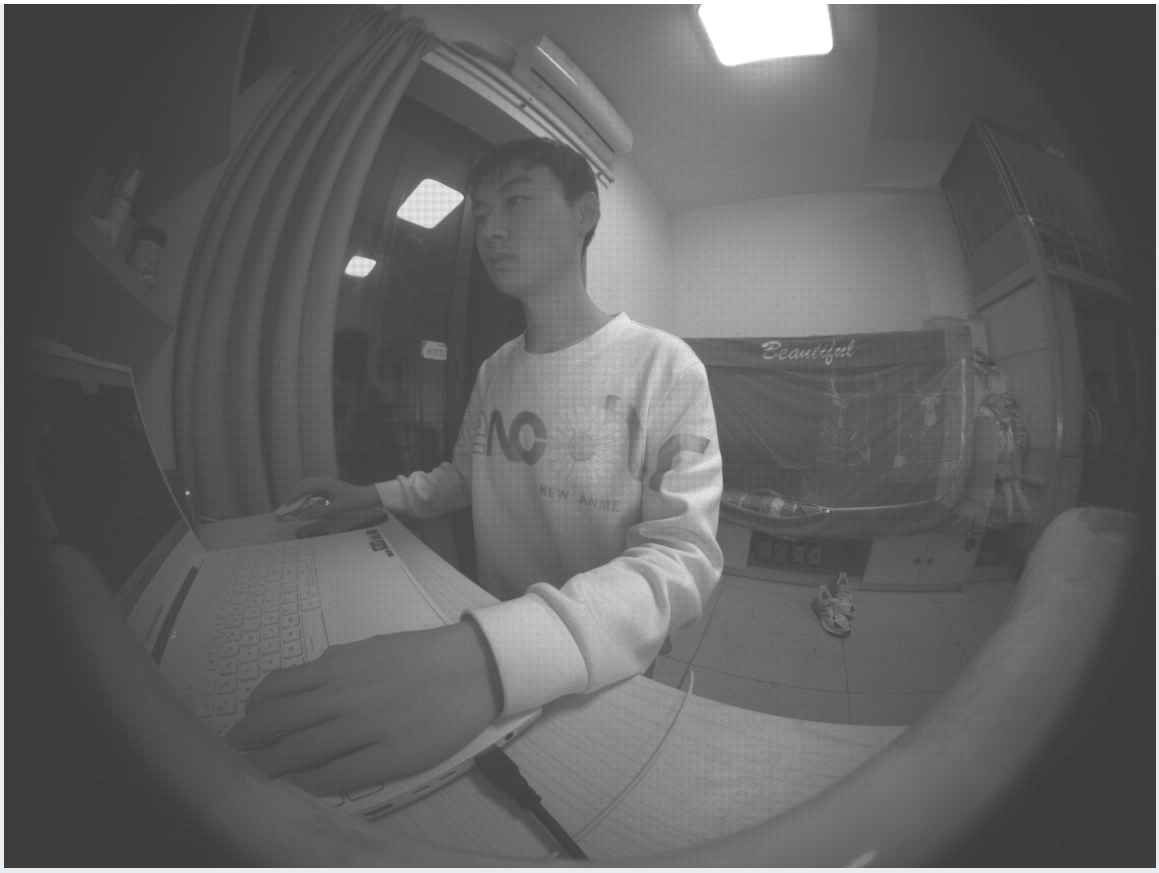
```
#  Using QCGP to do the AWB
        bayer = np.clip(bayer, 0, self.cfg.saturation_values.hdr)
        raw_r, raw_g, raw_b, raw_ir = get_rgbir_sub_array(bayer)
        #  Get mean value for each channel
        #  Noted that raw_r is the sub-array of the original picture
        mean_r = np.mean(raw_r) / 2
        mean_g = np.mean(raw_g) / 8
        mean_b = np.mean(raw_b) / 2
        mean_ir = np.mean(raw_ir) / 4
        mean_k = (mean_r + mean_g + mean_b + mean_ir) / 4

        #  Get the max value for each channel
        mask_r, mask_g, mask_b, mask_ir = get_mask_rgbir(bayer)
        RED = bayer * mask_r
        GREEN = bayer * mask_g
        BLUE = bayer * mask_b
        IR = bayer * mask_ir

        max_r = np.max(RED)
        max_g = np.max(GREEN)
        max_b = np.max(BLUE)
        max_ir = np.max(IR)
```

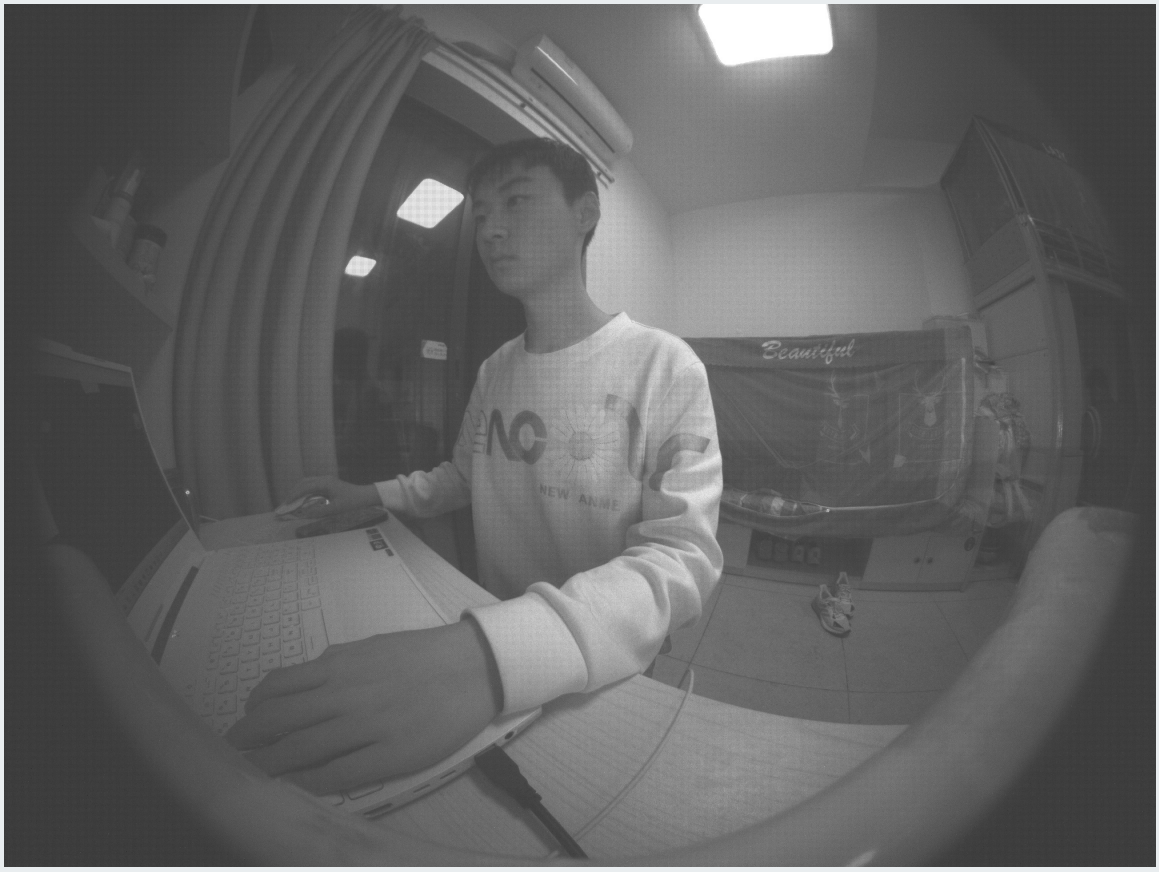
```
        max_k = (max_r + max_g + max_b + max_ir) / 4

        #  Get the uv for each channel
        u_red, v_red = self.getUV(max_r, max_k, mean_r, mean_k)
        u_green, v_green = self.getUV(max_g, max_k, mean_g, mean_k)
        u_blue, v_blue = self.getUV(max_b, max_k, mean_b, mean_k)
        u_ir, v_ir = self.getUV(max_ir, max_k, mean_ir, mean_k)



        m_RED = (u_red * RED * RED) + v_red * RED
        m_GREEN = (u_green * GREEN * GREEN) + v_green * GREEN
        m_BLUE = (u_blue * BLUE * BLUE) + v_blue * BLUE
        m_IR = (u_ir * IR * IR) + v_ir * IR

        m_RGBIR = m_RED * mask_r + m_GREEN * mask_g + m_BLUE *
mask_b + m_IR * mask_ir
        data['bayer'] = m_RGBIR.astype(np.uint16)
```

Chroma Noise Filtering

Result(No light, N0 AAF)

- Before
-



- After
-


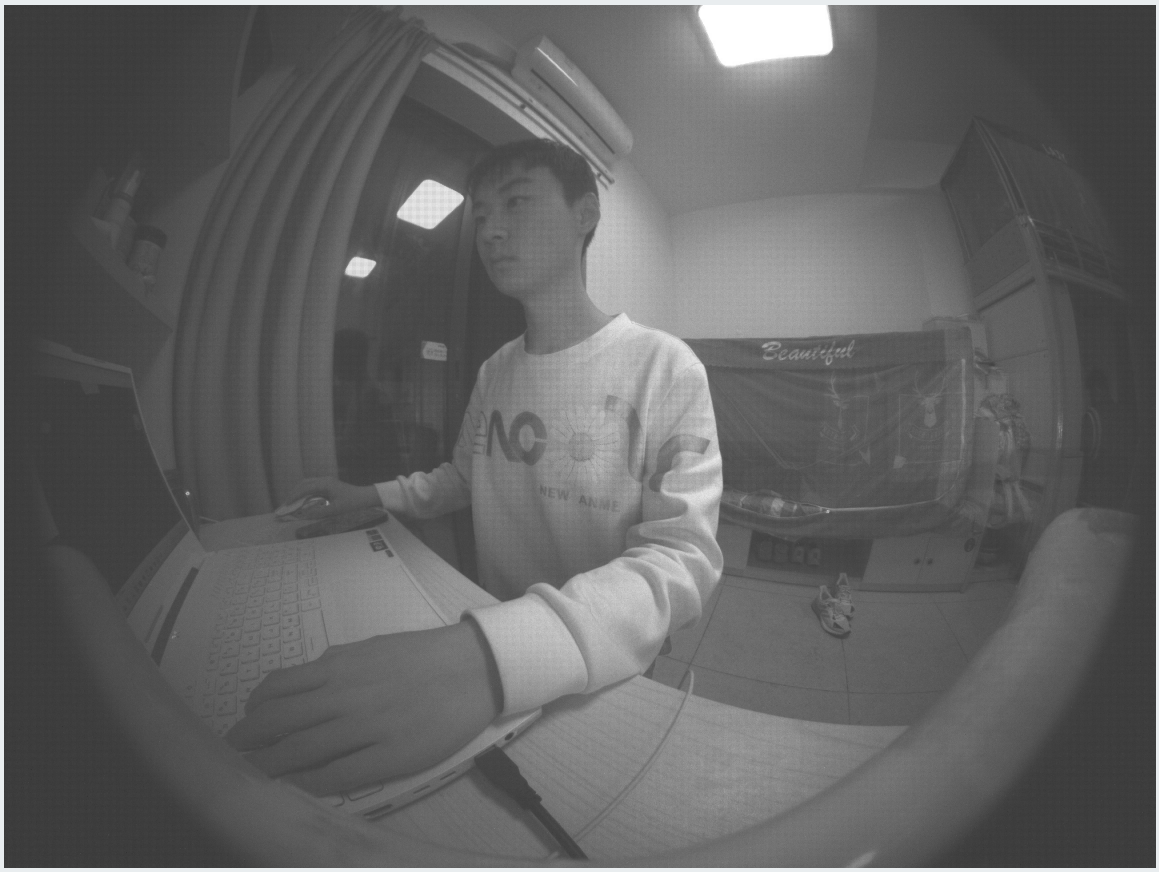
Idea

- Chroma Noise Filtering is a technique used to reduce or eliminate these noise artifacts specifically in the color channels of an image. The goal is to improve the overall image quality by preserving fine details and accurate color representation while minimizing the impact of chroma noise.
- The modification I made in this part is basically expand the original code to the `RGB-IR` pipeline by adding more calculation in cnc for more color components. No much novel modification made in this area.

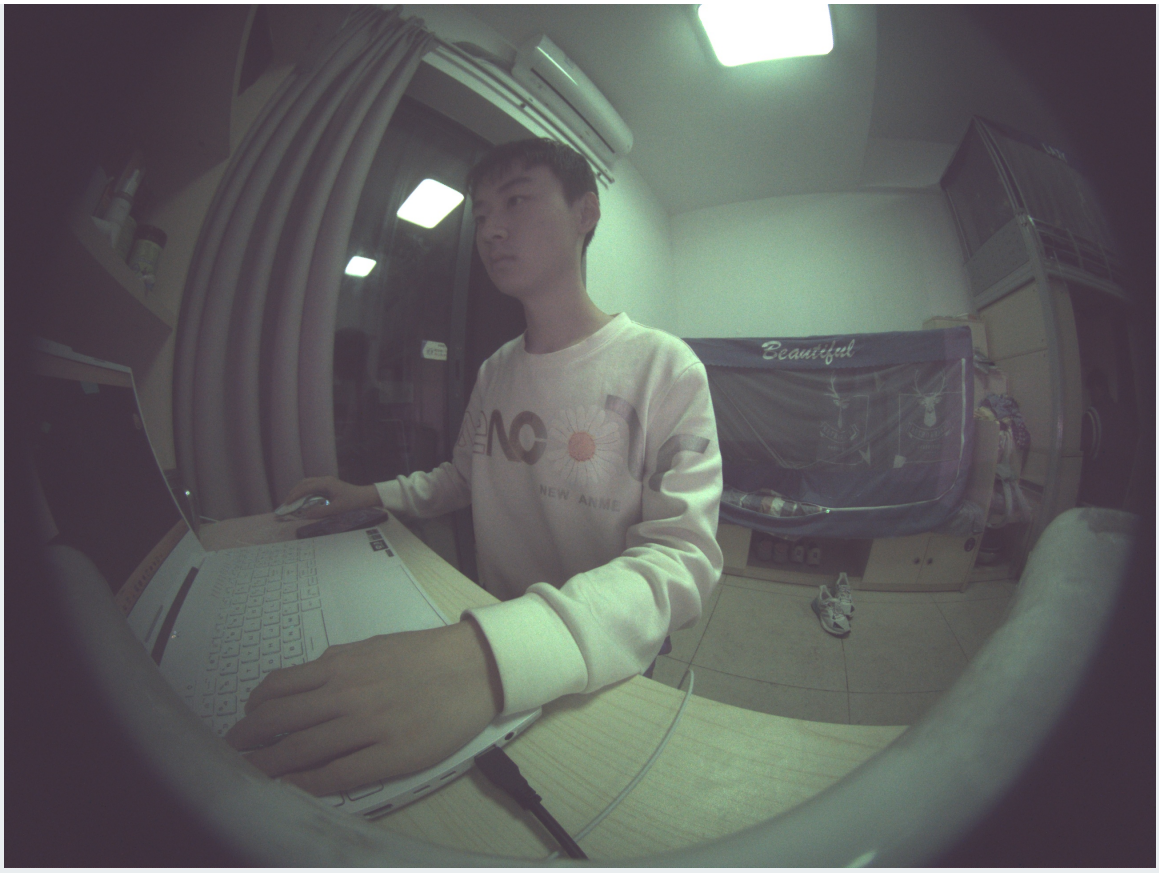
Color Filter Array

Result(No light, No AAF)

- Before



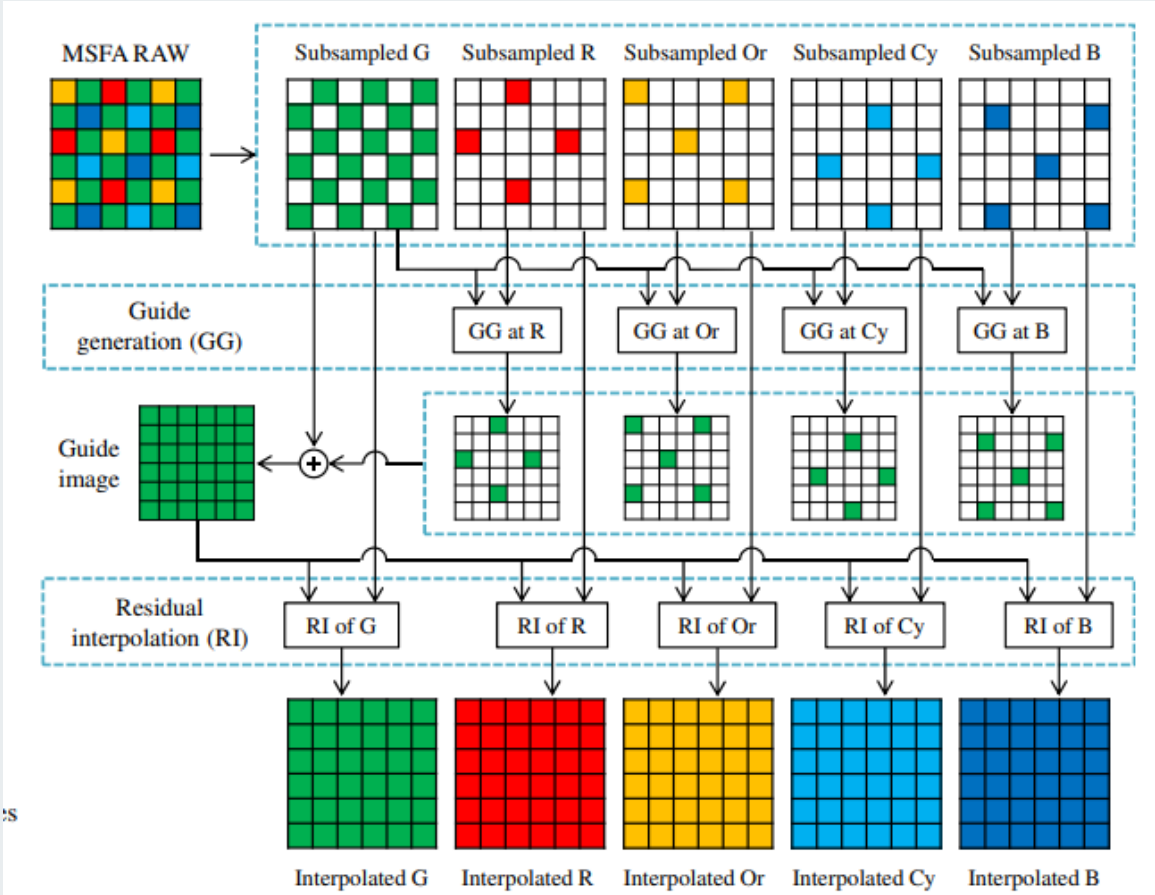
- After with IR channel added
-



Idea

- The purpose of the CFA is to capture color information by filtering incoming light into different color channels, typically red, green, and blue (RGB), which are the primary colors used to reproduce a full-color image.
- The CFA is a mosaic pattern of color filters placed over individual pixels on the image sensor. The most commonly used CFA pattern is the Bayer pattern, named after its inventor, Bryce Bayer. In the Bayer pattern, the sensor is divided into a grid of pixels, with each pixel covered by a color filter. The pattern consists of 50% green filters, 25% red filters, and 25% blue filters arranged in a repeating pattern.
- As stated before, the `RGB-NIR CFA pattern` would support the `RGB` and `IR` in one-shot acquisition. However, the additional `IR` pixels placed in a full pixel active area, it will cause the full pixel resolution to drop out half red and half blue pixels compared to a single RGB picture. This kind of mechanism will cause the `RGB` image performance to be low and requires a complex algorithm in the `ISP` to reproduce the `RGB` image.
- The essential process in `CFA` is the demosaicing algorithm used to reconstruct a full-color image from the incomplete color samples, demosaicing algorithms analyze the surrounding pixels' color information to estimate the missing color values at each pixel location, thereby reconstructing a full-color image.

CFA-Method-1

- This is based on the paper **MULTISPECTRAL DEMOSAICKING WITH NOVEL GUIDE IMAGE GENERATION AND RESIDUAL INTERPOLATION**
- The basic idea is to use the sub-sampled R, B, IR and the Guided Generation method to generate the Guidid Green image
- And then use this Guide image and residual interpolation, we can then get the interolated G, R, B, and IR channel image.
- The overall process can refer to this picture.
- 

(c) Overall flow of our proposed algorithm.
-

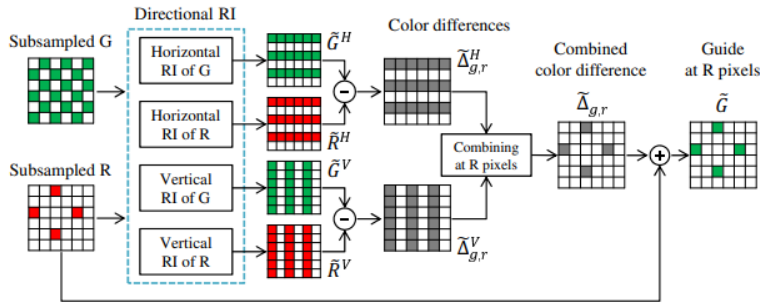


Fig. 2. Flowchart of the GG at the R pixels. The GG at the other pixels is similarly performed.

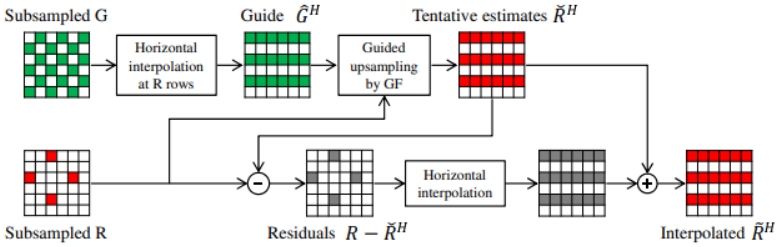


Fig. 3. Flowchart of the horizontal RI of the subsampled R band data in Fig. 2. The horizontal RI of G, the vertical RI of R, and the vertical RI of G are similarly performed.

- You may refer to the original [paper] to find more detailed implementation and discussion about this method. This is a very complicated method in terms of mathematical operation, So I will not discuss too much about it.
- The performance matrix of this method is as followed:
- **Table 1**. The average PSNR and CIEDE2000 [23] of all 16 scenes in the dataset.

	Algorithms	PSNR								CIEDE2000
		R	Or	G	Cy	B	sR	sG	sB	
3band	GBTF [21]	-	-	-	-	-	29.52	39.44	35.53	4.00
	LPA [22]	-	-	-	-	-	30.39	41.24	36.71	3.68
5band	BTES [11]	49.38	45.00	48.60	42.78	44.93	34.46	42.95	36.36	2.91
	AKU [1]	52.19	47.80	48.78	45.38	48.06	38.14	44.20	39.53	2.34
	GF [2]	53.12	51.06	49.61	47.94	48.89	40.75	45.73	40.51	2.06
	Proposed	**54.93**	**52.31**	**51.08**	**49.42**	**49.86**	**42.49**	**47.19**	**41.26**	**1.88**

- This method will generally took about 120 seconds to finish on my computer, I think this is too long to take for a image processing pipeline, So I went to find if there is any method that is faster then this one.

CFA-Method-2-Current

- This is based on the paper [Color Interpolation with Full Resolution for Hybrid RGB-IR CMOS Sensor]
- The basic idea behind this paper can be concluded easily.
- This method finds a way to reconstruct the missing Blue pixel and the missing Red pixel from the additional Red and IR pixel.

- For reconstructing the missing Blue pixel, we do:
-

(a)

$$B_{\text{to R center}} = \frac{B_t + B_l + B_r + B_d}{4}$$

- For reconstructing the missing red pixel, we do :
-

(a)

(b)

$$R_{Neg-oblique} = \frac{R_{right-up} + R_{left-down}}{2} \qquad R_{pos-oblique} = \frac{R_{left-up} + R_{right-down}}{2}$$

- After that, you got a raw image with simple bayer pattern `RGGB`.
- And then you can use the common Bayer CFA pattern demosacing algorithm to convert the `RGGB` into a full resolution `RGB` image.

- This method is superisingly simple and efficient with well-performed result
-

Image Test	Result			
	SNR	Color Accuracy	MTF	
	dB	*Delta-C00*	*Horizontal*	*Vertical*
Bayer	41.2	4.2	0.66	0.65
4x4 Hybrid	41.8	5.4	0.4	0.62

- Also, this method suites itself into the current pipeline seemlessly. After converting to the `RGGB` bayer pattern, the `Bilinear` or `Mavlar` method is used to reconstruct the `RGB` image for further processing.

IR

- Since the method-2 is using in the current implementation, the `IR` pixel is being interpolated into the `Red` channel, we need another way to extract and reconstruct a seperate `IR` image.
- This is then achieved by a simpler interpolation on the original raw image.

- 

Edge Enhancement

Result

- Edge detected
-

- After

- The Edge detection algorithm used here is a very simple one, namely a Sobel Filter in two directions.
- The original idea was to use the Canny Edge detector to do the EEH part, but the Canny Edge detector is time consuming and that the overall sharpness of the image is better with no AAF on.
- So the very simple and intuitive EEH method is used here to obtain a fast pipeline for real time application.

CCM

- Manually adjusted with respect to the color plate, the result is written into the `test.yaml`
- You can see two CCM matrix there, both can be good result
- The one commented is the one that will use IR