

facebook

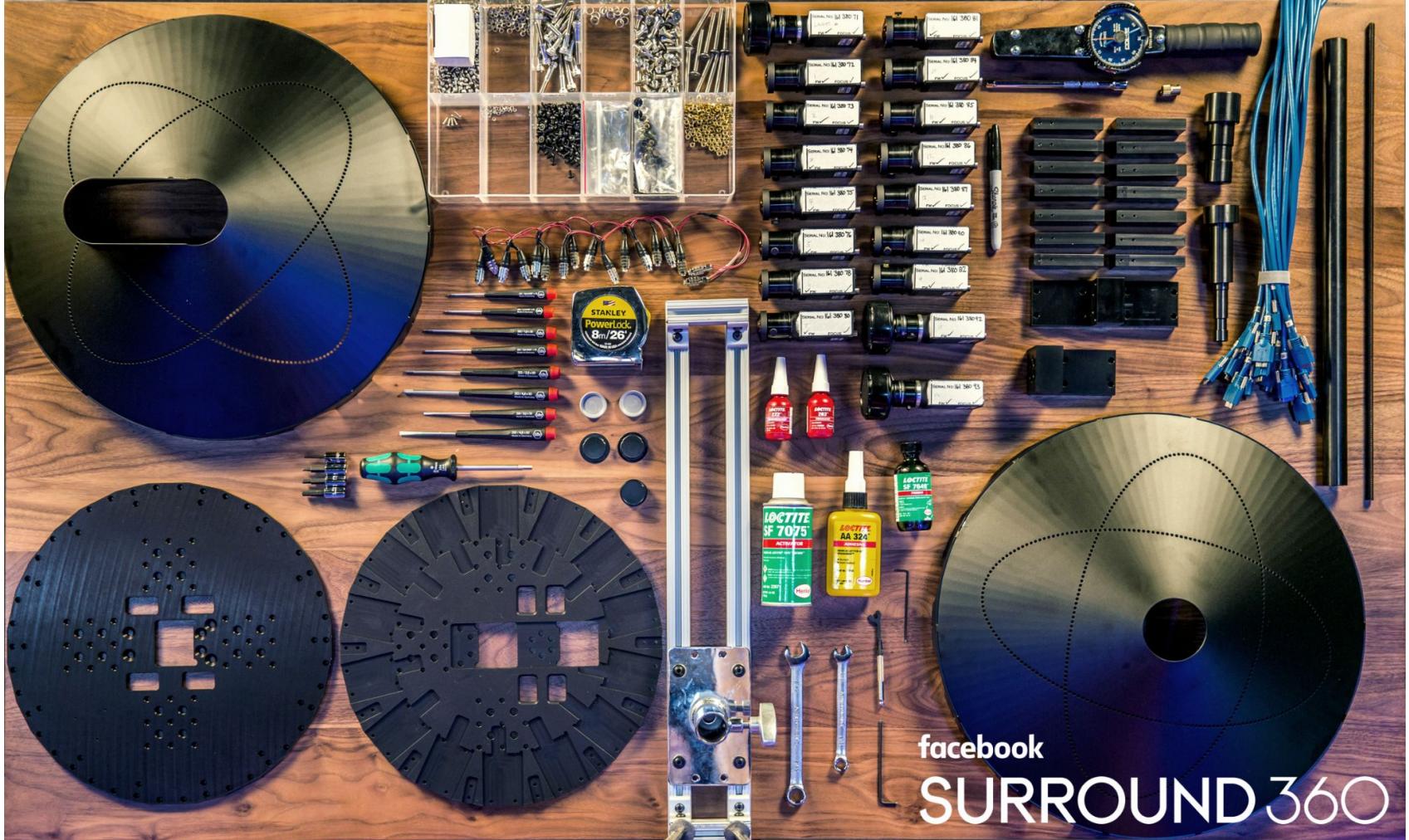
# Surround 360 분석

24.05.24

([https://docs.google.com/presentation/d/1oaHnHX\\_oYqIU59JIW7SJfTHKlIXrMoVyPU7n1FpnDo/edit?usp=sharing](https://docs.google.com/presentation/d/1oaHnHX_oYqIU59JIW7SJfTHKlIXrMoVyPU7n1FpnDo/edit?usp=sharing))

## 목차

- Surround360 System 개요
- Surround360 Build
- Sample Data 테스트
- Capture
- Color Calibration
- Optical Vignetting Calibration
- Geometric Calibration
- Rendering



# Surround360 System 개요

- Surround360은 VR에 적합한 고품질의 3D 360 비디오 캡쳐를 위한 하드웨어, 소프트웨어 시스템임

- 아래 3가지의 오픈소스로 구성됨

- surround360\_design/ (하드웨어 디자인)
  - 설계 도면 (blueprints)  
→ drawings/FB360\_V1\_ALL.pdf

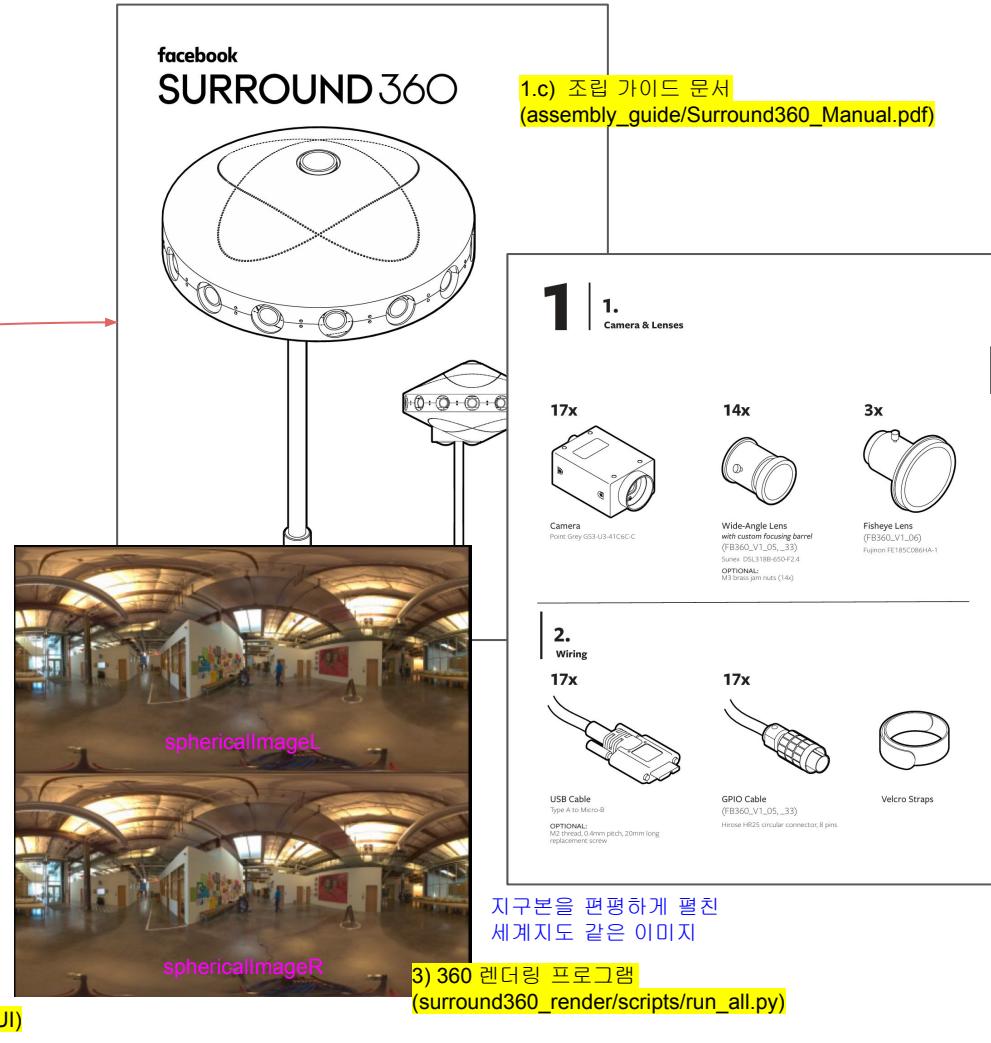
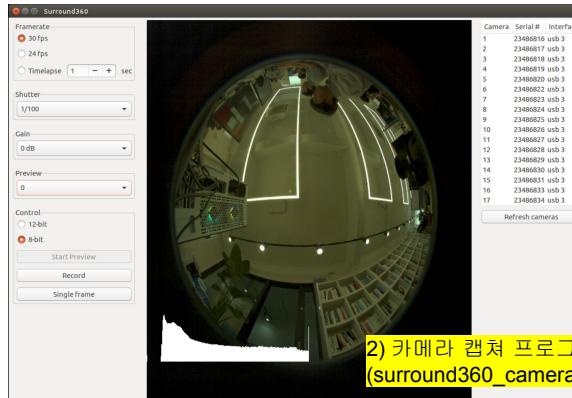
- b) 3D 모델 파일들(CAD)

→ 3d\_models/STEP\_files/FB360\_V1.STEP  
FB360\_V1\_21.STEP,  
...,  
FB360\_V1\_33.STEP

c) 조립 가이드 문서 (assembly instructions)  
→ assembly\_guide/Surround360\_Manual.pdf

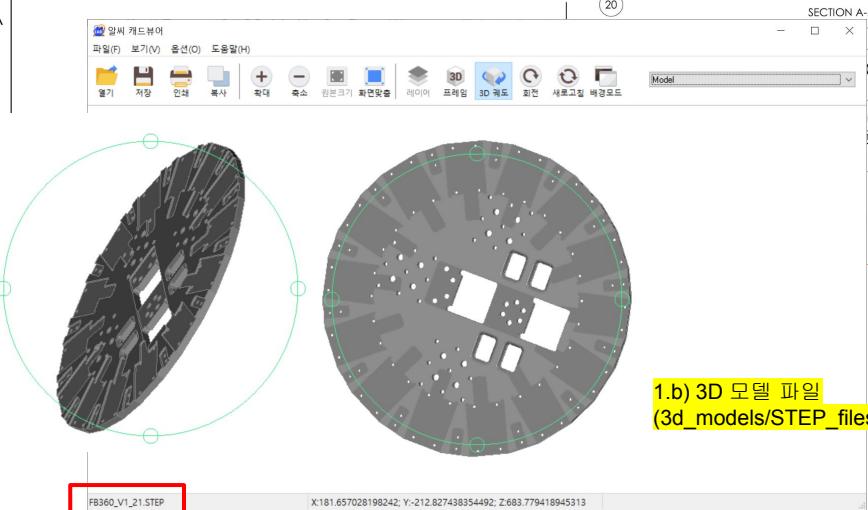
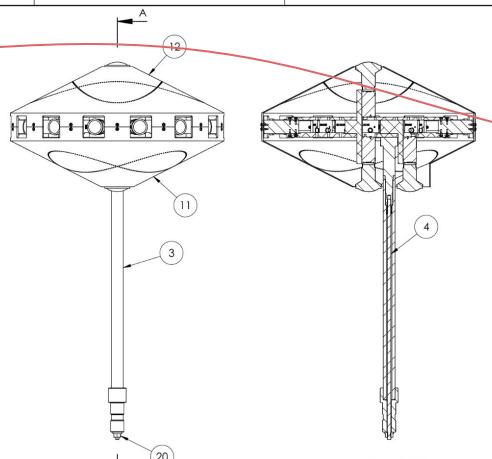
- surround360\_camera\_ctl\_ui (카메라 캡쳐 프로그램)  
raw data 캡쳐를 위한 리눅스 앱

- surround360\_render (360 렌더링 프로그램)  
캡쳐된 raw data를 VR 시청에 적절한 포맷으로 렌더링해주는 소프트웨어



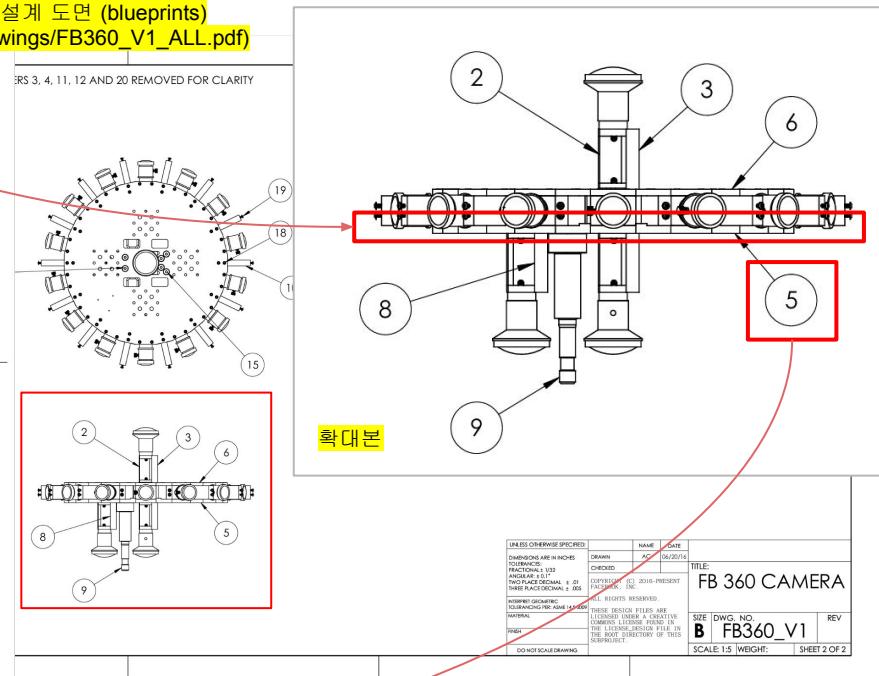
# Surround360 System 개요 (surround360\_design)

ITEM NO.	PART NUMBER	DESCRIPTION	Default/ QTY.
1	FB360_V1_05	HORIZONTAL CAMERA ASSEMBLY	14
2	FB360_V1_06	VERTICAL CAMERA ASSEMBLY	3
3	FB360_V1_07	EXTENSION TUBE ASSEMBLY	1
4	FB360_V1_08	THREADED ROD ASSEMBLY	1
5	FB360_V1_21	BASEPLATE	1
6	FB360_V1_22	TOP PLATE	1
7	FB360_V1_23	UPRIGHT	1
8	FB360_V1_24	CAMERA BRACKET	1
9	FB360_V1_25	POST	1
10	FB360_V1_28	SHELL SUPPORT	14
11	FB360_V1_29	BOTTOM COVER	1
12	FB360_V1_30	TOP COVER	1
13	90895A029	M6 LOCKWASHER	4
14	91292A142	M6 X 40 SHCS (18-8 SS)	4
15	92125A237	M6 X 14 FHCS (18-8 SS)	8
16	92125A252	M6 x 50 FHCS (18-8 SS)	2
17	90895A005	M3 LOCKWASHER	150
18	91292A111	M3 X 6 SHCS (18-8 SS)	150
19	92137A252	M3 X 10 FLANGED BHCS	28
20	90477A030	5/16 - 18 FLANGE NUT	1

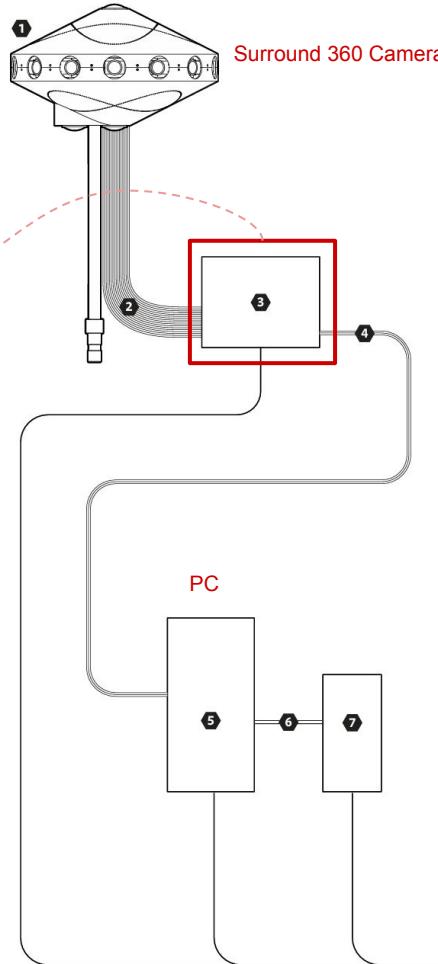
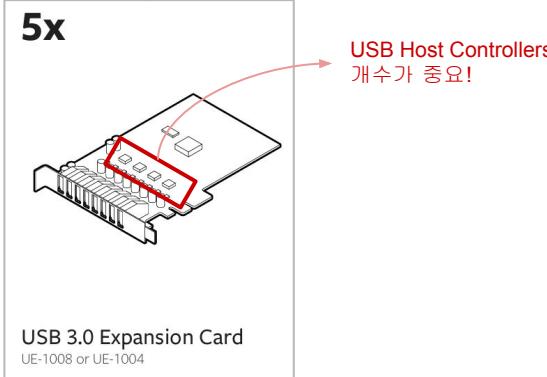
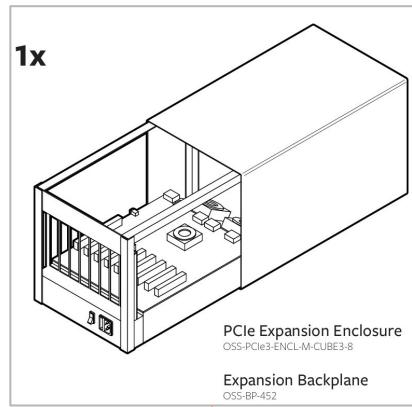


1.b) 3D 모델 파일  
(3d\_models/STEP\_files/FB360\_V1\_21.STEP)

1.a) 설계 도면 (blueprints)  
(drawings/FB360\_V1\_ALL.pdf)



# Surround360 System 개요 - 시스템 구성



## SURROUND 360 SYSTEM with Fiber Optic Extension

- ① Surround 360 Camera
- ② USB 3.0 High Speed Type A to Micro-B Cables x17
- ③ Fiber Optic Breakout Box
- ④ PCIe x8 Active Optical Cable
- ⑤ Lunchbox Computer (Computer)
- ⑥ SFF-8644 to SFF-8644 MiniSAS Cable x2
- ⑦ Raid Tower
- ⑧ Power Supply

### ① SURROUND 360 Camera

- 17x Point Grey Grasshopper Cameras
- 14x Wide-Angle Lenses
- 3x Fisheye Lenses
- GPIO Trigger Cable

### ③ FIBER OPTIC BREAKOUT BOX

- 1x Backplane With Power Supply
- 5x PCIe x4, 4 Ports USB 3.0 Expansion Card
- 1x PCIe x16 Host Interface Card

### ⑤ COMPUTER SPECIFICATIONS

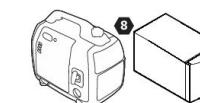
- Intel Core i7-5960X Haswell-E 8-Core 3.0 GHz LGA 2011-v3\*
- GIGABYTE GA-X99P-SLI (rev. 1.0) LGA 2011-v3 Intel X99 Motherboard\*
- 8GB DDR4 2400 288-PIN Memory ± (64GB of Memory Installed)\*
- CPU COOLING FAN FOR LGA 2011-v3\*
- 1GB NVIDIA PCIe x16 VIDEO CARD\*
- 700 WATT POWER SUPPLY\*
- 2.5" 128GB SSD\*
- OPERATING SYSTEM - UBUNTU 14.04 LTS†

### ⑦ RAID TOWER SPECIFICATIONS

- 8 x 1TB SSD RAID: 1 Hour of Continuous Raw Video Capture  
(Bandwidth: 2.1 GB/s)

### ⑧ POWER SUPPLY

- 110V AC
- 350W max.
- Alternative Options: UPS Backup Battery or Quiet Generator

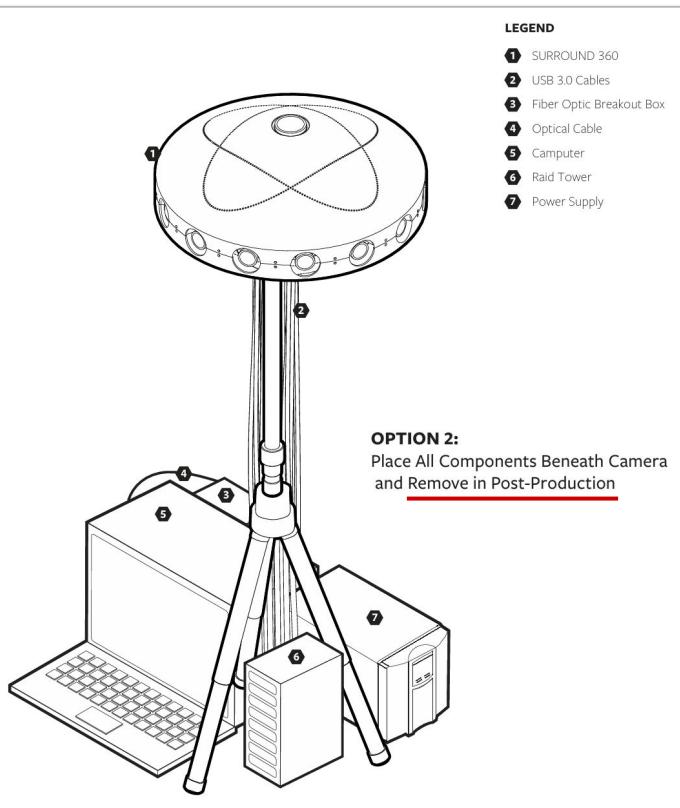
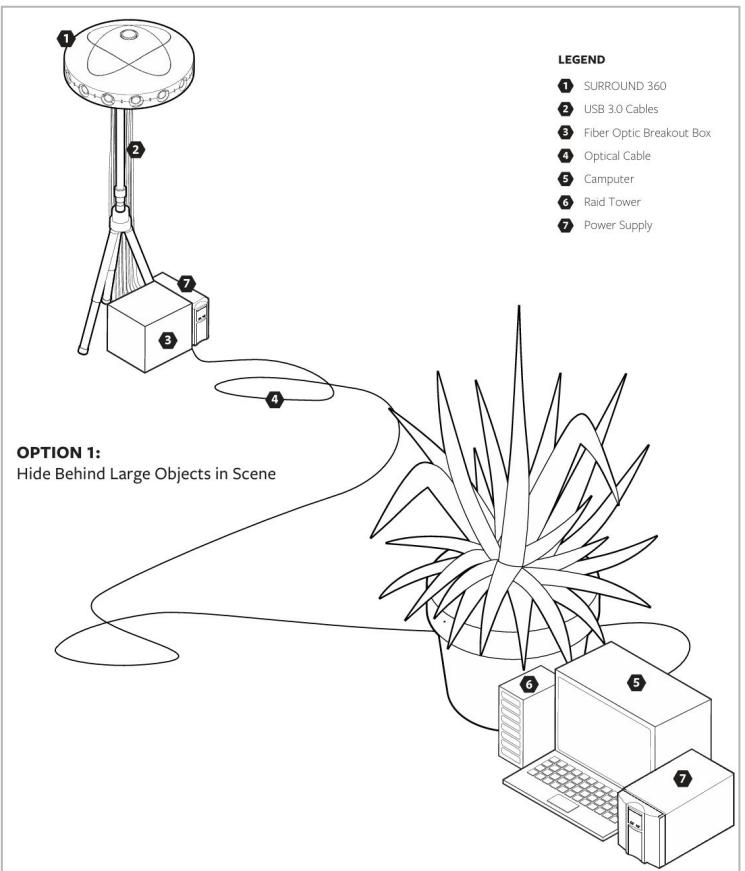


\*Recommended  
†Required

[surround360\\_design/assembly\\_guide/Surround360\\_Manual.pdf](http://surround360_design/assembly_guide/Surround360_Manual.pdf)

# Surround360 System 개요 - 촬영용 배치

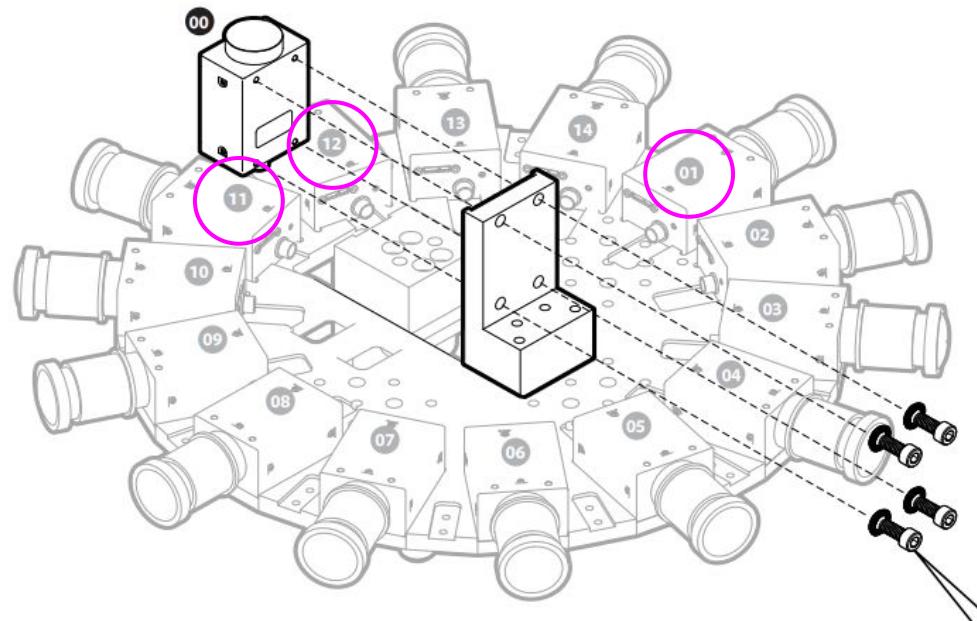
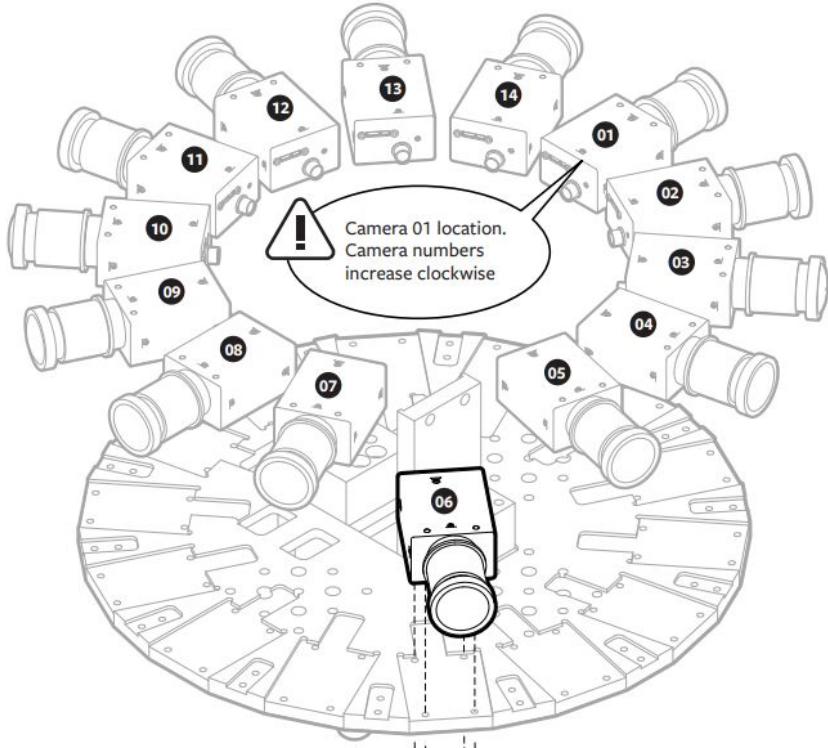
[surround360\\_design/assembly\\_guide/Surround360\\_Manual.pdf](surround360_design/assembly_guide/Surround360_Manual.pdf)



Surround 360 의 배치안 (1)

실제 구성  
(2048 x 2048, 15~20fps)

## Surround360 System 개요 - 카메라 배치



Top 카메라의 좌측에 첫번째 Side 카메라(01)가 위치하고 있음.

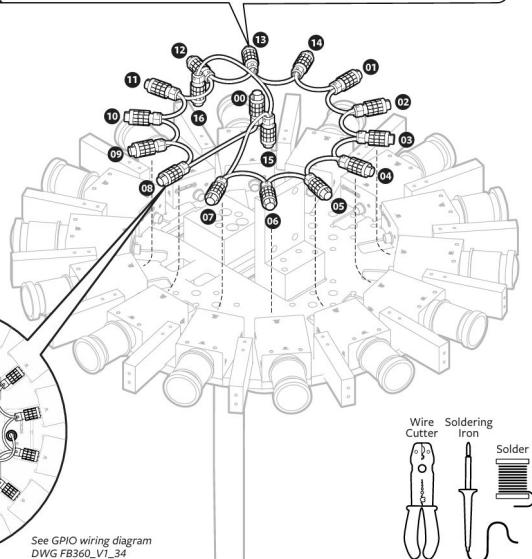
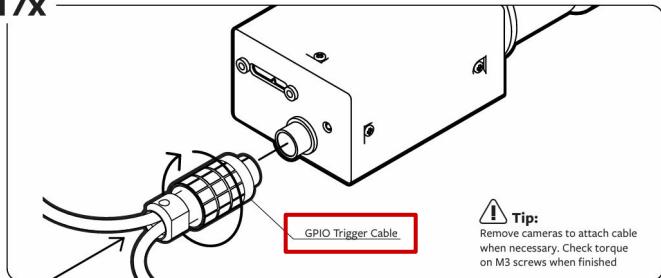
카메라 위치가 안 맞으면 스티칭이 안맞게 생성됨

3

12.

GPIO Trigger Cable

17x



### What is Synchronized Capture? (동기화 캡쳐)

- 다수의 카메라가 동시에 **exposing**을 시작하는 것을 의미함 (동시는 수 microseconds 이내를 의미)
- primary camera** 의 **strobe** 를 이용해 1대의 **primary camera** 로 다수의 **secondary** 카메라에서 동시에 캡쳐가 가능함 (이미지 캡쳐 시작시에 **strobe** 가 발생함)
- external hardware trigger** 를 사용하는 방법도 존재하지만 여기서 다루지는 않음

### 카메라 간의 physical link 설정 방법

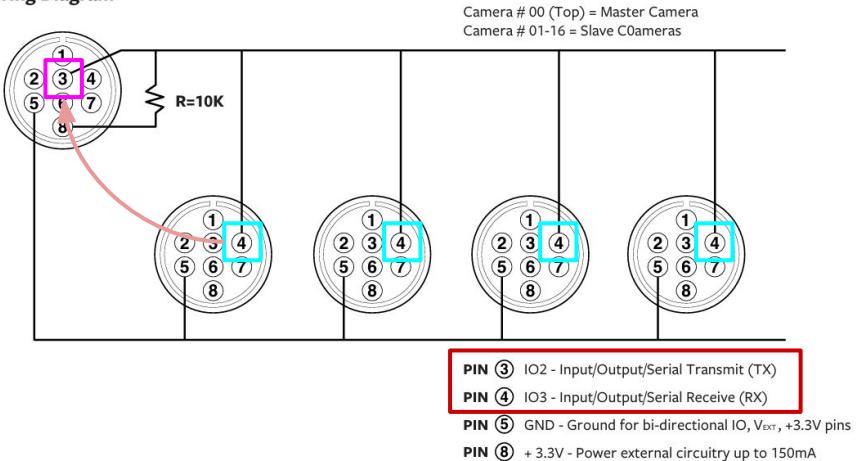
Grasshopper3 (GS3) 카메라의 경우 GPIO Trigger Cable 연결 방법

- primary camera** 빨간선(3번, output) + **secondary camera** 초록선(4번, input)
- primary** 갈색선(5번, ground) + **secondary** 갈색선(5번, ground) // 접지용

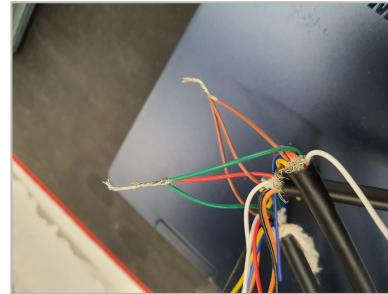
# Surround360 System 개요 - GPIO 케이블 연결

[surround360\\_design/assembly\\_guide/Surround360\\_Manual.pdf](#)

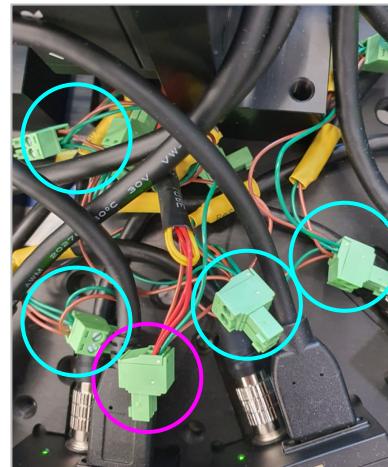
## A - Wiring Diagram



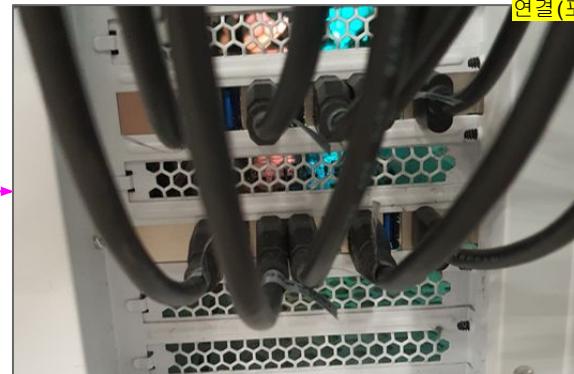
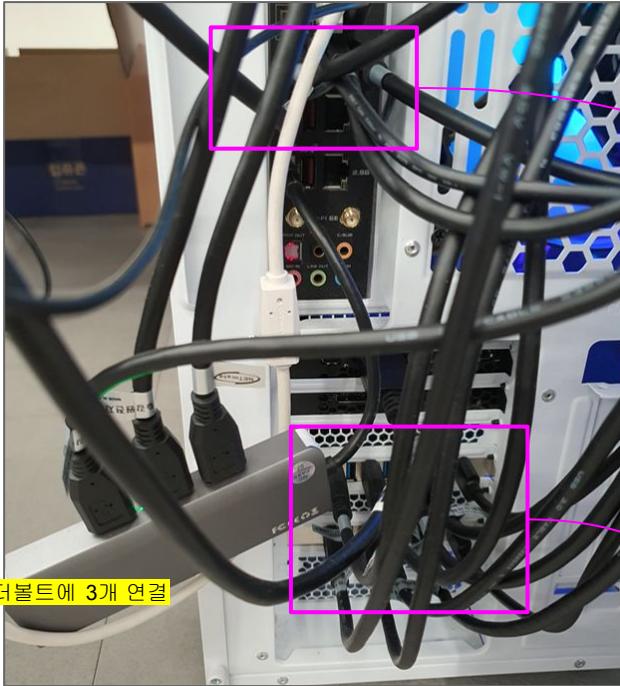
초기 예상 연결 방식



최종 연결 방식



## Surround360 System 개요 - USB 케이블 연결

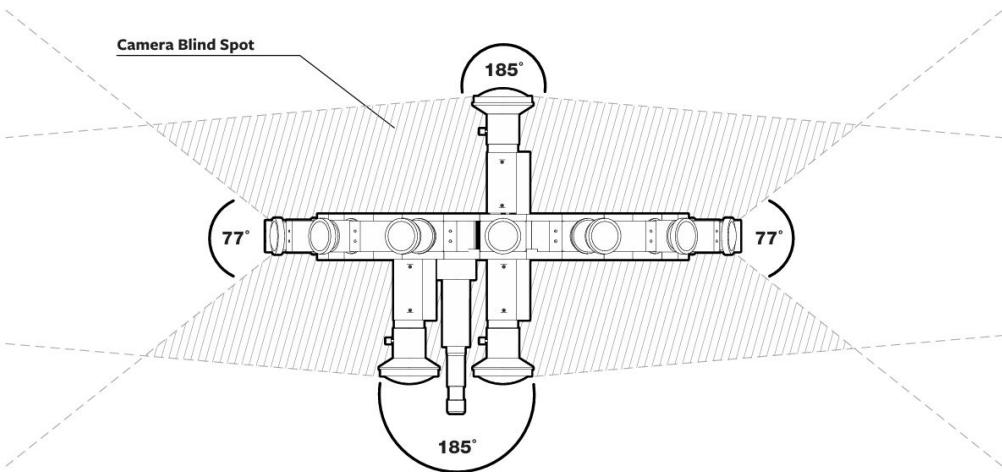


[TIP] usb 포트 연결 조합을 잘 맞춰줘야 17대까지 인식에 성공함  
usb 연결이 실패하는 카메라 발생시(빨간색등 점등시) usb 를 뺏다 꽂아본다.

**4**

**2a.**

## Blind Spot for Mounting Accessories



Point Grey GS3-U3-41C6C-C



해상도	2048 × 2048
메가픽셀	4.1
최대 프레임 속도 표준	90
화소 크기 // 센서크기 $2048 \times 5.5 \mu\text{m} = 11,264 \mu\text{m}$ 약 11 mm(1.1 cm)	5.5 $\mu\text{m}$
렌즈 마운트	C-마운트
센서 유형	CMOS
센서 모델	CMOSIS CMV4000-3E5
데이터 인터페이스	USB 3.1 Gen 1
스펙트럼	색상
부품 번호	GS3-U3-41C6C-C

## Surround360 System 개요 - 렌즈 스펙

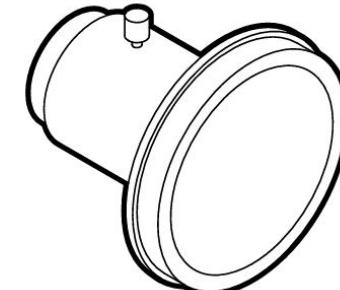
	1) Sunex DSL318B-650-F2.4 (Wide-Angle Lens)	2) Fujinon FE185C086HA-1 (Fisheye Lens)
Mount	CS	C
Focal Length	7.01 mm	2.7 mm
Focus Type	Fixed (focal)	Fixed (focal)
Iris Range	N/A	F1.8 - F16
Iris	N/A	Manual
Back Focal Distance	N/A	9.75 mm
Exit Pupil Position	N/A	-49 mm
Angular Field of View	142° // Field of View Horizontal	185° × 140°35'
Focusing Range	0.2 m	0.2 m
Sensor size	1 inch	N/A
Aperture	2.4	N/A
Image Circle [mm]	16	N/A

14x



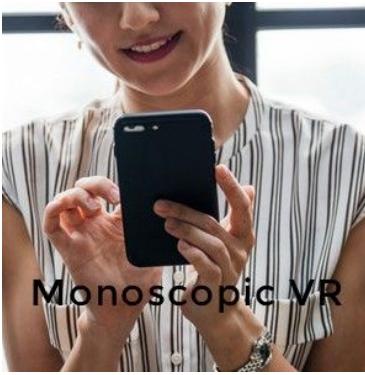
1) Sunex DSL318B-650-F2.4 (Wide-Angle Lens)

3x



2) Fujinon FE185C086HA-1 (Fisheye Lens)

# Surround360 System 개요 - VR 종류

Monoscopic VR (Virtual Reality)	Stereoscopic VR
 <p data-bbox="88 530 399 584">구좌표계에 투영된 1장의 flat 한 이미지로 구성됨</p>  <p data-bbox="501 516 827 551">Monoscopic VR</p>	 <p data-bbox="961 616 1288 670">보통 Top/Bottom 3D format 으로 구성됨</p>  <p data-bbox="1374 516 1720 551">Stereoscopic VR</p>

- 좀더 현실적이고 몰입감 넘치는 경험을 제공(by depth perception)
- VR 헤드셋으로만 시청이 가능함
- 역동적 움직임이 자연스럽게 느껴지도록 하는 것이 매우 어려움(e.g, sports and festivals)
- 제작이 훨씬 어려움(제작시의 작은 결점(flaws)이 품질에 큰 영향을 줌)
- Filming 과 post production 단계에서 좀더 주의가 필요함
- 카메라와 5 피트(1.5미터) 이상 20 피트(6미터) 이내여야 3D 효과가 발생됨
- Monoscopic VR 컨텐츠도 시청 가능

## Surround360 System 개요 - VR 도전 과제

### The challenges of VR capture (도전 과제):

- 오픈돼 있는 3D-360 video camera 시스템이 존재하지 않음
- 기존 시스템들은 카메라가 과열되거나, 마운트가 견고하지 않고, 수동 스티칭으로 인해 스티칭 시간이 오랜 걸리는 단점
- Monoscopic 360 은 360 scene 을 캡쳐하기 위해 2대나 그 이상의 카메라만 필요(depth 없이 flat 한 이미지 획득)
- 반면 3D-360 video(Stereoscopic 360) 는 depth 획득을 위해 사람의 눈 간격(6.4 cm)에 맞춘 (모든 방향을 향하는) 10 ~ 20 대의 카메라 캡쳐가 필요(VR 헤드셋으로 시청 가능)
- 동시에 30 ~ 60 FPS 캡쳐가 필요(**global shutter** vs **rolling shutter**)
- 2GB/s USB bandwidth 가 필요  
 $4MB * 17대 * 30fps = 2GB/s$
- 각 카메라의 이미지들을 1개의 **seamless video** 로 스티칭 필요



# Surround360 빌드

공식 오픈소스인 [github.com/facebookarchive/Surround360](https://github.com/facebookarchive/Surround360) 하위의 surround360\_render/README.md, surround360\_camera\_ctl\_ui/README.md 문서를 참고하여 아래 문서에 빌드 절차를 정리함

[https://github.com/CUBOX-Co-Ltd/Surround360/blob/main/cubox\\_Surround360\\_build\\_240416.pdf](https://github.com/CUBOX-Co-Ltd/Surround360/blob/main/cubox_Surround360_build_240416.pdf)

## [문제 사항]

우분투 16.04 는 공식 지원이 끊긴 상태라서

1) 소스 repository 가 사라져 버린 경우가 존재함

(e.g. python-wxgtk2.8 는 apt repository 가 사라져 에러가 발생하므로 강제로 repository 주소를 예전 주소로 변경해줘야 성공함)

2) 소스 repository 가 이동된 경우가 존재함

LLVM 빌드시 svn repository 를 github 로 옮기면서 기존 빌드 명령이 실패함  
(루트 CMakeLists.txt 가 사라짐). 예전 비슷한 시점의 소스

(llvm-3.7.0.src.tar.xz) 를 찾아 빌드해서 넘어감

3) python 버전 불일치로 파이썬 소스 수정이 필요함

python 소스에서 print “xxx” 로 된 부분을 전부 print(“xxx”) 로 수정해줘야 함  
개발환경 설정에서 Gooey 에서는 print(“xxx”) 형태를 사용중이므로  
python 2.x 버전이 아니, 3.x 대 버전으로 설정해줘야 하므로

4) vscode 도 예전 버전을 사용해야 Open folder 및 디버깅이 가능함  
(code\_1.64.2-1644445741\_amd64.deb)

디풀트 color calibration 인 경우  
(surround360\_render\res\config\is\p\cmosis\_sunex.json)



color calibration 를 진행한 경우



# Sample Data 테스트

샘플 데이터를 아래 2가지 형태로 제공하고 있음(from Surround360/README.md)

## Sample Data (1)

bin 파일에서 unpack 한 RAW 개별 이미지(.bmp)와 ISP 처리된 결과 이미지

We provide a sample dataset for those who are interested in testing the rendering software without first building a camera.

\* "Palace of Fine Arts Take 1" - 2 frames - (337.4MB)

\* Raw data: [http://surround360.hacktv.xyz/sample/sample\\_dataset.zip](http://surround360.hacktv.xyz/sample/sample_dataset.zip)

\* Sample result: [https://s3-us-west-2.amazonaws.com/surround360/sample/sample\\_result.zip](https://s3-us-west-2.amazonaws.com/surround360/sample/sample_result.zip)

\* NOTE: The Surround 360 hardware and camera control software records to a RAW binary format which needs to be unpacked to get to the individual images. In this smaller dataset, the 'unpack' and 'arrange' steps of the pipeline have already been run (see run\_all.py), so you do not need to run them again.

→ 단순히 raw 와 isp\_out 이미지 확인용 파일들임

## Sample Data (2)

bin 파일들과 최종 렌더링된 결과 파일들임

\* "Facebook Building 20" - 190 frames - (21.15GB)

\* Binary file 1: [\(10.76GB\)](https://s3-us-west-2.amazonaws.com/surround360/github_samples/test/0.bin)

\* Binary file 2: [\(9.56GB\)](https://s3-us-west-2.amazonaws.com/surround360/github_samples/test/1.bin)

\* Sample result:

[\(823.6MB\)](https://s3-us-west-2.amazonaws.com/surround360/github_samples/test/render.zip)

\* NOTE: the render directory contains calibrated config files and two rendered frames

\* The file NOTES.txt contains sample commands to process the binaries

Sample Data (2) 의 경우,

~/Desktop/test 폴더를 생성 후, 0.bin, 1.bin, render/ 를 위치시킨 후, 아래 명령어를 실행하면 ~/Desktop/test/render/eqr\_frames/eqr\_000000.png 와 같은 최종 결과 이미지가 생성된다.

```
python ~/cubox/Surround360/surround360_render/scripts/run_all.py \
--data_dir ~/Desktop/test \
--dest_dir ~/Desktop/test/render \
--start_frame 0 \
--frame_count 30 \
--quality 6k \
--steps_unpack \
--steps_render \
--steps_ffmpeg \
--enable_top \
--enable_bottom \
--enable_pole_removal \
--cubemap_format video \
--cubemap_height 0 \
--cubemap_width 0 \
--verbose \
--save_raw \
--save_debug_images
```

### [Fix Error]

print "Terminating process: " 에서 에러가 발생하면 print("Terminating") 식으로 소스를 수정

ValueError: can't have unbuffered text I/O 에러가 발생하므로  
open(dest\_dir + "/runtimes.txt", 'w', 0) 에서 마지막 0 인자를 제거

## Sample Data (1)

"Palace of Fine Arts Take 1" - 2 frames - (337.4MB)

→ 단순히 raw 와 isp\_out 이미지 확인용임

[sample\_dataset.zip]

```
├── cube_frames (이하 전부 빈폴더)  
├── eqr_frames  
└── logs  
├── pole_masks  
├── raw  
├── single_cam  
└── vid
```

1st frame

```
000000  
├── flow  
├── flow_images  
└── isp_out
```

```
    └── cam0.png  
    ├── cam1.png  
    ├── ...  
    └── cam16.png
```

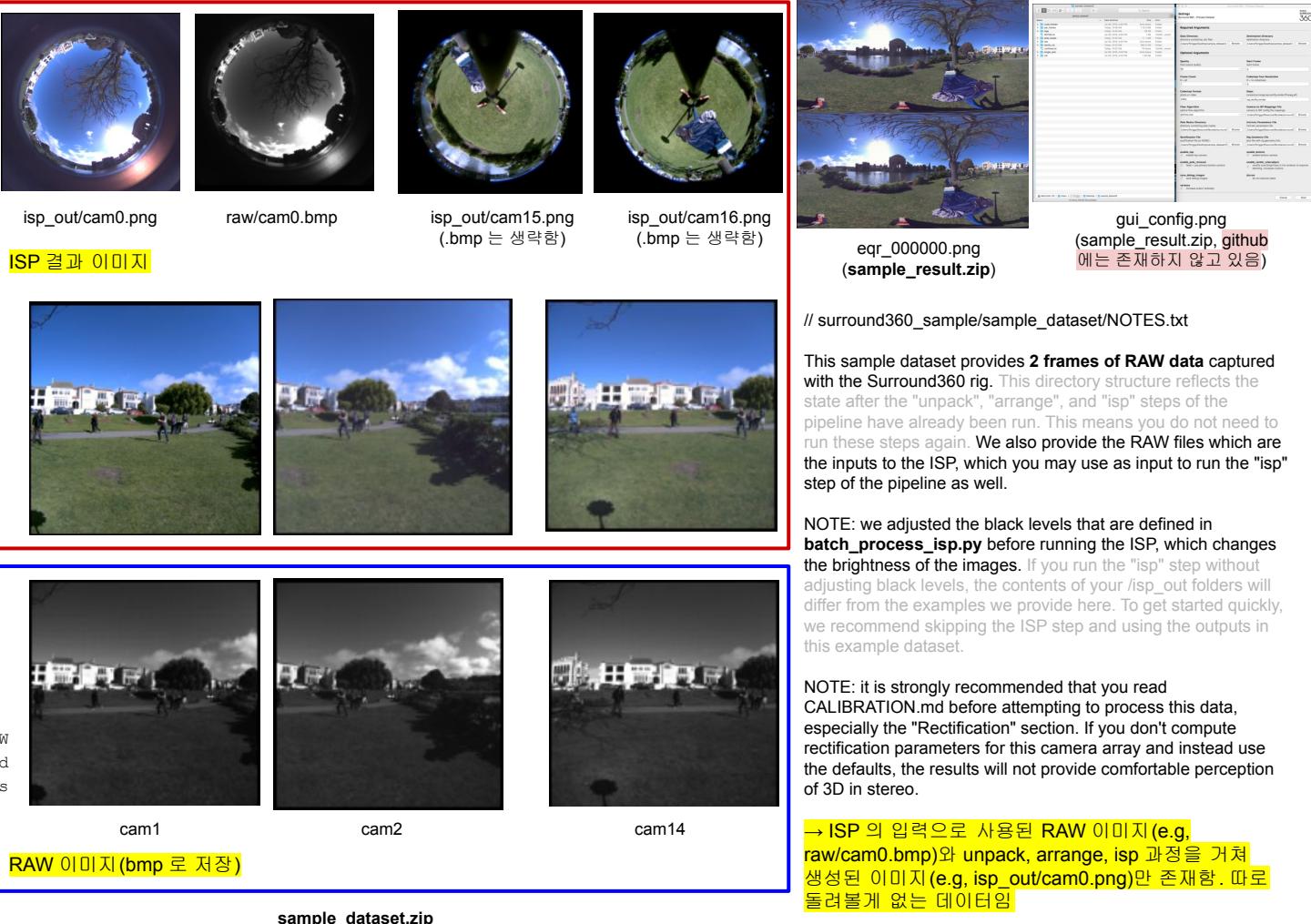
```
└── projections
```

```
raw  
├── cam0.bmp  
├── cam1.bmp  
├── ...  
└── cam16.bm
```

```
000001  
└── 2nd frame
```

// Surround360/NOTES.txt

NOTE: The Surround 360 hardware and camera control software records to a RAW binary format which needs to be unpacked to get to the individual images. In this smaller dataset, the 'unpack' and 'arrange' steps of the pipeline have already been run (see run\_all.py), so you do not need to run them again.



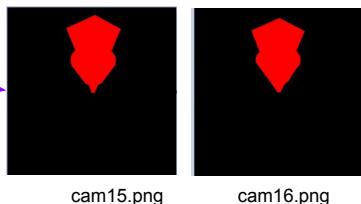
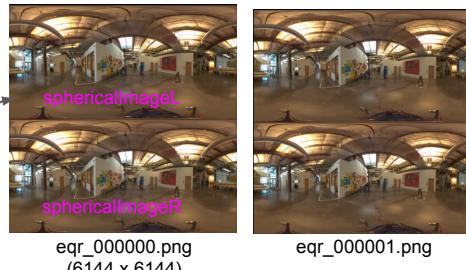
## Sample Data (2) ROOT\_DIR

"Facebook Building 20" - 190 frames - (21.15GB)

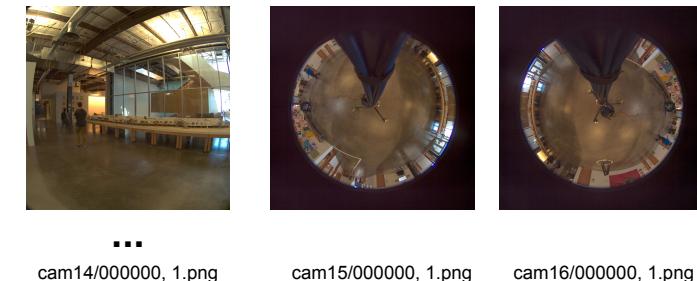
```
[~/Desktop/test]
└── render (823.6MB)
    ├── config
    │   ├── isp
    │   │   ├── 15636976.json
    │   │   ├── 16237674.json
    │   │   ...
    │   │   └── 16283829.json
    │   ├── camera_rig.json
    │   └── eqr_frames
    │       ├── eqr_000000.png
    │       └── eqr_000001.png
    ├── pole_masks
    │   ├── cam15.png
    │   └── cam16.png
    └── rgb
        ├── cam0
        │   ├── 000000.png
        │   └── 000001.png
        ├── cam1
        ├── cam2
        ...
        └── cam16
```

NOTE: the render directory contains calibrated config files and two rendered frames

The file NOTES.txt contains sample commands to process the binaries



render.zip 압축 해제시  
캘리브레이션된 config 파일 + .bin +  
pole\_masks 등이 존재함



run\_all.py 실행시 --save\_raw 옵션 존재시 ~/Desktop/test/render/raw 폴더도 생성됨 (e.g. 000000.tif)

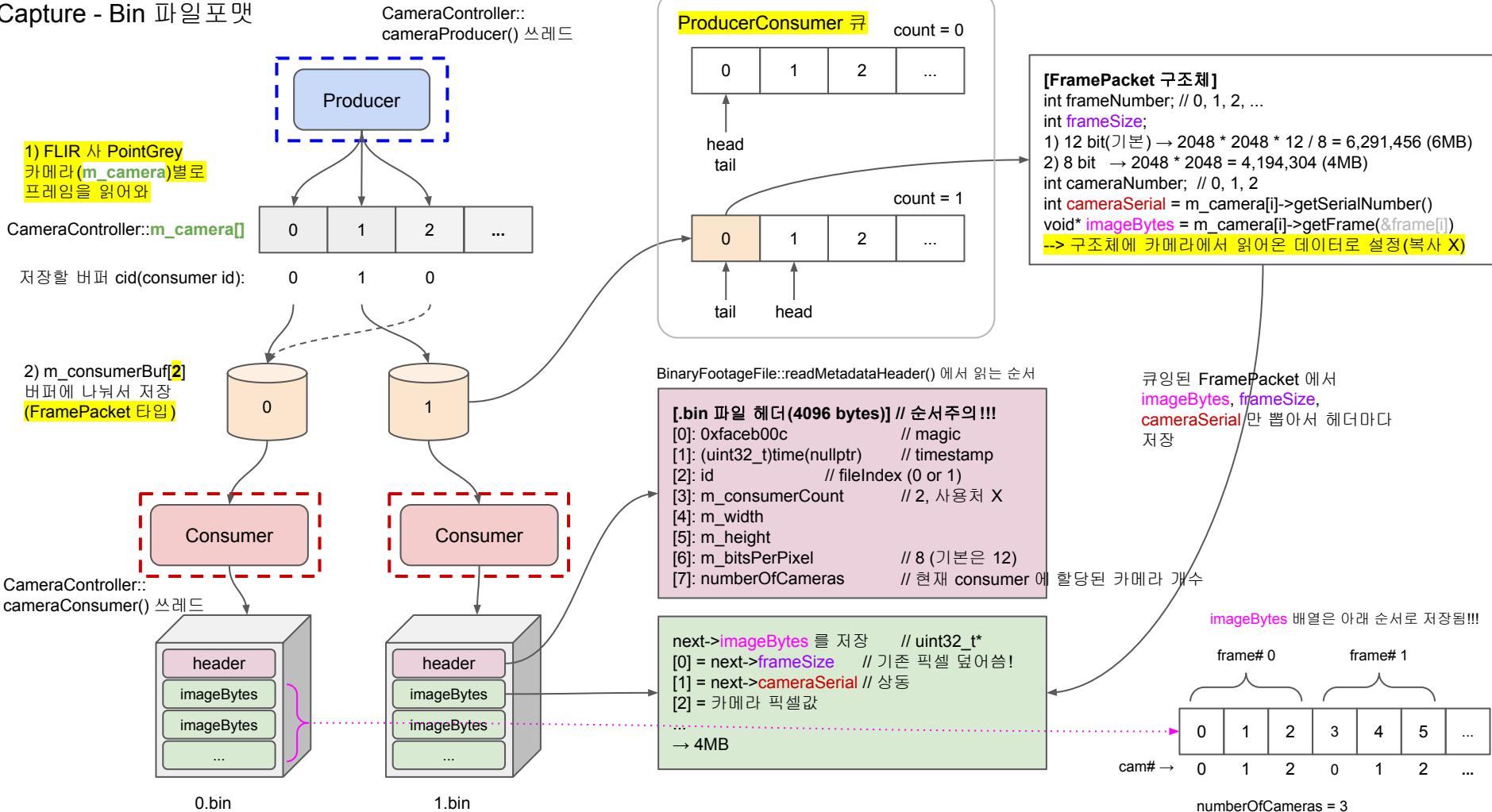
도중에 ~/Desktop/test/rgb/15636976 폴더가 생성되지만, Unpacker에서 최종적으로 ~/Desktop/test/rgb/cam0 폴더로 rename 시킴

기존의 rgb/cam0/000000.png 등의 output 파일들은 run\_all.py 실행시 새로 생성이 되지만, 비교용으로 들어 있는 것으로 보임

~/Desktop/test/render/eqr\_000000\_stereo\_comparison.gif 와 test\_259927\_6k\_TB.mp4 등의 동영상 파일도 생성된다.  
<https://drive.google.com/drive/folders/19J00fk-2OTmCEzIFGarrUgsn9zEmSF9U>

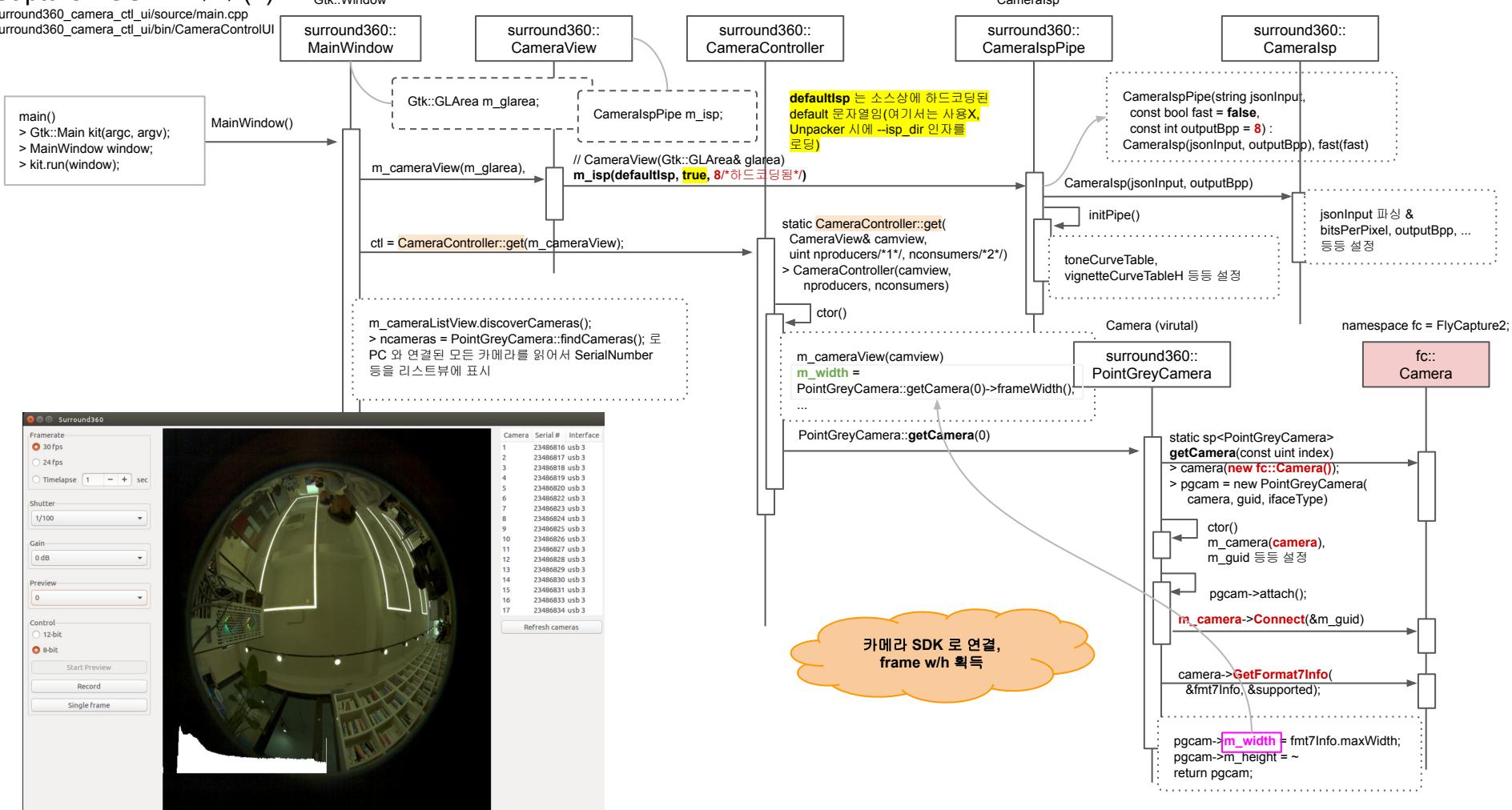
두 번째 실행시에는 raw/ 폴더를 지우고 진행해야 한다.  
(raw/ 폴더가 이미 존재시 에러 발생함!!!)

## Capture - Bin 파일 포맷

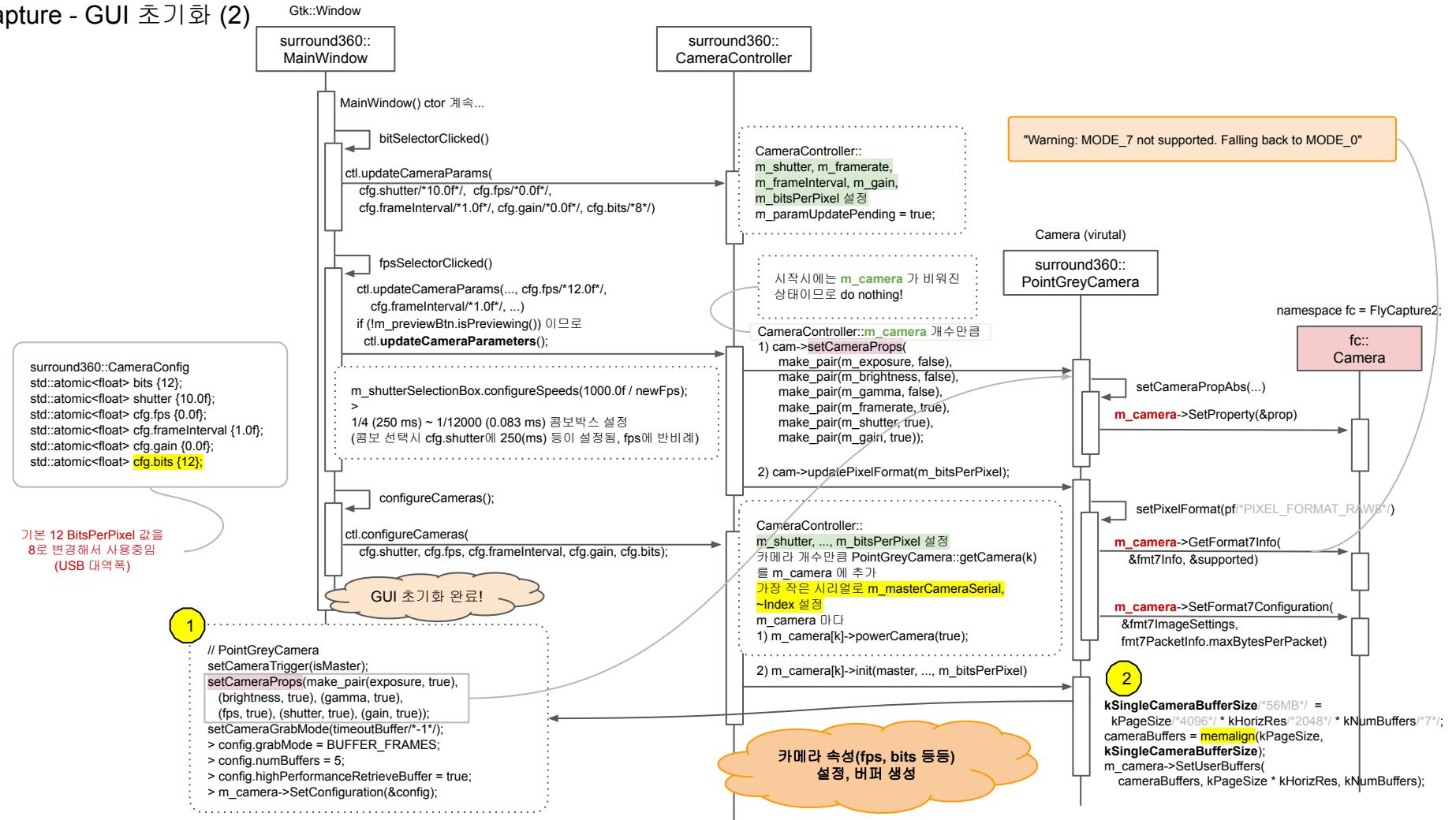


# Capture - GUI 초기화 (1)

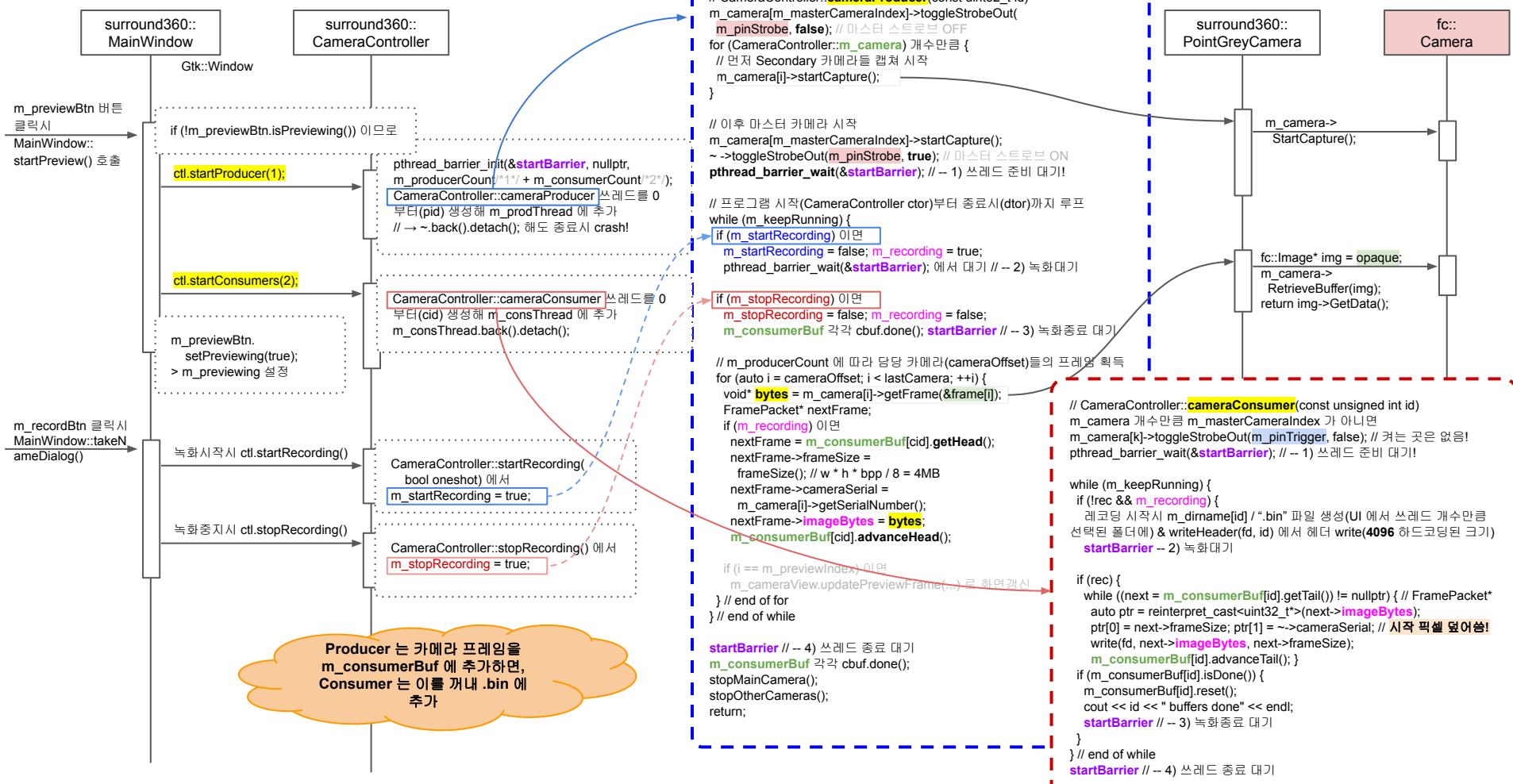
surround360\_camera\_ctl\_ui/source/main.cpp  
surround360\_camera\_ctl\_ui/bin/CameraControlUI



## Capture - GUI 초기화 (2)



# Capture - Saving .bin file



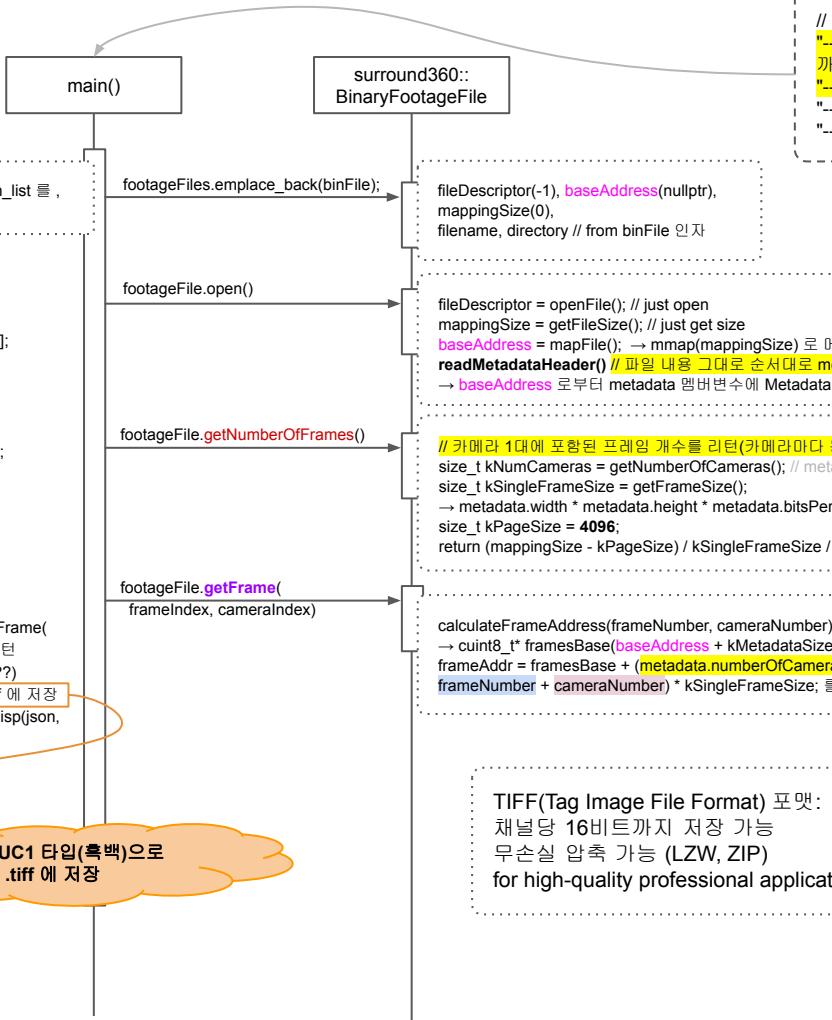
# Capture - Unpacker

surround360\_render/bin/Unpacker  
surround360\_render/source/camera\_isp/U  
npacker.cpp

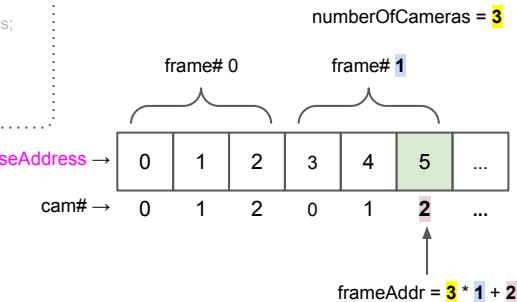
```
// 각 bin 파일마다 loop
for (fileIdx < footageFiles.size()) {
    auto& footageFile = footageFiles[fileIdx];
    footageFile.open();
    cint numCameras =
        footageFile.getNumberOfCameras();
    cint endFrameMax =
        footageFile.getNumberOfFrames() - 1;

    // 각 bin 파일 내의 카메라마다
    for (camIndex < numCameras) {
        // 카메라별 모든 프레임을 비동기로
        for (frameIndex <= endFrame)
            auto frame = footageFile.getFrame(
                frameIndex, cameralIndex);
            upscaled = RawConverter::convert8toFrame(
                frame, width, height); // uint16_t* 리턴
            upscaled 를 (0x101 를 끌어서 why??)
            output_raw_dir/23486816/000000.tiff 에 저장
            isp_dir 폴더 존재시, CameraIspsPipe isp(json,
            kFast, kOutputBpp) 로 png 도 생성
    }
}
```

CV\_16UC1 타입(흑백)으로  
.tiff 에 저장



```
// "--isp_dir", "~/Surround360/cubox/color_calibration/isp",
"--isp_dir", "NOT_EXISTING_DIR", // if --isp_dir 가 존재하지 않는 경로이면 tiff
까지만 생성됨
"--bin_list", "~/Desktop/0.bin,/home/cubox/Desktop/1.bin",
"--output_raw_dir", "~/Desktop/Surround360_output/unpacker@output_raw_dir",
"--output_dir", "~/Desktop/Surround360_output/unpacker@output_dir",
```



TIFF(Tag Image File Format) 포맷:  
채널당 16비트까지 저장 가능  
무손실 압축 가능 (LZW, ZIP)  
for high-quality professional applications

해당 프레임 시작 위치로 이동 후,  
(카메라 개수 x 프레임 인덱스)  
카메라 인덱스 위치의 버퍼 주소를 획득

# Capture - Unpacker

surround360\_render/bin/Unpacker  
surround360\_render/source/camera\_isp/  
Unpacker.cpp

-- 계속 (isp 적용된 png 생성과정이며 optional임) --

```
isp_dir 폴더 존재시,  
"~/Desktop/test/render/config/isp/<23486816>.json"  
을(color, vignetting 캘리브레이션 결과물) json  
액체에 읽어들여  
CameralspPipe isp(json, kFast/*false*/,  
kOutputBpp/*16*/);  
isp.setBitsPerPixel(footageFile.getBitsPerPixel());  
→ Cameralsp::bitsPerPixel 설정  
isp.enableToneMap(); // gamma 등 사용  
isp.loadImage(upscaled->data(), width, height);  
isp.setup();  
isp.initPipe();
```

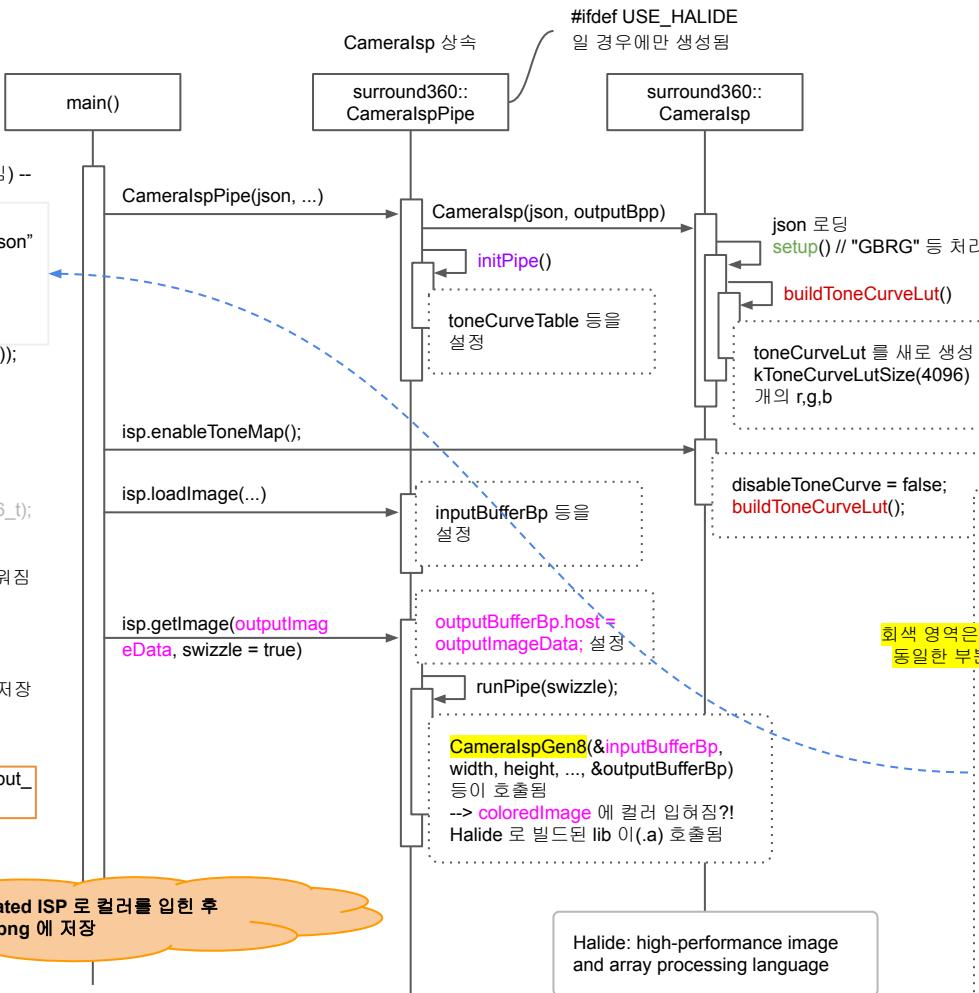
```
int imageSize = width * height * 3 * sizeof(uint16_t);  
auto coloredImage =  
make_unique<vector<uint8_t>>(imageSize);  
isp.getImage(coloredImage->data()); // 컬러 채워짐
```

```
Mat outputImage(height, width, CV_16UC3,  
coloredImage->data());
```

"~output\_dir/<23486816>/000000.png" 등으로 저장  
후, 폴더명을 23486816에서 cam0 식으로 변경  
(rename)

→  
~/Desktop/Surround360\_output/unpacker@output\_  
dir/cam0/000000.png에 저장

Halide accelerated ISP로 컬러를 입힌 후  
.png에 저장

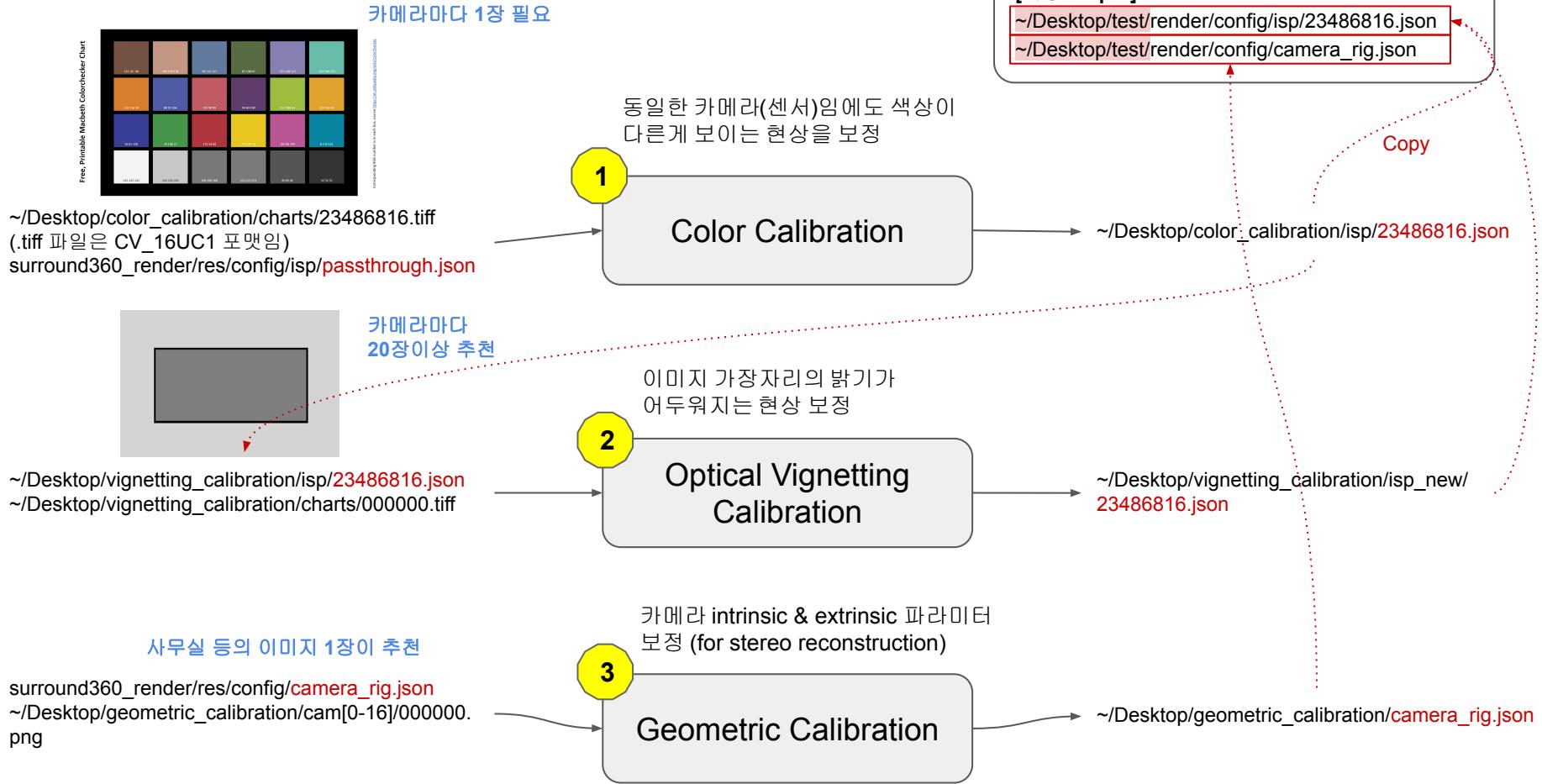


```
#ifdef USE_HALIDE
일 경우에만 생성됨

"Cameralsp" : {
    "serial" : 0,
    "name" : "PointGrey Grasshopper",
    "bitsPerPixel" : 16,
    "compandingLut" : [[0.0, 0.0, 0.0],
                        [0.6, 0.6, 0.0],
                        [0.7, 0.7, 0.0],
                        [1.0, 1.0, 0.0]],
    "blackLevel" : [0.0, 0.0, 0.0],
    "vignetteRollOffH" : [[1.1, 1.1, 1.1],
                          [1.0, 1.0, 1.0],
                          [1.0, 1.0, 1.0],
                          [1.1, 1.1, 1.1]],
    "vignetteRollOffV" : [[1.1, 1.1, 1.1],
                          [1.0, 1.0, 1.0],
                          [1.0, 1.0, 1.0],
                          [1.1, 1.1, 1.1]],
    "bayerPattern" : "GBRG"
},
res\config\isp\passthrough.json (참고용)

"Cameralsp" : {
    "serial" : 0, "name" : ~,
    "bitsPerPixel" : 16,
    "compandingLut" : [[0.0, 0.0, 0.0], ...],
    "blackLevel" : [1285.0, 1285.0, 1285.0],
    "vignetteRollOffH" : [[1.1, 1.1, 1.1], ...],
    "vignetteRollOffV" : [[1.1, 1.1, 1.1], ...],
    "whiteBalanceGain" : [1.1, 1.0, 1.65],
    "stuckPixelThreshold" : 5,
    "stuckPixelDarknessThreshold" : 0.11,
    "stuckPixelRadius" : 0,
    "denoise" : 0.6,
    "denoiseRadius" : 2,
    "ccm" : [[1.02169, -0.05711, 0.03543],
             [0.16789, 1.13419, -0.30208],
             [-0.15726, -0.07864, 1.2359]],
    "sharpening" : [0.5, 0.5, 0.5],
    "saturation" : 1.2,
    "contrast" : 1.0,
    "lowKeyBoost" : [-0.2, -0.2, -0.2],
    "highKeyBoost" : [0.2, 0.2, 0.2],
    "gamma" : [0.4545, 0.4545, 0.4545], // 하드 코딩
    "bayerPattern" : "GBRG"
},
res\config\isp\cmosis_sunex.json (참고용)
```

# Surround 360 Calibration

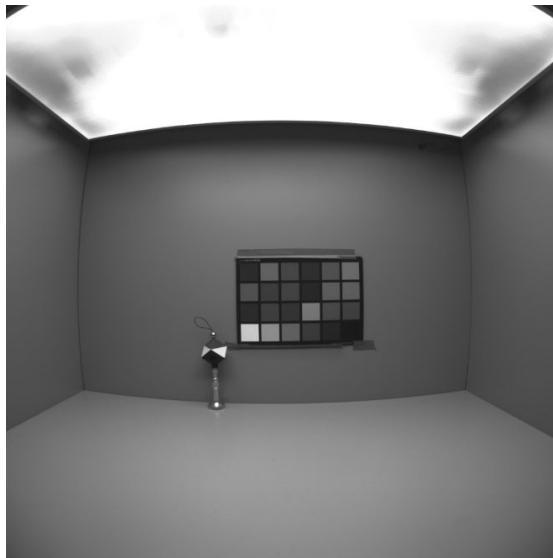


# Color Calibration

[surround360\_render/.vscode/launch.json]

```
// 아래 명령어는 surround360_render\CALIBRATION.md 에 따름
"program": "surround360_render/scripts/color_calibrate_all.py",
"args": [
"--data_dir", "~/Desktop/color_calibration",
"--min_area_chart_perc", "0.3"          // default 0.5, 화면에서 차터 영역의 최소
%                                         // 최소 영역 비율
// "--max_area_chart_perc", "40.0"        // default 40.0
"--black_level_hole",                  // 화면에서 black hole 을 찾아 black level 을 설정
]

```



Color Calibration 샘플 이미지 파일  
surround360\_render\res\example\_data\color\_calibration\_input.png

~/Desktop/color\_calibration/ -> data\_dir  
└ charts/

  └ 23486816.tif  
  | ...  
  └ 23486833.tif

INPUT

└ output/  
  └ 23486816/  
    └ 1\_D50\_raw.png  
    | ...  
    └ 16\_isp\_out.png  
    └ isp\_out.json

복사

└ 23486817/  
  | ...  
  └ 23486833/  
    └ runtimes.txt

isp/  
  └ 23486816.json  
  | ...  
  └ 23486833.json

OUTPUT

whiteBalanceGain, blackLevel, ccm 등  
전반적으로 업데이트됨

Vignetting Calibration을 위해  
생성된 color calibrated json 파일들을  
~/Desktop/vignetting\_calibration/isp/ 하위에  
복사해줘야 함!

카메라마다 1장씩만  
촬영!

# Color Calibration

surround360\_render/scripts/color\_calibrate\_all.py

다음 페이지에 설명

for each tiff 파일(카메라별 이미지):

bin/TestColorCalibration 를 호출하여 카메라별로 isp\_out.json 를 저장

if black\_level\_adjust: // 현재 default 로 false 임

카메라별 “~/Desktop/color\_calibration/output/23486820/black\_level.txt”

파일을 읽어(존재 必), 평균 black\_level 를 계산해서

for each 카메라:

--black\_level X X X 인자로 다시 bin/TestColorCalibration 를 호출

(이 경우, black\_level 를 계산하지 않고 전달된 값을 바로 사용)

모든 카메라의 “intercept\_x.txt” 를 읽어, 모든 카메라의 최대/최소

intercept\_x\_min/max 를 저장 후,

for each 카메라:

“--update\_clamps --clamp\_min XXX --clamp\_max XXX” 인자로

다시 bin/TestColorCalibration 를 호출

--> updateISPWithClamps() 만 수행하고 바로 리턴됨

(<serial\_number>.json 의 clampMin/Max 를 한번 더 업데이트함)

```
// 아래 명령어는 surround360_render\CALIBRATION.md 에 따른
"program": "surround360_render/scripts/color_calibrate_all.py",
"args": [
    "-data_dir", "~/Desktop/color_calibration",
    "-min_area_chart_perc", "0.3"           // default 0.5
    // "--max_area_chart_perc", "40.0"        // default 40.0
    "-black_level_hole",
]
```

이후, serial\_number>.json 로딩 시 clampMin/Max 값은  
Camerasp::clampMin/Max 멤버에 로딩됨

이후, Camerasp::getImage() ->

executePipeline(const bool swizzle)에서

blackLevelAdjust(); // blackLevel 을 가미(0: 원본, 1: black)

antiVignette(); // whiteBalanceGain 곱하기

whiteBalance(); // clampAndStretch();

removeStuckPixels(); // demosaicedImage 픽셀값 설정

demosaic(); // colorCorrect(); // toneCurveLut 가(gamma) 적용됨

colorCorrect();

sharpen();

clampAndStretch() 호출시 rawImage 의 픽셀값의 범위를  
clampMin/Max 내로 한정시킨다.

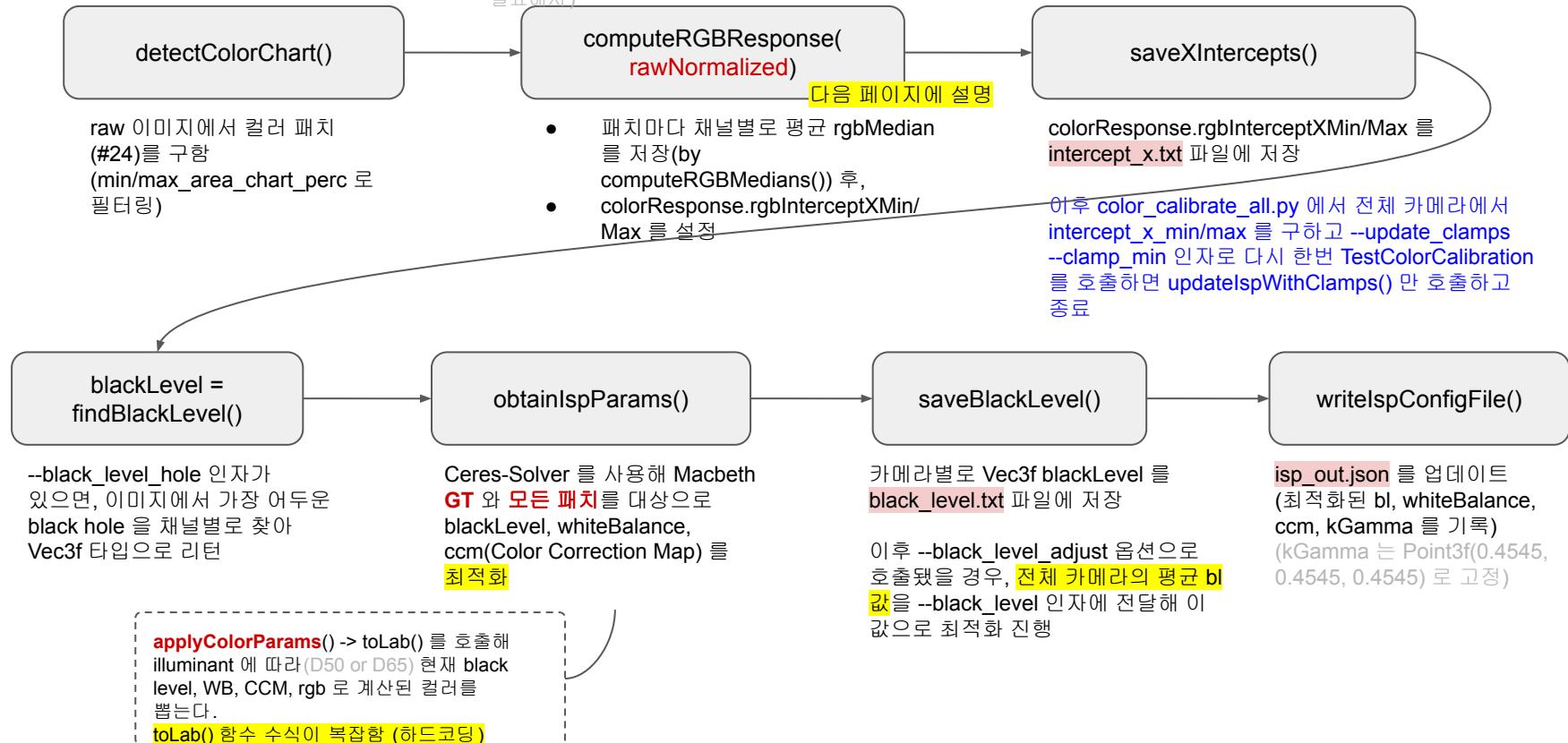
# Color Calibration - 시퀀스

bin/TestColorCalibration  
> 각 카메라마다 호출됨!

```
const Mat rawNormalized =
getRaw(FLAGS_isp_passthrough_path, raw16.clone());
-> resize(1) 이라서 normalized 만 됨(픽셀값),
passthrough.json 은 의미없음(단지 생성자 인자로
필요해서)
```

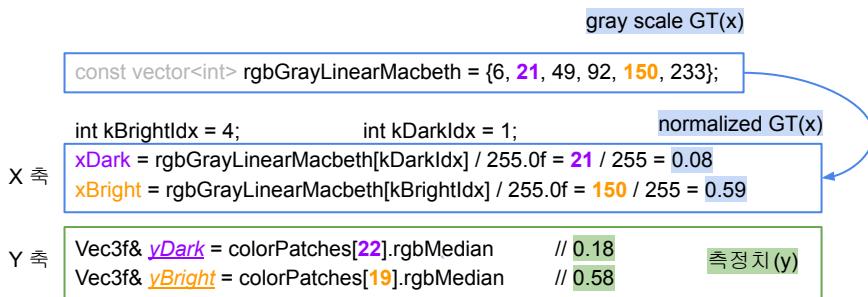
## [Ceres-Solver]

대규모 비선형 최적화 문제를 푸는 오픈소스 C++ 라이브러리. 자동미분 지원  
<https://zzzito.tistory.com/92>

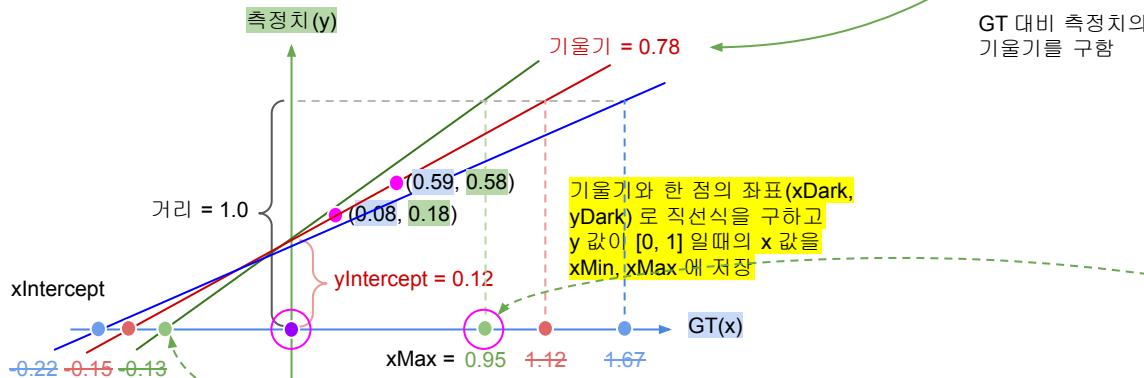
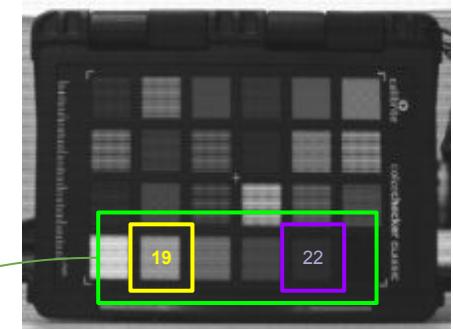


# Color Calibration

ColorResponse computeRGBResponse(...)에서 rgbInterceptXMin/Max를 설정



여기서 구해진  $x_{\text{Min}}, x_{\text{Max}}$ 는  
--update\_clamps 시에  
--clamp\_min에  $x_{\text{Min}}$  중 최대값(vs 0)  
--clamp\_max에  $x_{\text{Max}}$  중 최소값(vs 1)  
에 사용됨!  
(e.g. 0 (-0.13) ~ 0.95)  
-> 픽셀의 최소, 최대값을 clamping 시킴



추후의 Color  
입혀진 이미지  
(참고용)

카메라별로 color\_calibration\output\23486820\intercept\_x.txt에  $x_{\text{Min}}, x_{\text{Max}}$ 를 저장  
--> e.g. [[-0.153058, **-0.136291**, -0.222664], [1.12417, **0.956929**, 1.67439]] // GT 값 범위?  
 $x_{\text{Min}}$ 의 최대값을 구함(RGB 순)  
 $x_{\text{Max}}$ 의 최소값을 구함

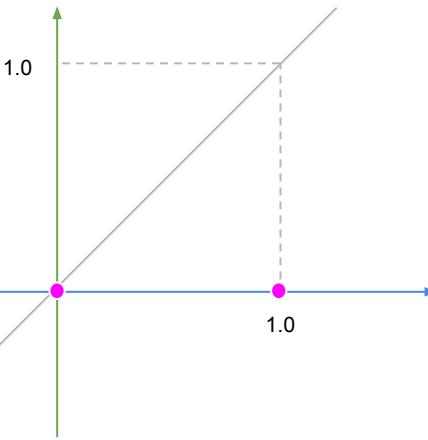
## Color Calibration

clampAndStretch() 호출시 rawImage의 픽셀값의 범위를 clampMin/Max 내로 한정하는 방법

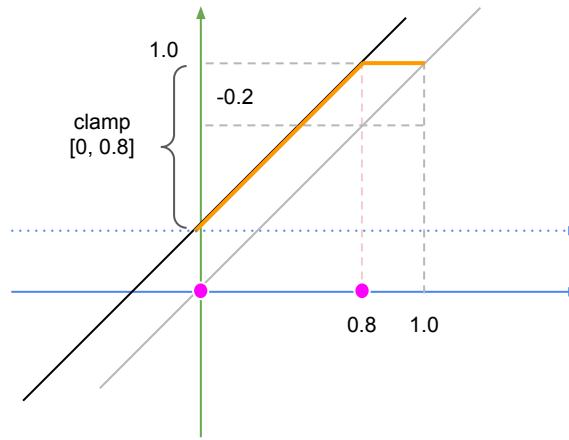
카메라간 밝기 정도를  
맞추려는 의도로 보임!!!

--update\_clamps 시에  
--clamp\_min 에 xMin 최대값(vs 0)  
--clamp\_max 에 xMax 최소값(vs 1)에 따라 clamping 시켜준다.

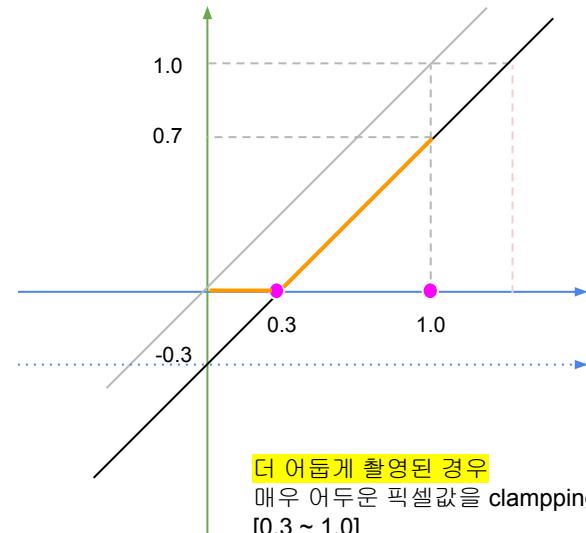
기울기는 같고, 밝기만 다른 경우를 가정하면  
아래와 같이 clamping 된다.  
(Camerasp::clampAndStretch() 참고)



MacBeth 와 밝기값이  
일치하는 경우  
[0 ~ 1.0]  
픽셀값 그대로 사용  
(clamping 없음)



더 밝게 촬영된 경우  
매우 밝은 픽셀 값을 clamping  
[0 ~ 0.8]  
0.8 이상은 0.8로 clamping



더 어둡게 촬영된 경우  
매우 어두운 픽셀값을 clamping  
[0.3 ~ 1.0]  
0.3 이하는 0.3으로 clamping

# Color Calibration - 디버깅용 output

~/Desktop/color\_calibration/output/

block 별 평균치를 threshold로  
binary image로 변경

detectColorChart() 출력 이미지들



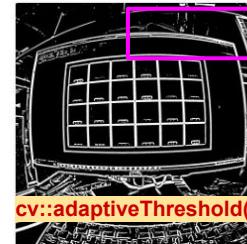
1\_D50\_raw.png



2\_raw\_clamped\_pixels.png



3\_scaled\_blurred.png



4\_adaptive\_threshold.png



5\_fill\_gaps.png



6\_no\_small\_objects.png



7\_clean.png



8\_contours.png



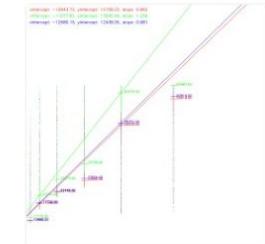
9\_contours.png



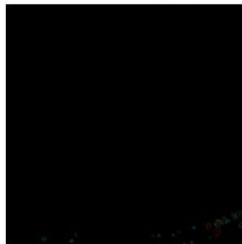
10\_contours.png



11\_detected\_patches.png



12\_gray\_patches\_raw.png



15\_black\_hole.png



16\_illumination.png





8\_contours.png



9\_contours.png

## Color Calibration - 디버깅용 output (2)

촬영 이미지



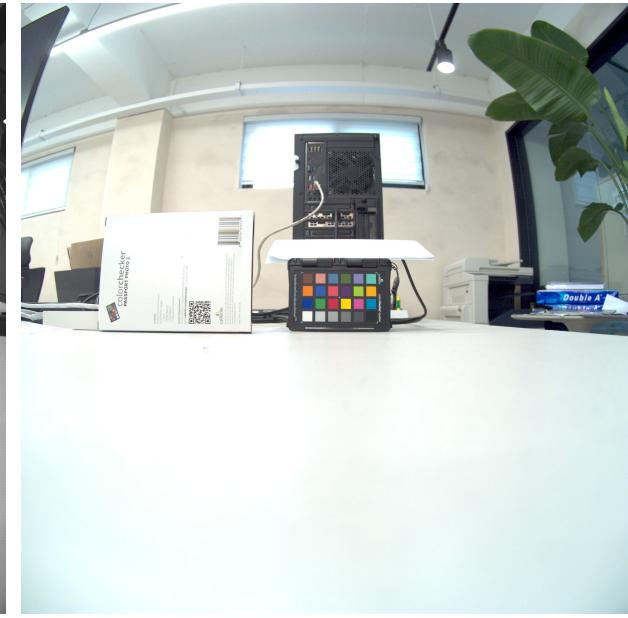
~/Desktop/color\_calibration/output/23486820/  
1\_D50\_raw.png

컬러 패치 표시 (#24)



14\_detected\_patches.png

컬러 캘리브레이션 적용 이미지



20\_isp\_out.png

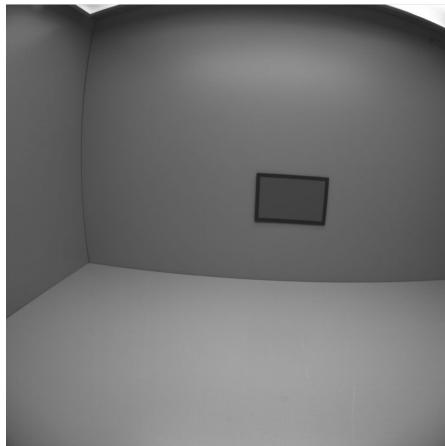
[주의] min/max\_area\_chart\_perc에(% 단위)  
맞춰 Color Charter 와의 거리를 맞춰줘야 24  
패치를 정확히 탐지한다!!!  
(e.g, ColorChecker 거리 40cm)

# Optical Vignetting Calibration

surround360\_render\scripts\vignetting\_calibrate.py

```
// data_dir should have serial_number folders!  
"--data_dir", "~/Desktop/vignetting_calibration",  
"--save_debug_images",
```

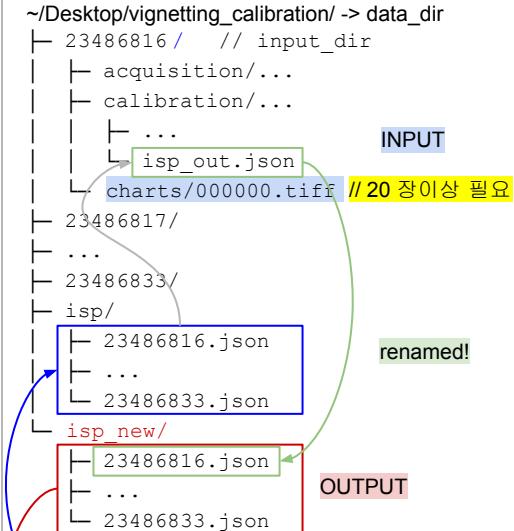
카메라마다 20장 이상 촬영이 추천됨!!!



surround360\_render\res\example\_data/  
vignetting\_calibration\_sample.tif (참고용)



실제 촬영 이미지(1장으로만 테스트함)



~/Desktop/color\_calibration/isp/  
하위 json 파일들을 복사해와야 함!

vignetteRollOffH/V 값만 변경됨

완료 후,  
~/Desktop/test/render/config/isp/23486816.json  
에 복사한 후에 run run\_all.py 를 실행!

# Optical Vignetting Calibration

surround360\_render\scripts\vignetting\_calibrate.py

for <data\_dir>/<serial\_number>.json 각각(카메라별):

1) bin/TestVignettingDataAcquisition 를 호출

ISP\_JSON 인자로 <data\_dir>/isp/<serial\_number>.json 를 전달

2) bin/TestVignettingCalibration 를 호출

3) 생성된 <data\_dir>/23486816/calibration/isp\_out.json 를  
<data\_dir>/isp\_new/23486816.json 로 rename 시키고 있음

카메라별 이미지 각각에 대해  
(e.g, <data\_dir>/23486816/charts/000000.tiff)  
<data\_dir>/23486816/acquisition/data.json 에  
image\_id 별로 패치 중점과 패치의 평균 rgb 값을  
저장

채널별로 Ceres-Solver 를 생성해

- BezierCurve 제어점(5개) paramsX/Y 를 패치 중점 색상에  
가까워지게(?) 움직여 최적화(curve fit)시키고(BezierFunctor)
- 최적화된 paramsX/Y 를  
Cameralsp::setVignetteRollOffH/V(vignetteRollOffH/V) 로  
vignetteRollOffH/V 멤버에 설정
- <data\_dir>/isp/23486816.json 에 비네팅  
("vignetteRollOffH/V")을 업데이트해서  
<data\_dir>/23486816/calibration/isp\_out.json 를 생성
- 이후
  - 1) CameralspPipe::initPipe()에서 v = curveHAtPixel(j);  
vignetteCurveTableH 에 v 를 저장 후, Halide 에 전달되어 ISP  
시에 비네팅보정(?)
  - 2) Cameralsp::getImage() -> executePipeline() ->  
antiVignette() 에서 Cameralsp::rawImage 에 바로 비네팅보정  
(?)

# Optical Vignetting Calibration

```
"Cameraslsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.0, 0.0, 0.0],  
                        [0.6, 0.6, 0.0],  
                        [0.7, 0.7, 0.0],  
                        [1.0, 1.0, 0.0]],  
    "blackLevel" : [0.0, 0.0, 0.0],  
    "vignetteRollOffH" : [[1.1, 1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "vignetteRollOffV" : [[1.1, 1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "bayerPattern" : "GBRG"  
}
```

surround360\_render\res\config\isp\passthrough.json

```
"Cameraslsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.0, 0.0, 0.0],  
                        [0.6, 0.6, 0.0],  
                        [0.7, 0.7, 0.0],  
                        [1.0, 1.0, 0.0]],  
    "blackLevel" : [1285.0, 1285.0, 1285.0],  
    "vignetteRollOffH" : [[1.1, 1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "vignetteRollOffV" : [[1.1, 1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "whiteBalanceGain" : [1.1, 1.0, 1.65],  
    "stuckPixelThreshold" : 5,  
    "stuckPixelDarknessThreshold" : 0.11,  
    "stuckPixelRadius" : 0,  
    "denoise" : 0.6,  
    "denoiseRadius" : 2,  
    "ccm" : [[1.02169, -0.05711, 0.03543],  
             [0.16789, 1.13419, -0.30208],  
             [-0.15726, -0.07864, 1.2359]],  
    "sharpening" : [0.5, 0.5, 0.5],  
    "saturation" : 1.2,  
    "contrast" : 1.0,  
    "lowKeyBoost" : [-0.2, -0.2, -0.2],  
    "highKeyBoost" : [0.2, 0.2, 0.2],  
    "gamma" : [0.4545, 0.4545, 0.4545],  
    "bayerPattern" : "GBRG"
```

surround360\_render\res\config\isp\cmosis\_sunex.json

```
"Cameraslsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.0, 0.0, 0.0],  
                        [0.6, 0.6, 0.0],  
                        [1.0, 1.0, 0.0]],  
    "blackLevel" : [1542.0, 1542.0, 1542.0],  
    "vignetteRollOffH" : [[1.3, 1.3, 1.3],  
                          [1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1],  
                          [1.3, 1.3, 1.3]],  
    "vignetteRollOffV" : [[1.3, 1.3, 1.3],  
                          [1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1],  
                          [1.3, 1.3, 1.3]],  
    "whiteBalanceGain" : [1.1, 1.0, 1.65],  
    "stuckPixelThreshold" : 5,  
    "stuckPixelDarknessThreshold" : 0.11,  
    "stuckPixelRadius" : 0,  
    "denoise" : 0.8,  
    "denoiseRadius" : 4,  
    "ccm" : [[1.02169, -0.05711, 0.03543],  
             [0.16789, 1.13419, -0.30208],  
             [-0.15726, -0.07864, 1.2359]],  
    "sharpening" : [0.5, 0.5, 0.5],  
    "saturation" : 1.2,  
    "contrast" : 1.0,  
    "lowKeyBoost" : [-0.2, -0.2, -0.2],  
    "highKeyBoost" : [0.2, 0.2, 0.2],  
    "gamma" : [0.4545, 0.4545, 0.4545],  
    "bayerPattern" : "GBRG"
```

surround360\_render\res\config\isp\cmosis\_fujinon.json

# Optical Vignetting Calibration

```
"Cameralsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.0, 0.0, 0.0],  
                        [0.6, 0.6, 0.0],  
                        [0.7, 0.7, 0.0],  
                        [1.0, 1.0, 0.0]],  
    "blackLevel" : [0.0, 0.0, 0.0],  
    "vignetteRollOffH" : [[1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "vignetteRollOffV" : [[1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "bayerPattern" : "GBRG"  
}
```

surround360\_render\res\config\isp\passthrough.json

```
"Cameralsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.000, 0.000, 0.000],  
                        [0.600, 0.600, 0.000],  
                        [0.700, 0.700, 0.000],  
                        [1.000, 1.000, 0.000]],  
    "blackLevel" : [6939.000, 6939.000, 6939.000],  
    "clampMin" : [0.000, 0.000, 0.000],  
    "clampMax" : [0.962, 0.967, 0.967],  
    "vignetteRollOffH" : [[1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "vignetteRollOffV" : [[1.1, 1.1, 1.1],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.0, 1.0, 1.0],  
                          [1.1, 1.1, 1.1]],  
    "whiteBalanceGain" : [1.0, 1.0, 1.0],  
    "stuckPixelThreshold" : 0,  
    "stuckPixelDarknessThreshold" : 0.000,  
    "stuckPixelRadius" : 0,  
    "denoise" : 0.000,  
    "denoiseRadius" : 0,  
    "ccm" : [[1.18936, -0.39975, -0.03933],  
             [0.03914, 1.54172, -0.53255],  
             [-0.32338, -0.18724, 1.52166]],  
    "sharpening" : [0.000, 0.000, 0.000],  
    "saturation" : 1.000,  
    "contrast" : 1.000,  
    "gamma" : [0.454, 0.454, 0.454],  
    "bayerPattern" : "GBRG"
```

surround360\_camera\_ctl\_ui\source\CameraView.cpp  
defaultisp 문자열

```
"Cameralsp" : {  
    "serial" : 0,      "name" : "PointGrey Grasshopper",  
    "bitsPerPixel" : 16,  
    "compandingLut" : [[0.000, 0.000, 0.000],  
                        [0.600, 0.600, 0.000],  
                        [0.700, 0.700, 0.000],  
                        [1.000, 1.000, 0.000]],  
    "blackLevel" : [7196.005, 7452.968, 6938.977],  
    "clampMin" : [0.000, 0.000, 0.000],  
    "clampMax" : [0.957, 0.957, 0.957],  
    "vignetteRollOffH" : [[2.956, 2.956, 2.956],  
                          [1.001, 1.001, 1.001],  
                          [0.772, 0.772, 0.772],  
                          [0.833, 0.833, 0.833],  
                          [1.587, 1.587, 1.587]],  
    "vignetteRollOffV" : [[2.018, 2.018, 2.018],  
                          [0.890, 0.890, 0.890],  
                          [0.767, 0.767, 0.767],  
                          [0.911, 0.911, 0.911],  
                          [2.187, 2.187, 2.187]],  
    "whiteBalanceGain" : [1.144, 1.000, 1.684],  
    "stuckPixelThreshold" : 0,  
    "stuckPixelDarknessThreshold" : 0.000,  
    "stuckPixelRadius" : 0,  
    "ccm" : [[1.77691, -0.31608, 0.00825],  
             [-0.56222, 1.58205, -0.84747],  
             [-0.21469, -0.26597, 1.83922]],  
    "sharpening" : [0.000, 0.000, 0.000],  
    "saturation" : 1.000,  
    "contrast" : 1.000,  
    "lowKeyBoost" : [0.000, 0.000, 0.000],  
    "highKeyBoost" : [0.000, 0.000, 0.000],  
    "gamma" : [0.454, 0.454, 0.454],  
    "bayerPattern" : "GBRG"
```

e.g. ~/Desktop/test/render/config/isp/23486820.json  
(~Desktop/vignetting\_calibration/isp\_new/23486820.json)

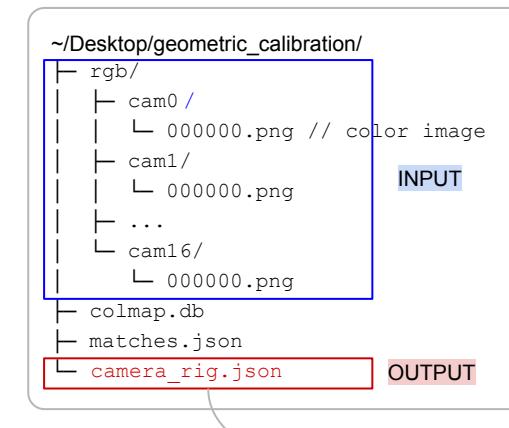
vignetting\_calibration  
시제만 수정되는 부분임

## Geometric Calibration

surround360\_render\scripts\geometric\_calibration.py

feature 가(설명한 경계선, 코너 등)  
많은 scene 촬영 1장이 주천됨!!!  
(e.g. 사무실 인테리어)

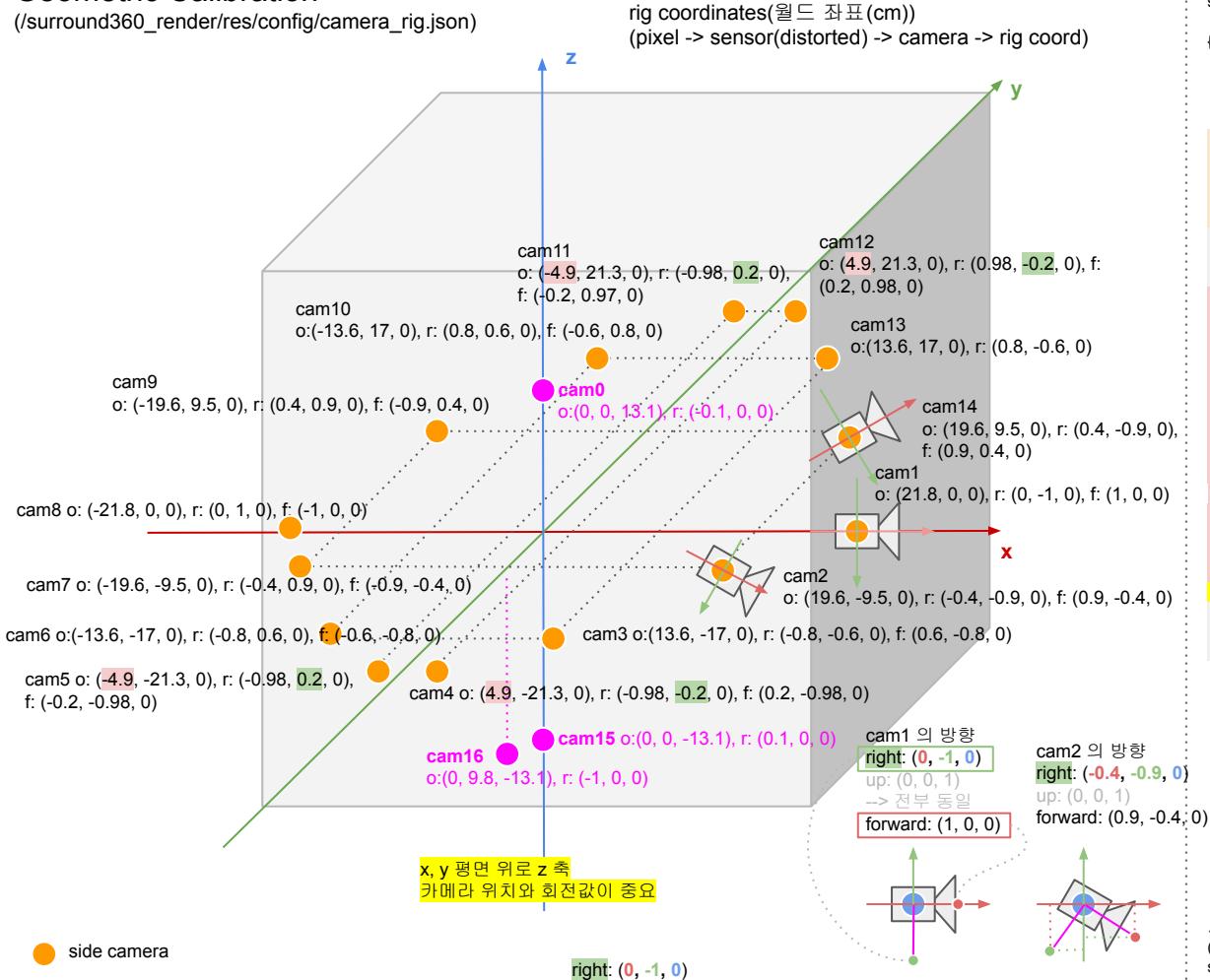
```
--data_dir ~/Desktop/geometric_calibration \ --> 입력 이미지 파일들  
--colmap_dir /usr/local/bin \  
--rig_json $PWD/res/config/camera_rig.json // --> 기본 rig 파일  
--output_json ~/Desktop/geometric_calibration/camera_rig.json \  
--save_debug_images
```



이후, Geometric Calibration 까진 마친 최종  
**camera\_rig.json** 를  
~/Desktop/test/render/config/**camera\_rig.json** 에 복사 후  
run\_all.py 를 실행!!!

# Geometric Calibration

(/surround360\_render/res/config/camera\_rig.json)



```

surround360_render\res\config\camera_rig.json

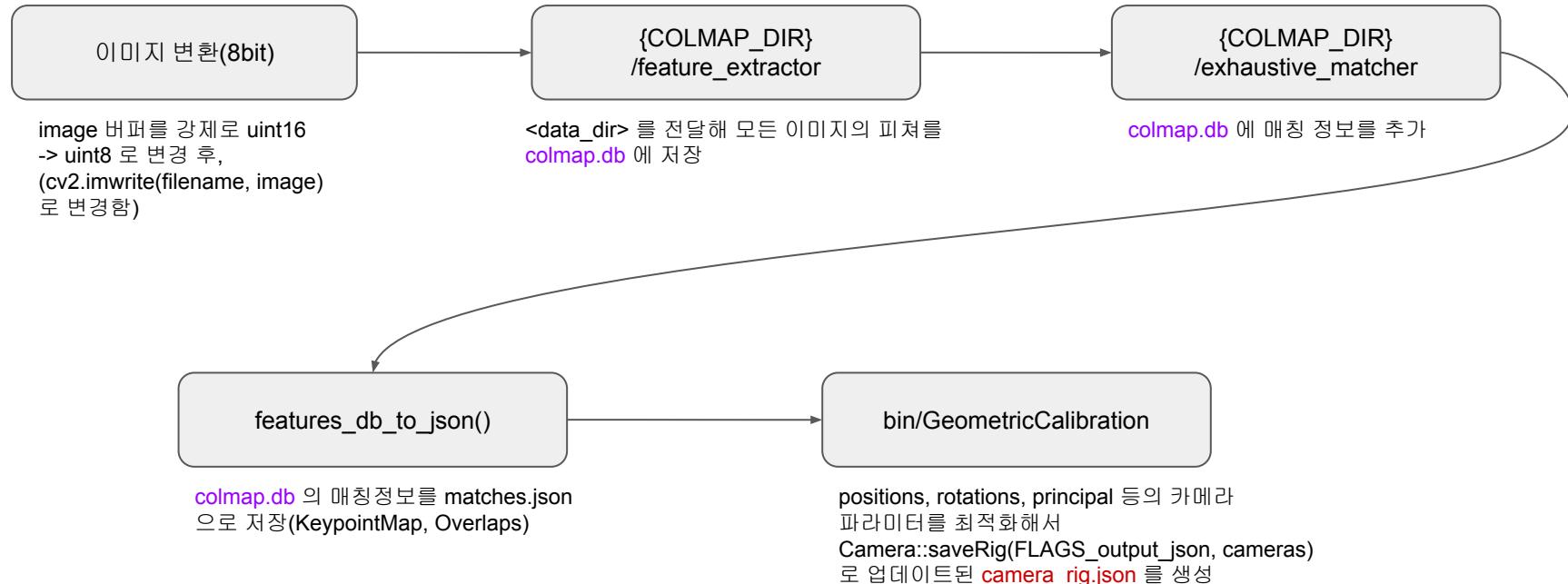
{
  "cameras": [
    {
      "group": "side camera", // 왜곡이 같음!
      "id": "cam1",
      "origin": [ // rig 중심 기준 cm 좌표
        21.799999237060547,
        2.4202861089795465E-15,
        -7.26085832693864E-15
      ], // --> Camera:position
      "principal": [ // --> 전부 동일!
        1024, 1024
      ],
      "right": [ // --> -Y axis (unit vector)
        3.3306690738754696E-16,
        -1,
        1.1102230246251565E-16
      ],
      "up": [ // --> Z axis (unit vector)
        1.1102230246251565E-16,
        3.3306690738754696E-16,
        1 // --> 전부 동일(side cameras)
      ],
      "forward": [ // --> X axis (unit vector)
        1,
        1.1102230246251565E-16,
        -3.3306690738754696E-16
      ],
      "focal": [ // --> pixels / radian
        1269.580673376528,
        -1269.580673376528
      ], // 초점거리, side cameras 동일
      "resolution": [
        2048, 2048
      ],
      "type": "FTHETA",
      "distortion": [
        0, 0
      ],
      "fov": 1.61443(라디안), // --> 92.5도
      // fisheye 만 fov가 존재하며 전부 동일
      "version": 1
    },
    {
      "group": "", "id": "cam0",
      "origin": [0, 0, 13.1],
      "principal": [1024, 1024],
      "right": [-1, 0, 0],
      "up": [0, 1, 0],
      "forward": [0, 0, 1],
      "focal": [483.76220324, -483.76220324],
      // --> 어안렌즈라서 focal 이 작다!!!
      "resolution": [2048, 2048],
      "type": "RECTILINEAR",
      "distortion": [
        0, 0
      ],
      "version": 1
    },
    {
      "group": "", "id": "cam15",
      "origin": [-0, -0, -13.1],
      "principal": [1024, 1024],
      "right": [1, 0, 0],
      "up": [0, 0, 1],
      "forward": [0, 0, 1],
      "focal": [1269.580673376528, -1269.580673376528]
    }, // 초기거리, side cameras 동일
    {
      "group": "", "id": "cam16",
      "origin": [0, 9.8, -13.1],
      "principal": [1024, 1024],
      "right": [1, 0, 0],
      "up": [0, 0, 1],
      "forward": [0, 0, 1],
      "focal": [1269.580673376528, -1269.580673376528]
    }
  ]
}

```

사이드 캠은 fov 항목이 없음  
(RIG\_JSON.md에서 lens fov covers the entire sensor 라고 함)

# Geometric Calibration

surround360\_render\scripts\geometric\_calibration.py



## Geometric Calibration -- refine()

surround360\_render\scripts\geometric\_calibration.py

**refine(cameras, keypointMap, overlaps, ~);** 함수는

- 1) 매칭된 2d 키포인트들의 평균 3d 좌표를 구하고, 이 3d 좌표의 투영이 2d kpt 좌표와 일치되게 3d 좌표를 최적화 한 후(**TriangulationFunctor**)
- 2) positions, rotations, principal 등의 카메라 파라미터도 최적화 시켜 다시 3d 좌표의 투영이 2d 키포인트와 일치하게 최적화를 수행한다(**ReprojectionFunctor**).

```
void refine(std::vector<Camera>& cameras,  
           KeypointMap keypointMap, std::vector<Overlap> overlaps,  
           const int pass, // pass_count = 0 ~ 10  
           const std::string& debugDir)
```

```
>  
std::vector<Trace> traces = disconnectedTraces(keypointMap, overlaps);  
> overlaps 된 (카메라, 매칭 인덱스) 를 traces 에 추가해서 리턴  
모든 overlaps 개수만큼 traces 배열에 추가 &  
매칭된 키포인트들을 trace.references 멤버에 (카메라, 매칭 인덱스) pair 로 추가
```

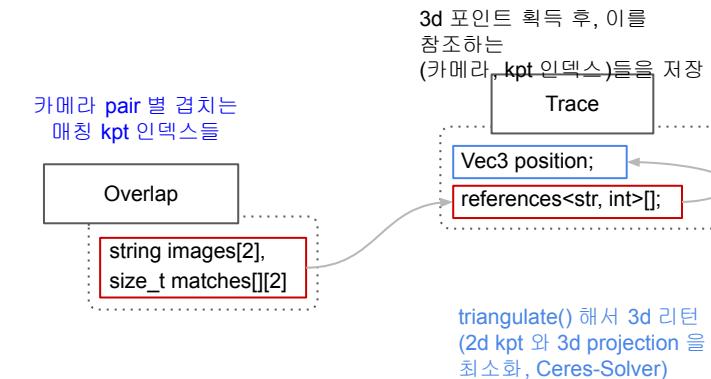
**triangulateTraces(traces, keypointMap, cameras);**에서 Trace::position 3d 좌표를 설정  
> traces 의 2d kpt 들을 triangulate 한 3d point 를 구해서 trace.position 에 설정  
각 traces마다 observations 배열을 생성 & (camera/\*ref~ str로 구함\*/,  
keypoint.position/\*2d\*/) 를 추가(카메라별 keypoint)

trace.position = triangulate(observations);에서는 3d 월드좌표(world)를 투영한 좌표가 2d  
키포인트들과 오차가 최소가 되는 3d 좌표를 리턴(**TriangulationFunctor::addResidual()**)

```
removeOutliers(overlaps, keypointMap, traces, cameras, FLAGS_outlier_factor);  
> overlaps.matches 에 inlier 만 남김
```

assembleTraces(), triangulateTraces(), removeOutliers() 를 2번 더 호출 후,

이후, traces마다 **ReprojectionFunctor::addResidual(problem, positions[camera],  
rotations[camera], ...)**로 카메라의 positions, rotations 멤버 등을 최적화함 !!!



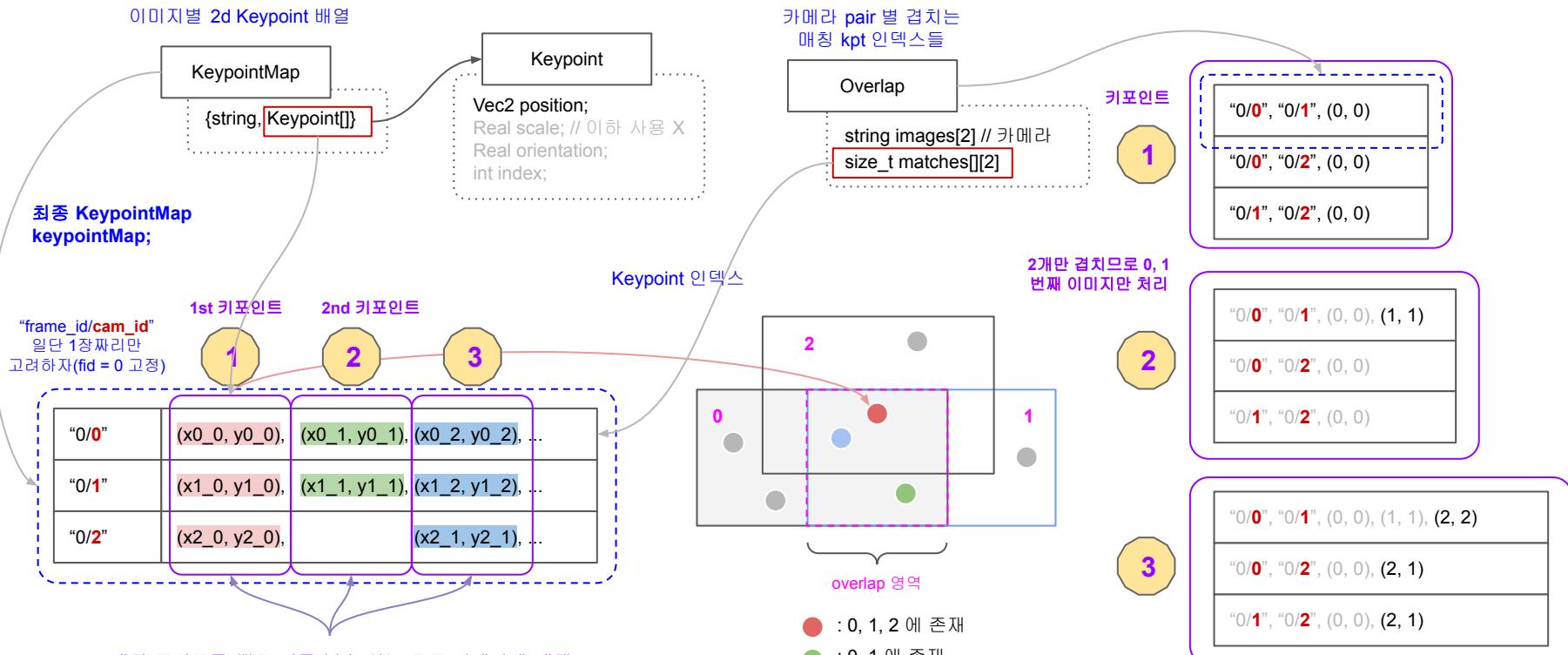
2d kpt 들을 (평균)3d로 보냈다가  
다시 2d로 projection 시킨  
좌표가 같아지게 최적화

# Geometric Calibration

surround360\_render/source/calibration/GeometricCalibration.cpp

generateArtificalPoints() 함수 설명  
(matchs.json 도 유사함)

같은 3d 점(rig point)을 바라보는 카메라가 2개  
이상이면 overlaps 배열에 pair 단위로 추가



1) 3d 랜덤 포인트를 뽑고 이를 볼수 있는 모든 카메라에 대해 keypointMap에 추가(카메라 2d 포인트)

2) 랜덤 포인트를 2대 이상의 카메라에서 보는 경우, overlaps 배열에도 추가

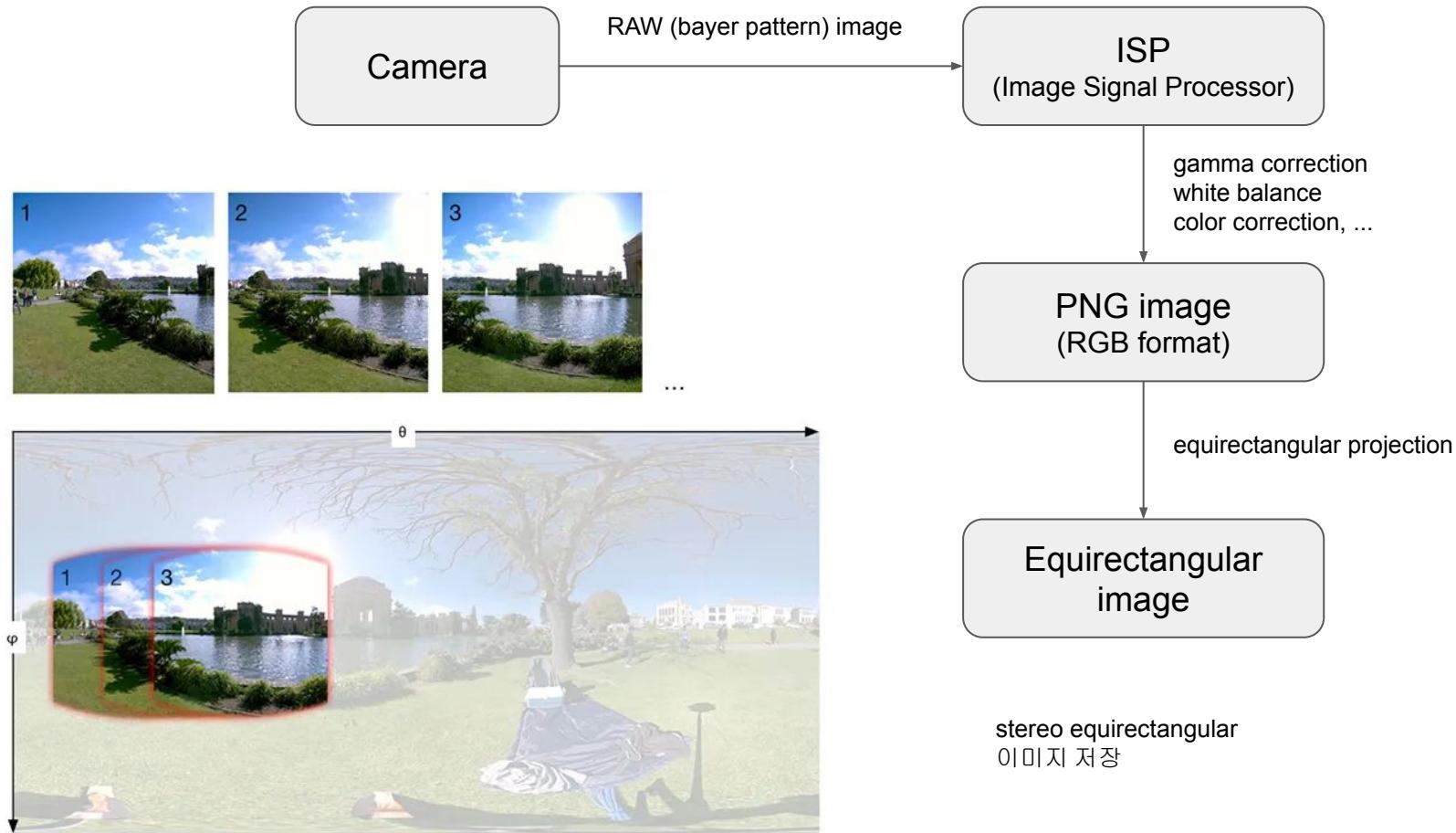
--> 루프(10000번) 반복

$$\begin{aligned} \text{조합}(nCr) &= n! / ((n - r)! * r!) \\ &= 17! / ((17 - 2)! * 2!) \\ &= 355687428096000 / \\ &(1307674368000 * 2) = 136 \\ \text{--> 최대 overlaps 배열크기} &= 136 \end{aligned}$$

최종 Overlaps[];

# Rendering pipeline

surround360\_render\scripts\run\_all.py



# Render

surround360\_render\scripts\batch\_process\_video.py

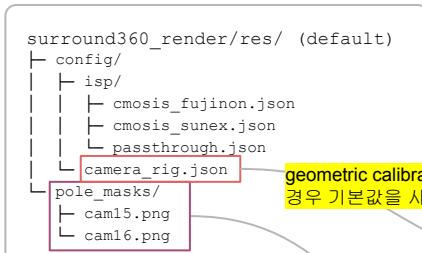
```
// batch_process_video.py
for () 각 프레임 이미지들에 대해서
// "6k" 일 경우(디폴트 값),
render_params["SHARPENING"] = 0.25
render_params["EQR_WIDTH"] = 6300
render_params["EQR_HEIGHT"] = 3072
render_params["FINAL_EQR_WIDTH"] = 6144
render_params["FINAL_EQR_HEIGHT"] = 6144

// "4k" 일 경우,
render_params["SHARPENING"] = 0.25
render_params["EQR_WIDTH"] = 4200
render_params["EQR_HEIGHT"] = 1024
render_params["FINAL_EQR_WIDTH"] = 4096
render_params["FINAL_EQR_HEIGHT"] = 2048

// bin/TestRenderStereoPanorama 를 호출(render_params 를 전달받음)
--rig_json_file "{RIG_JSON_FILE}" // dest_dir/config/camera_rig.json
--imgs_dir "{SRC_DIR}/rgb" // dest_dir/rgb
--frame_number {FRAME_ID} // 000000
--output_data_dir "{SRC_DIR}" // dest_dir
--output_equirect_path "{OUT_EQR_DIR}/eqr_{FRAME_ID}.png"
// --> ( e.g. dest_dir/eqr_frames/eqr_000000.png )
--output_cubemap_path "{OUT_CUBE_DIR}/cube_{FRAME_ID}.png"
// --> dest_dir/cube_frames/cube_000000.png
--eqr_width {EQR_WIDTH} // 6300
--eqr_height {EQR_HEIGHT} // 3072
--final_eqr_width {FINAL_EQR_WIDTH/HEIGHT} // 6144 / 6144

main() 에서 renderStereoPanorama() 호출
```

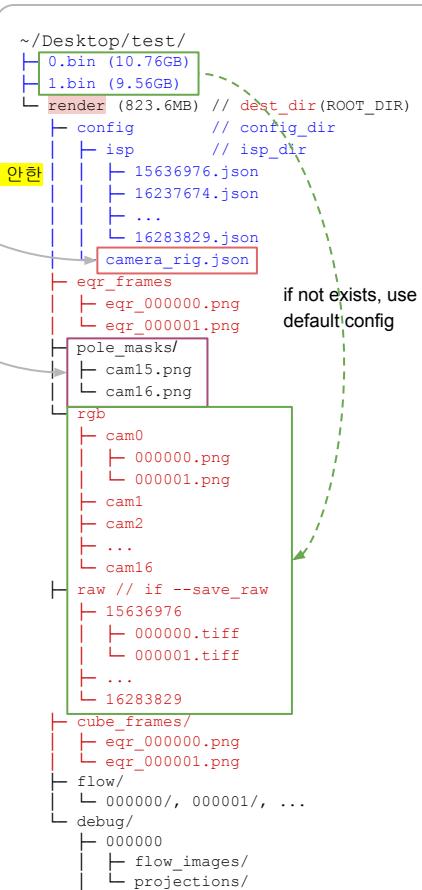
run\_all.py 에서는 총 3 단계를 수행  
 1) bin/Unpacker (png 이미지 획득)  
 2) batch\_process\_video.py (렌더링)  
 3) .mp4 파일 생성



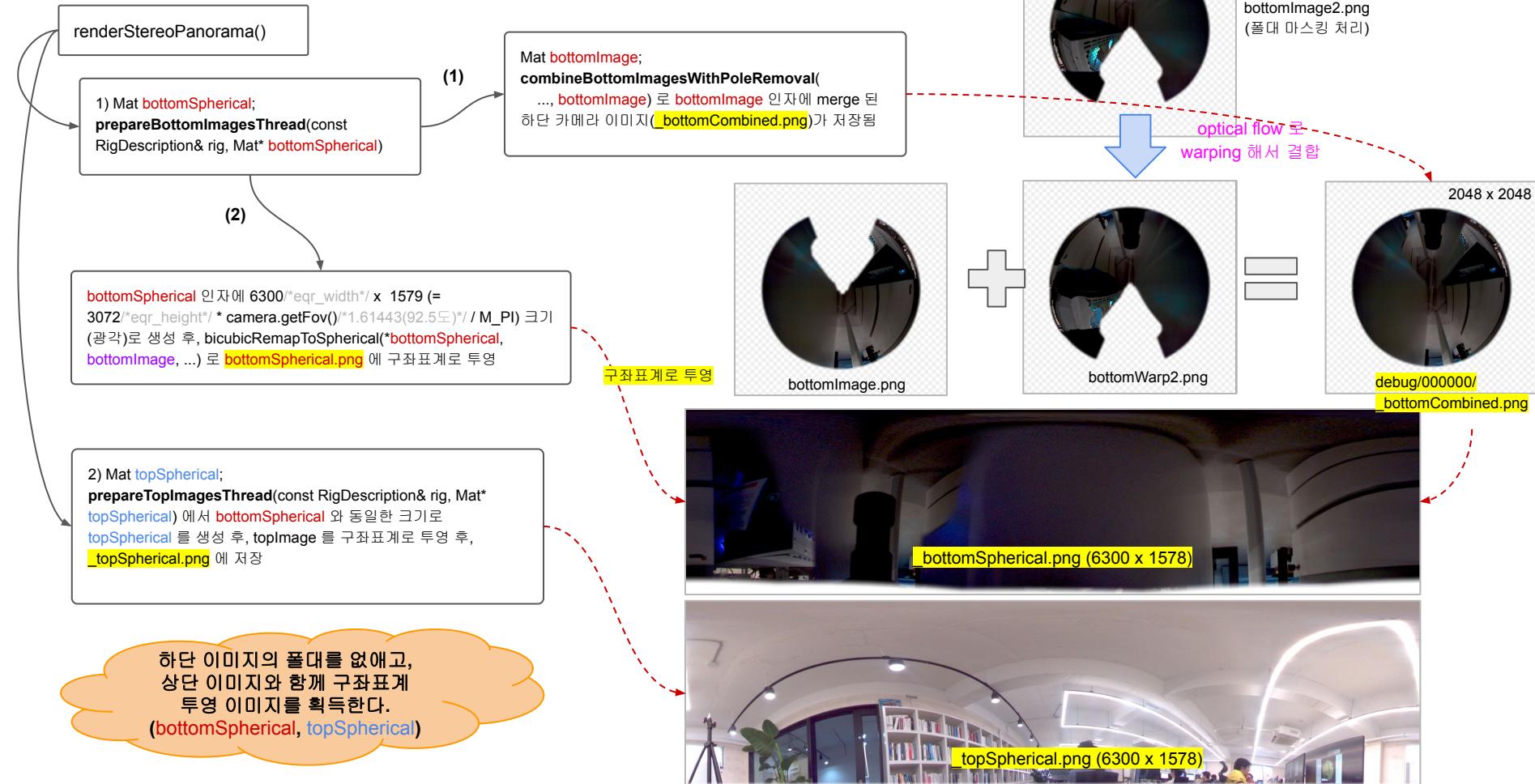
// run\_all.py 호출  
 --data\_dir "~/Desktop" // .bin 파일이 포함된 디렉토리  
 --dest\_dir "~/Desktop/test/render" // 하위에 config/camera\_rig.json, config/isp/ 등 필요  
 기타 파라미터들 전달

// run\_all.py에서 1) Unpacker 후, 2) batch\_process\_video.py 를 호출  
 --flow\_alg {FLOW\_ALG} // "pixflow\_low"  
 --root\_dir "(ROOT\_DIR)" // dest\_dir  
 --surround360\_render\_dir "{SURROUND360\_RENDER\_DIR}"  
 // ~/cuboX/Surround360/surround360\_render  
 --quality {QUALITY} // 디폴트 "6k"  
 --start\_frame {START\_FRAME} // 0  
 --end\_frame {END\_FRAME} // dest\_dir/rgb/cam0/ 파일 개수 - 1  
 --cubemap\_width {CUBEMAP\_WIDTH} // 1536 or 0 (not generate)  
 --cubemap\_height {CUBEMAP\_HEIGHT} // 1536 or 0  
 --cubemap\_format {CUBEMAP\_FORMAT} // 'video'  
 --rig\_json\_file "{RIG\_JSON\_FILE}"  
 // --> dest\_dir/config/camera\_rig.json  
 {FLAGS\_RENDER\_EXTRA} // --enable\_top --enable\_bottom  
 --enable\_pole\_removal --save\_debug\_images --verbose

3) mp4 저장 (e.g. ~/Desktop/test/render/test\_453380\_6k\_TB.mp4)



## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)

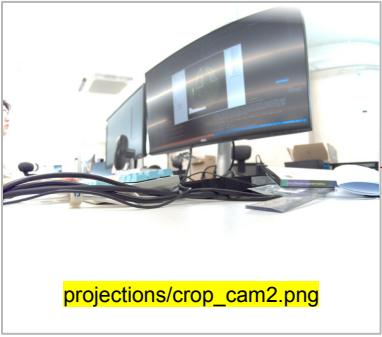


## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)

```
renderStereoPanorama()
```

사이드 이미지들을 구좌표계  
투영 이미지들로 만든다.  
(projectionImages)

3) side 카메라 이미지 각각을 projectionImages 배열에 투영(구좌표계)  
vector<Mat> projectionImages;  
**projectSphericalCamImages(rig, FLAGS\_imgs\_dir,**  
**FLAGS\_frame\_number, projectionImages);**



// side 카메라 이미지를 spherical coordinates 에 (projectionImages) 투영

**void projectSphericalCamImages()**

```
const RigDescription& rig, const string& imagesDir, // ~/Desktop/test/render/rgb
const string& frameNumber, vector<Mat>& projectionImages)
```

vector<Mat> camImages = rig.loadSideCameralImages(imagesDir, frameNumber);
> side 카메라마다 ~/Desktop/test/render/rgb/cam1/000000.png 등을 읽어옴

fov는 focal, resolution  
으로 계산

projectionImages.resize(camImages.size()); // 14개

```
const float hRadians = 2 * approximateFov(rig.rigSideOnly, false); // 1.195, 68.5°
const float vRadians = 2 * approximateFov(rig.rigSideOnly, true); // 1.195, 68.5°
```

사이드 카메라 개수만큼 projectionImages[camIdx].create(FLAGS\_eqr\_height/\*50727\*/,
vRadians / M\_PI /\*= 1169\*/ , FLAGS\_eqr\_width/\*6300\*/ \* hRadians / (2 \* M\_PI)/\*= 1198\*/,
CV\_8UC4) // partial 이미지를 생성

// 카메라마다 쓰레드로 구면 좌표계에 투영

float direction = -float(camIdx) / float(camImages.size()) \* 2.0f \* M\_PI; // 시계 방향 360°  
**projectSideToSpherical** 쓰레드에서 bicubicRemapToSpherical()로 구면 좌표계에 그려줌

```
ref(projectionImages[camIdx]), // <- 여기에 그려짐(구면 좌표계로)
cref(camImages[camIdx]), // 사이드캠 이미지를 읽어서
cref(camera),
direction + hRadians / 2/*leftAngle*/, direction - hRadians / 2/*rightAngle*/,
vRadians / 2/*topAngle*/, -vRadians / 2/*bottomAngle*/);
```

// direction = -0

leftAngle = 1.195 (68.5°)

rightAngle = -1.195 (-68.5°)

topAngle = 1.195

bottomAngle = -1.195

// direction = -0.159 (2pi / #14)

leftAngle = 1.036 (-0.159 + 1.195)

rightAngle = -1.354 (-0.159 + -1.195)

topAngle = 1.195

bottomAngle = -1.195

overlapAngleDegrees

= (137° \* 14 - 360°) / 14  
= 111.2° (겹치는 각도)

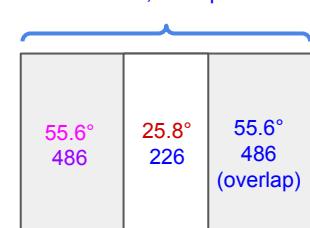
overlapImageWidth

= 1198 \* (111.2 / 137)  
= 972 (겹치는 폭)

numNovelViews

= 1198 - 972  
= 226 (안겹치는 폭)

137°, 1198 px



360° / 14 = 25.8° 만

안겹치게 된다

(계산 과정만 참고)

# Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)

```
renderStereoPanorama()
```

4) novel view 와 stereo spherical panoramas 를 생성  
`Mat sphericalImageL, sphericalImageR;`  
`generateRingOfNovelViewsAndRenderStereoSpherical(`  
 `rig.getRingRadius(), fovHorizontal/*137°*,`  
 `projectionImages, sphericalImageL, sphericalImageR,`  
 `opticalFlowRuntime, novelViewRuntime);`  
 디버그용  
`sphericalImgL/R.png, sphericalImg_offsetwrapL/R.png`  
 이미지 저장 (다음 페이지 참고)



```
void prepareNovelViewGeneratorThread()
const int overlapImageWidth, // 972
const int leftIdx, // only used to determine debug image filename
Mat* imageL, Mat* imageR,
NovelViewGenerator* novelViewGen){
    overlaplmgL 에 imageL 에서 겹치는 영역 이미지를 저장(복사)
    novelViewGen->prepare(overlapImageL, overlapImageR,
    prevFrameFlowLtoR, prevFrameFlowRtoL,
    prevOverlapImageL, prevOverlapImageR);
```

```
// novel view 조각으로 left/right eye equirect panorama 생성
void generateRingOfNovelViewsAndRenderStereoSpherical(
```

```
cameraRingRadius, camFovHorizontalDegrees/*137°*/,
vector<Mat>& projectionImages,
Mat& panolImageL, Mat& panolImageR, ...)
```

projectionImages 개수 만큼 (side 캠)

novelViewGenerators 배열에 추가

for () (prepareNovelViewGeneratorThread,

overlapImageWidth, leftIdx,

&projectionImages[leftIdx], &projectionImages[rightIdx], // 1198 x 1169

novelViewGenerators[leftIdx])

vector<Mat> panoChunksL(projectionImages.size(), Mat());

for () (renderStereoPanoramaChunksThread 쓰레드는

leftIdx, numCams, camImageWidth/Height/\*1198 x 1169\*,

numNovelViews/\*226\*, fovHorizontalRadians/\*137°\*,

vergeAtInfinitySlabDisplacement,

novelViewGenerators[leftIdx],

&panoChunksL/R[leftIdx] // 여기에 채워짐(로컬 변수임)

최종적으로 전달받은 Mat 에 채워줌(sphericalImageL = panolImageL)

panolImageL/R = stackHorizontal(panoChunksL/R); // 단순히 옆으로 붙임

panolImageL/R = offsetHorizontalWrap(panolImageL/R, // shift + wrap

+ zeroParallaxNovelViewShiftPixels/\*147.78917, 뒷 페이지에 설명\*/);

스티칭해서 최종 좌/우  
파노라마 이미지를 생성  
(sphericalImageL/R)

NovelViewGenerator

Mat imageL, imageR;

Mat flowLtoR, flowRtoL;

NovelViewGeneratorLazyFlow

NovelViewGeneratorAsymmetricFlow

renderStereoPanoramaChunksThread(

leftIdx, numCams, ..., Mat\* chunkL/R)

> novelViewGen->combineLazyNovelViews()

로 이어붙여서 chunkL/R 인자에 리턴

겹치는 영역 이미지로  
optical flow 를 저장

```
void NovelViewGeneratorAsymmetricFlow::prepare(
const Mat& colorImageL/colorImageR, // 겹치는 영역 이미지
const Mat& prevFlowLtoR, prevFlowRtoL,
const Mat& prevColorImageL/R) {
    NovelViewGeneratorLazyFlow::imageL, imageR, flowLtoR,
```

flowRtoL 멤버를 computeOpticalFlow() 를 호출해서 설정(복잡함)

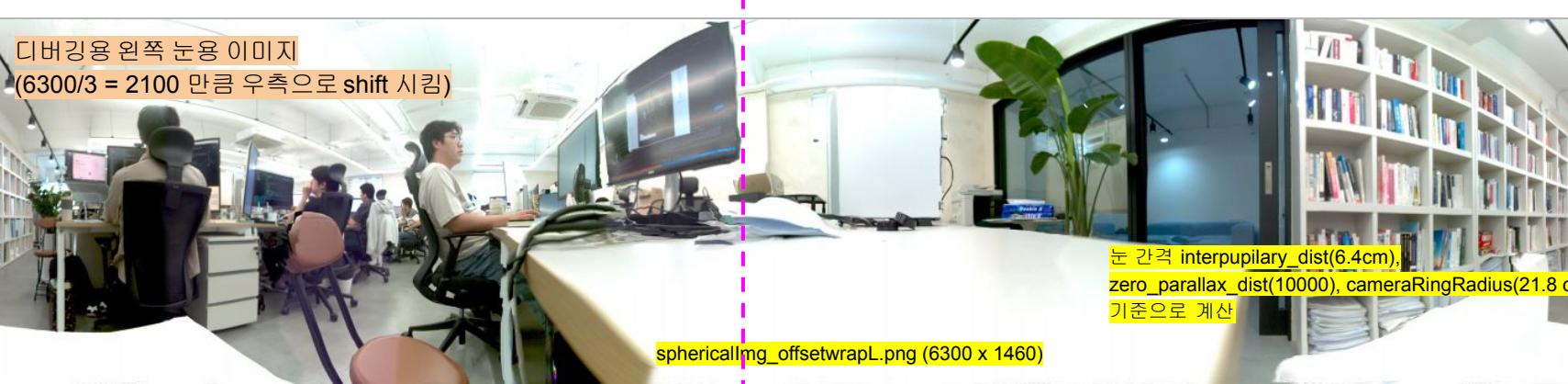
## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)



[디버깅용] 최종 좌/우  
파노라마 이미지 확인 (shift)  
(사용처 X)

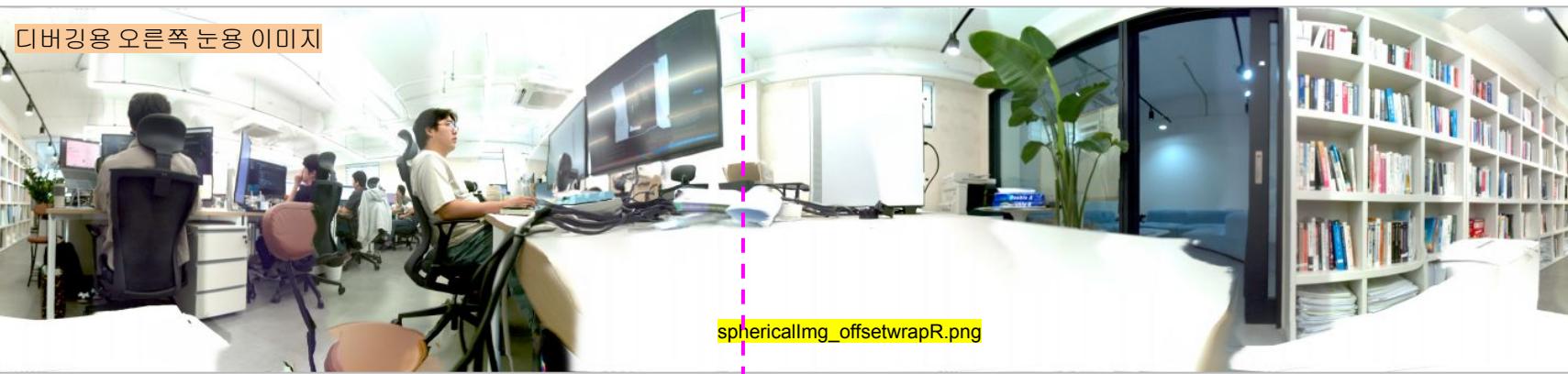
디버깅용 왼쪽 눈용 이미지

( $6300/3 = 2100$  만큼 우측으로 shift 시킴)



sphericalImg\_offsetwrapL.png (6300 x 1460)

디버깅용 오른쪽 눈용 이미지



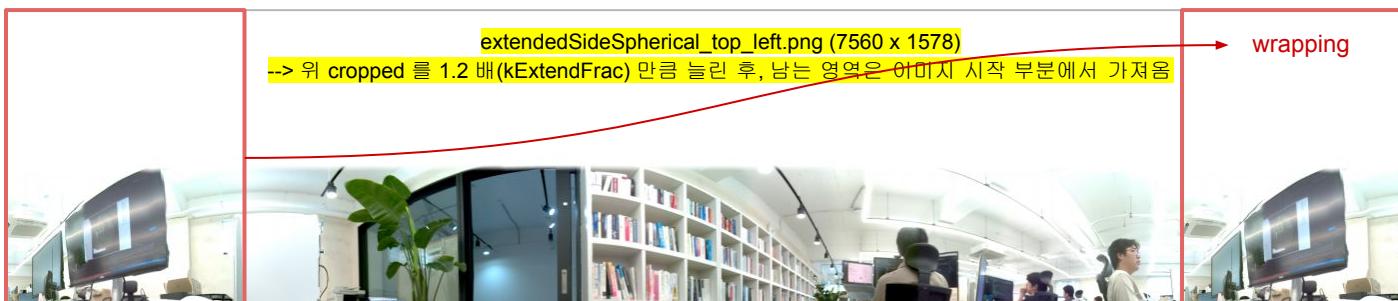
sphericalImg\_offsetwrapR.png

## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)



5) top camera 와 사이드 카메라 간에 optical flow 수행  
topFlowThreadL = std::thread poleToSideFlowThread,  
    "top\_left", // "top\_right" 도 구동  
    cRef(rig), &sphericalImageL, &topSpherical,  
    &topSphericalWarpedL/R // 여기에 채워짐(로컬 변수);  
비슷하게 bottomSphericalWarpedL/R 도 획득

상단/하단 과 사이드  
(sphericalImageL/R)간에  
스티칭된 상단/하단 이미지를  
리턴





상단, 하단, 사이드 이미지를 합침  
(fisheye top, bottom,  
sphericalImageL/R)

TODO:

- 1) 상/하단 + 사이드 색상차이 보정  
(상/하단 카메라 Color Calibration 필요)
- 2) 하단 + 사이드 스티칭 보정  
(하단에 삼각대 설치 후, 테스트 필요)

## 6) 큐브맵 저장

FLAGS\_cubemap\_width > 0 이면,  
Mat cubemapImageL = stackOutputCubemapFaces(...) 한 이미지를  
FLAGS\_output\_cubemap\_path에 저장  
최종적으로 Mat stereoEquirect =  
stackVertical(vector<Mat>({sphericalImageL, sphericalImageR}));  
이미지를 **FLAGS\_output\_equirect\_path**에 저장



7) 최종 stereo equirectangular 이미지 저장

```
Mat stereoEquirect =  
stackVertical(vector<Mat>({sphericalImageL,  
sphericalImageR}));  
imwriteExceptionOnFail(FLAGS_output_equirect_path,  
stereoEquirect);
```



# TODO List

- 상/하단 + 사이드 색상차이 보정
  - 상/하단 카메라 Color Calibration 후, 테스트 필요  
(현재는 Side 카메라 1대에서 Color Calibration 한 json 을 전체 카메라에서 복사해 사용중임)
- 하단 + 사이드 스티칭 보정
  - 하단에 삼각대 설치 후, 테스트 필요  
(현재 임시로 책상에 걸쳐서 하단 카메라가 아래를 향하게 설치돼 있어, 하단 이미지는 매우 어둡게 표시되고 있음)
- Geometric Calibration 결과가 맞는 것인지 확인 필요
  - 디폴트 camera\_rig.json 대비 principal 값 등이 오차가 많이 발생중임  
e.g, (1024, 1024) 고정 vs (1067, 463), (1262, 754) 등등
  - Geometric Calibration 수행한 결과 반영해서 최종 렌더링까지 테스트  
(현재 Side 카메라 1대로 Color/Vignetting Calibration 결과까지만 반영한 상태임)

**EOD**

# Color Calibration - detectColorChart()

1) 블러 처리

```
cv::GaussianBlur(imageScaled/*src*/, imageBlur/*dst*/, kBlurSize/*Size(15, 15)*/, kSigmaAuto/*0*/)로  
블러 처리 --> ~/Desktop/color_calibration/output/3_scaled_blurred.png
```

2) grayscale image 를 binary image 로 변경(영상 이진화)

```
Mat bw; // bw = black and white 의미
```

```
adaptiveThreshold(
```

```
    imageBlur,           // src, Source 8-bit single-channel image
    bw,                 // dst, Destination image
    kMaxValue,          // maxValue, 임계값을 넘었을 때 적용할 값, 255.0
    ADAPTIVE_THRESH_MEAN_C, // adaptiveMethod(블록 평균 계산방법), 주변영역의(blockSize) 평균 - C
    THRESH_BINARY_INV,   // thresholdType, if src(x, y) > T(x, y) 0 else kMaxValue
    kBlockSize,          // blockSize, 19
    kWeightedSub,        // C, 2
); --> 4_adaptive_threshold.png
```

3) Fill Gap (모풀로지 닫힘 연산으로 구멍 메우기)

```
const float imageSize = raw8.rows/*2048*/ * raw8.cols/*2048*/;           // 2048 * 2048 = 4,194,304(S)
const float minAreaChart = (FLAGS_min_area_chart_perc/*0.5*/ / 100.0f) * imageSize/*S*/; // 20,971.52
const float maxAreaChart = (FLAGS_max_area_chart_perc/*40.0*/ / 100.0f) * imageSize/*S*/; // 1,677,721.6
```

```
const int numPatches = numSquaresW/*6*/ * numSquaresH/*4*/;           // 24
const float minAreaPatch = minAreaChart/*20,971.52*/ / numPatches/*24*/; // 873.81
const float maxAreaPatch = maxAreaChart/*819.2*/ / numPatches/*24*/; // 69,905.06
const float bwArea = bw.rows * bw.cols;                                // 2048 * 2048 = S
static const float kScaleElement = 10.0f;
const float morphElementSize = kScaleElement/*10.0f*/ * minAreaPatch/*873.81*/ / bwArea/*S*/; // 0.002083
```

```
bw = fillGaps(bw, morphElementSize/*0.002083*/, saveDebugImages, outputDir, stepDebugImages);
Mat imageBwOut;
Mat element = createMorphElement(imageBwIn.size(), elementSize, MORPH_CROSS);
> return getStructuringElement(shape, ...) // 2048 * elementSize 로(morphRadius) 앵커 및 커널크기 인자전달
morphologyEx(imageBwIn, imageBwOut, MORPH_CLOSE/*닫힘 연산*/, element);
--> 5_fill_gaps.png
```

4) 작은 객체 제거

```
static const float kScaleSmallestObject = 0.3f; // 아래에서 minAreaPatch의 30% 미만 크기를 걸러냄!
const float smallestObjectSize = kScaleSmallestObject/*0.3f*/ * minAreaPatch/*873.81*/; // 262.143
bw = removeSmallObjects(bw, smallestObjectSize, saveDebugImages, ~);
> connectedComponentsWithStats(imageBwIn, labels/*출력 레이블 맵 행렬*/, stats/*각 레이블의 bbox, area*/,
centroids/*각 레이블 영역의 중심좌표*/); 결과를 이용해 작은 객체를 제거(검은색으로 처리)
--> 6_no_small_objects.png
```

5) Dilate Gap (모풀로지 팽창 연산으로 노이즈 제거)

```
bw = dilateGaps(bw, morphElementSize/*0.002083*/, saveDebugImages, ~);
Mat element = createMorphElement(imageBwIn.size(), elementSize, MORPH_RECT);
dilate(imageBwIn, imageBwOut, element);
--> 7_dilate.png
```

```
int la = connectedComponentsWithStats(bw, labels, stats, centroids, 8); // 객체(레이블)수 반환
```

```
for 각각의 la에 대해 {
    findContours()에서 윤곽선 추출 후, approxPolyDP()로 윤곽선 포인트 개수를 줄이고 윤곽선(contours)
개수가 numPatches(24) + 1 이상이 되면(차트 경계선 포함) contours 배열에 (포인트 배열을) 추가
--> 즉, 모든 레이블에 대해 포함한 박스 개수가 24개 이상인 박스(포인트 배열)의 배열(contours)을 저장
}
```

for 각각의 contours에 대해 { // 박스의 배열

```
Moments mu = moments(cont, false); 와 const int area = mu.m00; 를 통해 area에 폐곡선의 넓이를 저장
area(폐곡선의 넓이)가 minAreaPatch, maxAreaPatch를 벗어나거나,
컨투어 개수가 kNumEdges/*4*/ 가 아니거나,
boundingBox의(minAreaRect(cont)) 종횡비가 kMaxAspectRatio/*2.0f*/ 보다 큰 경우,
또는 isContourConvex(cont)가 아닌 경우에 필터링 시킴
```

```
LOG(INFO) << "Patch found (" << countPatches++ << ")"; // 필터링에서 살아남은 경우
```

```
Mat patchMask(bw.size(), CV_8UC1, Scalar::all(0));
cv::drawContours(patchMask/*in & out*/, contours/*in*/, i, 255/*흰색*/, CV_FILLED/*-1, 내부 채움*/);
> contours 배열에서 i 번째 컨투어를 patchMask에 그림
```

```
ColorPatch colorPatch;
colorPatch.centroid = centroid; colorPatch.mask = patchMask/*컨투어 그려진 이미지*/;
colorPatchList.push_back(colorPatch); // colorPatchList 배열에 패치를 추가!!!
} // End of for()
```

```
vector<ColorPatch> colorPatchListClean = removeContourOutliers(colorPatchList); // 평균 패치간 거리의 2배
를 초과하는 아웃라이어 제거
vector<ColorPatch> colorPatchListSorted = sortPatches(colorPatchListClean, numSquaresW, image.size()); // colorPatch.centroid 좌표기준으로 정렬(연산이 복잡함)
```

```
LOG(INFO) << "Number of patches found: " << colorPatchListSorted.size();
--> 11_detected_patches.png
```

```
return colorPatchListSorted; // detectColorChart() 리턴
```

# Color Calibration - getRaw() 등등

```
// surround360_render\source\test\TestColorCalibration.cpp
```

-- main() 계속 --

```
const Mat rawNormalized = getRaw(FLAGS_isp_passthrough_path, raw16.clone());
```

```
// surround360_render\source\calibration\ColorCalibration.cpp
```

```
Mat getRaw(const string& ispConfigFile, const Mat& image/*raw16 이미지*/) {
```

```
    Cameralsp cameralsp(getJson(ispConfigFile), getBitsPerPixel(image));
```

```
> res\config\isp\passthrough.json 로딩, setup()에서 bayer pattern tables 등 설정 & buildToneCurveLut() 호출
```

```
> buildToneCurveLut()에서는 gamma correction(보통 0.4545) + low/high key boost(보통 0) + contrast(보통 1)
```

결과를 Cameralsp::toneCurveLut에 저장

```
cameralsp.loadImage(image);
```

```
> resizeInput(inputImage)를 호출해서 Cameralsp::rawImage 멤버에 resize 만해서(현재 기본 1이라서 동일)
```

```
크기, setResize()로 case X) image를 복사(픽셀값은 0 ~ 1로 normalize 해서)
```

```
return cameralsp.getRawImage(); // 위의 Cameralsp::rawImage를 단순히 리턴
```

```
}
```

2) ColorResponse colorResponse = computeRGBResponse(

```
    rawNormalized.clone(), // const Mat& raw
    true, // const bool isRaw
```

```
    colorPatches,
```

```
    FLAGS_isp_passthrough_path,
```

```
    FLAGS_save_debug_images,
```

```
    FLAGS_output_data_dir,
```

```
    stepDebugImages,
```

```
    "raw");
```

(1) computeRGBMedians(colorPatches, raw, isRaw, ispConfigFile); // 패치마다 평균 rgb 저장!

```
> 모든 colorPatches에 대해 colorPatches[i].rgbMedian = getRgbMedianMask(image, colorPatches[i].mask,
```

```
    ispConfigFile, isRaw); // 패치 마스크 영역(colorPatches[i].mask)을 찾아 패치마다 평균 rgb 값을
```

```
colorPatches[i].rgbMedian에 저장
```

(2) ColorResponse colorResponse;

모든 채널(3)마다 rgbGrayLinearMacbeth[6]의 밝기 대비 컬러 패치의 rgbMedian 차이로 기울기

```
colorResponse.rgbSlope를 구하고 rgblInterceptY, rgblInterceptXMin / ~Max 등도 획득
```

```
saveDebugImages이면, plotGrayPatchResponse()를 호출 & 여기서 12_gray_patches_raw.png 저장
```

```
return colorResponse;
```

3) saveXIntercepts(colorResponse, FLAGS\_output\_data\_dir);

```
> intercept_x.txt 파일에 단순히 colorResponse.rgblInterceptXMin / ~Max[3]를 저장
```

4) obtainIspParams()

```
colorPatches, FLAGS_illuminant, raw16.size(),
isBlackLevelSet, // false
```

```
FLAGS_save_debug_images, FLAGS_output_data_dir, stepDebugImages,
blackLevel, whiteBalance, ccm // 전부 red
```

```
);
```

```
> vector<double> wbAndCcm(kNumChannels/*3*/ * kNumChannels/*3*/, 0.0); 선언
```

```
for (int i = 0; i < kNumChannels/*3*/; ++i) {
    wbAndCcm[i * (kNumChannels + 1)] = 1.0; // 0, 4, 8 만 1로 설정( --> Identity Matrix)
}
```

```
vector<double> bezierX(kBezierOrderX/*4*/ + 1, 1.0); // bezierY도 생성(둘다 5차원 값)
```

(1) 모든 패치마다 addResidual()로 추가

```
IspFunctor<kBezierOrderX, kBezierOrderY>::addResidual(
    problem, // 이하 인자들이 전부 const 가 아님
```

```
    bezierX, bezierY,
```

```
    bl, // vector<double> bl(kNumChannels/*3*/, 0.0)로 초기화, isBlackLevelSet이면 blackLevel가 사용됨
```

```
    wbAndCcm, // Color Correction Matrix (CCM)
```

```
    centroid.x / width, centroid.y / height,
```

```
    rgbsRef[], // 패치 rgbMedian
```

```
    illuminant,
```

labRef); // GT(labMacbeth 컬러 값) 아래 operator()에서 residuals 계산시 사용됨

```
> auto* cost = new CostFunction(new IspFunctor(x, y, rgbs, illuminant, labRef));
problem.AddResidualBlock(
    cost,
```

```
    nullptr, // loss
```

bezierX.data(), // 이하 인자들이 최적화됨!!! (bezierX / Y는 colorPatches[i].labMedian 저장 시 사용?!)

```
    bezierY.data(),
```

```
    bl.data(), // bl[i] = blackLevel[i] // Cameralsp::blackLevel; // cv::Point3f
```

```
    wbAndCcm.data());
```

(2) 이후 Solve()를 호출하여 bool IspFunctor::operator()(bezierX/Y, bl, wbAndCcm, residuals) 함수에서

```
vector<double> lab = applyColorParams(rgbs, illuminant, illumScale, bl, wbAndCcm); 하면
```

```
> applyColorParams()에서는 WB * CCM * rgbs를 적용해서 3채널 lab[3]을 리턴
```

residuals[i] = labRef[i] - lab[i]; // 최소가 되게 Ceres-Solver가 bezierX/Y, bl, wbAndCam를 최적화

main()에서는 obtainIspParams()의 전달인자였던 blackLevel 등을

```
saveBlackLevel(blackLevel, FLAGS_output_data_dir);를 호출해서 black_level.txt에 저장
```

```
writelSpConfigFile(ispConfigPathOut, // isp_out.json에 이하 인자들 저장!!
```

```
    cameralsp, blackLevel * maxPixelValue // const Vec3f& blackLevel,
```

```
    whiteBalance, ccm.t(), kGamma); // kGamma = Point3f(0.4545, 0.4545, 0.4545);로 하드코딩됨
```

최종 확인용으로 ispConfigPathOut(isp\_out.json)를 Cameralsp cameralspTest 객체로 로딩해

17\_isp\_out.png 저장

Ceres-Solver 설명  
<https://zzziito.tistory.com/92>

# OpenCV functions

```
int cv::connectedComponentsWithStats( // 레이블맵 통계 정보 확인, 레이블 개수 반환
    InputArray image,           // 8-bit single-channel 입력 이미지
    OutputArray labels,         // 출력 레이블맵(객체영역별 고유번호 행렬) CV_32S
    OutputArray stats,          // 레이블별 통계 출력
    OutputArray centroids,      // 레이블별 무게 중심좌표(CV_64F)
    int connectivity = 8,       // 연결성, 8 or 4 for 8-way or 4-way connectivity
    int ltype = CV_32S          // 출력행렬 타입. CV_32S or CV_16U
)

void cv::inRange( // src (I) 가 lowerb, upperb 사이면 (<=) dst (I) 가 255 아니면 0
    InputArray src,             // 입력 이미지
    InputArray lowerb,           // inclusive lower boundary array or a scalar
    InputArray upperb,           // upper boundary array or a scalar
    OutputArray dst,             // output array of the same size as src and CV_8U type(255 or 0)
) --> connectedComponentsWithStats() 로 모든 레이블맵(labels)을 찾은 후, 각 레이블의 해당 픽셀을 inRange() 로 찾아 255로 설정(나머지는 black(0))
e.g, inRange(labels, label, label, bwLabel)

void cv::findContours( // binary image에서 윤곽선을 획득
    InputOutputArray image,     // 8-bit single-channel image
    OutputArrayOfArrays contours, // 탐지된 윤곽선 포인트 배열의 배열
    OutputArray hierarchy,       // 윤곽선마다의 전후좌우 윤곽선 인덱스를 리턴해줌
    int mode,                  // 윤곽선 획득방법(RETRE_TREE: 모든 윤곽선 + 내포된 윤곽선들의 전체 hierarchy 까지 획득)
    int method,                // Contour approximation 방법(알고리즘), CHAIN_APPROX_SIMPLE =
수직, 수평, 대각선 방향의 마지막 포인트만 남김(e.g, up-right rectangular(수직 직사각형)은 4)
    Point offset = Point()      // 모든 윤곽 포인트의 shift offset, ROI 와 전체이미지 처리시 유용)

void cv::approxPolyDP( // Approximates a polygonal curve(s) with the specified precision
    InputArray curve,           // 입력 2d point 들
    OutputArray approxCurve,     // 근사치(approximation) 결과
    double epsilon,              // approximation accuracy, 원본 curve 와 approximation 간의 최대거리
    bool closed,                // approximated curve is closed(its first and last vertices are connected) 여부
)
double cv::arcLength( // closed contour perimeter(닫힌 컨투어 둘레) or 곡선 길이를 계산
    InputArray curve,
    bool closed )
```

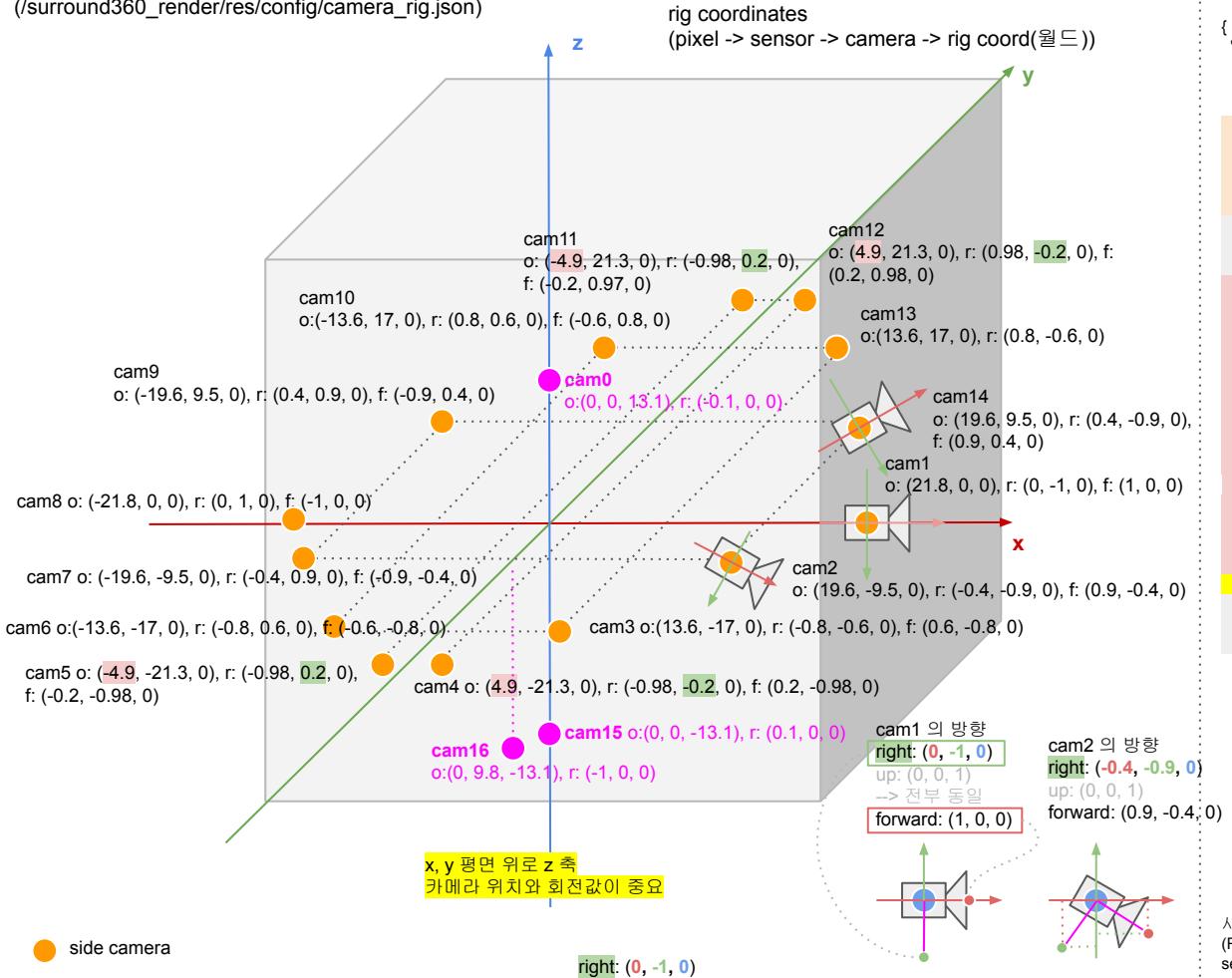
```
RotatedRect cv::minAreaRect(
    InputArray points
) // 포인트들을 전부 포함하는 최소 사각형 리턴

Moments cv::moments( // Calculates all of the moments up to the third order of a polygon or
rasterized shape.
    InputArray array,           // 입력 2d 포인트 배열
    bool binaryImage = false // If it is true, all non-zero image pixels are treated as 1's.
) --> 보통 오브젝트 중심을 찾는데 많이 사용된다(cX = m10 / m00, cY = m01 / m00)

void cv::drawContours( // Draws contours outlines or filled contours.
    InputOutputArray image,     // 타겟 이미지
    InputArrayOfArrays contours, // (입력) 컨투어 포인트 배열
    int contourIdx,            // 그릴 컨투어 인덱스(-1 이면 전체 컨투어가 그려짐)
    const Scalar & color,       // 컨투어 컬러
    int thickness = 1,          // 컨투어 라인의 두께(thickness=FILLED(-1) 이면 내부가 채워짐)
    int lineType = LINE_8,
    InputArray hierarchy = noArray(), // Optional
    int maxLevel = INT_MAX,
    Point offset = Point()
)
void cv::findNonZero( // Returns the list of locations of non-zero pixels.
    InputArray src,             // single-channel array (type CV_8UC1)
    OutputArray idx,             // the output array, type of cv::Mat or std::vector<Point>,
corresponding to non-zero indices in the input
)
void cv::calcHist(
    const Mat * images,          // 히스토그램을 계산할 입력 이미지 배열
    int nimages,                 // 이미지 개수
    const int * channels,         // 히스토그램을 계산할 채널 번호들 배열
    InputArray mask,              // 히스토그램을 계산할 영역(Mat()) 이면 전체?!
    OutputArray hist,             // 히스토그램 계산 결과
    int dims,                    // 히스토그램 계산 결과를 저장한 hist 의 차원
    const int * histSize,          // 각 차원의 bin 개수
    const float ** ranges,        // 각 차원의 분류 bin 의 최소값, 최대값
    bool uniform = true,
    bool accumulate = false
)
```

## Geometric Calibration

(/surround360\_render/res/config/camera\_rig.json)



surround360\_render\res\config\camera\_rig.json

```
{  
  "cameras": [  
    {  
      "group": "side camera", // 왜곡이 같음!  
      "id": "cam1",  
      "origin": [ // rig 중심 기준 cm 좌표  
        21.799999237060547,  
        19.641120632888047,  
        9.45866518173573,  
        -1.2101430544897733E-15  
      ],  
      "group": "side camera",  
      "id": "cam14",  
      "origin": [  
        19.641120632888047,  
        9.45866518173573,  
        -1.2101430544897733E-15
```

```
2.4202861089795465E-15,  
-7.26085326938864E-15  
], // --> Camera::position  
"principal": [           --> 전부 동일!  
    1024,      1024  
],  
"right": [ --> -Y axis (unit vector)  
    3.3306690738754696E-16,  
    -1,  
    1.1102230246251565E-16  
],  
"up": [           --> Z axis (unit vector)  
    0,      1,      0  
],  
"forward": [           --> X axis (unit vector)  
    0,      0,      1  
],  
"focal": [  
    483.76220324, -483.76220324  
]
```

e.g. 185도 & 센서 크기가 2448 픽셀이면

"resolution": [ 2048, 2048 ],

```
"type" : "FTHETA",
"distortion" : [      0,      0     ],
"fov" : 1.61443, // --> 92.5도
```

--> fisheye 만 fov가 존재하며 전부 동일  
1) Camera::setFov(json["fov"]) 로 설정됨  
> Real cosFov = cos(1.61443) -

$\rightarrow \text{RealCostUV} = \cos(1.01443) = -0.04361983$ , `fovThreshold` 멤버에  $\text{cosFov} * \text{abs}(\text{cosFov}) = -0.0019027$  을

서장(나머지 캠들은 setDefaultFov()로  
fovThreshold 멤버변수 = 0;)

2) Camera::getFov() 는 return  
fovThreshold < 0 ?  
acos(-sqrt(-fovThreshold)) 해서  
acos(-0.04362) = 1.61443(92.5도)로  
원래 값이 리턴됨

```
        "version": 1
    },
    {
        "group": "",      "id": "cam15",
        "origin": [-0,       -0,       -13.1],
    ...
    {
        "group": "",      "id": "cam16",
        "origin": [0,        9.8,      -13.1],
```

사이드 캠은 fov 항목이 없음  
(RIG\_JSON.md에서 lens fov covers the entire sensor라고 함)

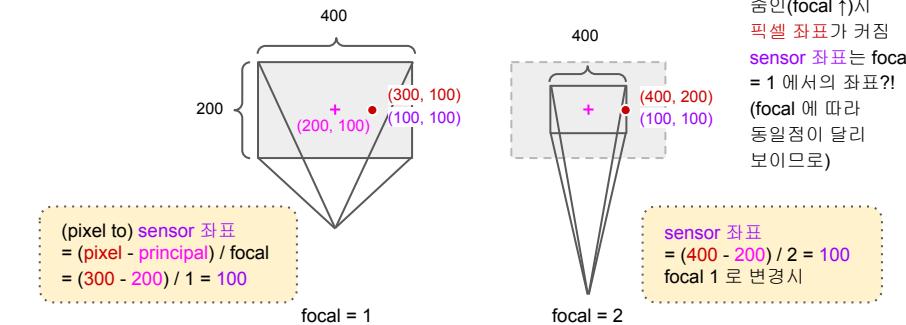
## Geometric Calibration -- refine()

surround360\_render\scripts\geometric\_calibration.py

focal length = 1 일 경우의 센서 좌표를 구한다.

(줌인 했을 경우, pixel 좌표값이 커짐)

sensor 좌표 = 픽셀을 principal 기준 좌표계로 이동 후, focal로 나눈다.

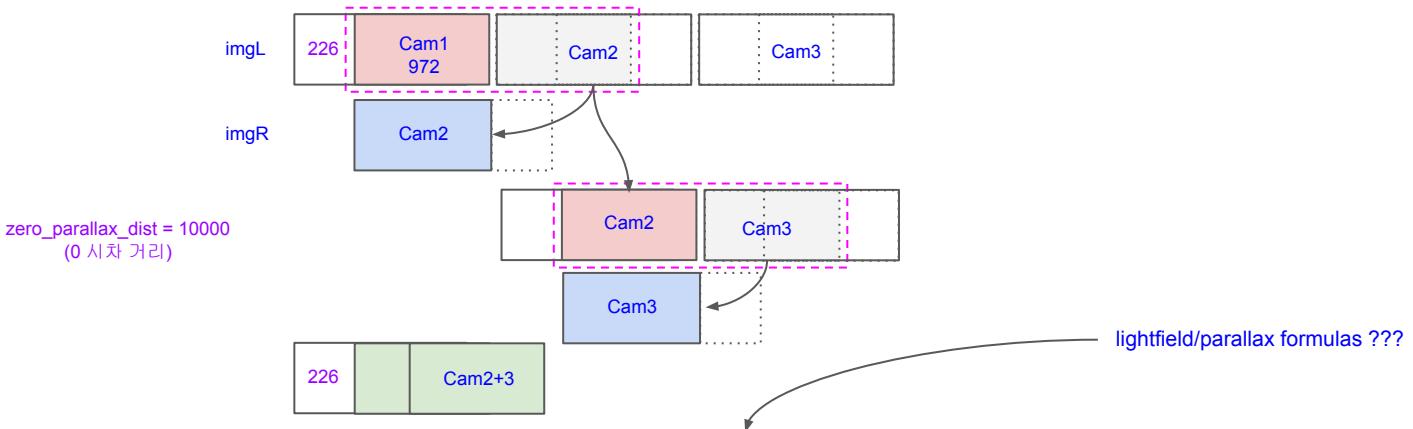


```
// compute rig coordinates, returns a ray, inverse of pixel()
Ray rig(const Vector2& pixel) const {
    // 거리 계산을 위해서는 focal = 1에서의 좌표로 변경해줘야 한다!
    // transform from pixel to distorted sensor coordinates
    Vector2 sensor = (pixel - principal).cwiseQuotient(focal); // 뒷 연산

    // transform from distorted sensor coordinates to unit camera vector
    Vector3 unit = sensorToCamera(sensor);
    -> (중심에서) 센서 좌표 거리에 따라 undistort()

    // transform from camera space to rig space
    return Ray(position, rotation.transpose() * unit); // origin, direction ?
}
```

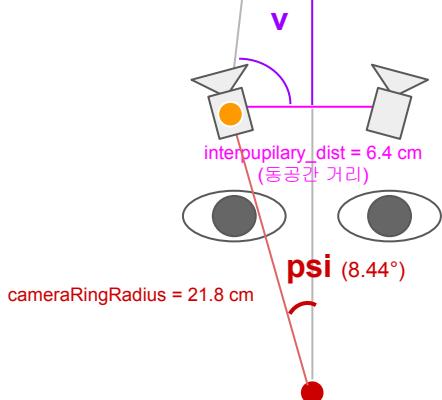
## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)



```
// generateRingOfNovelViewsAndRenderStereoSpherical(camFovHorizontalDegrees/*137°*/, ...)
tan(v) = zero_parallax_dist / (interpupillary_dist / 2); // tan(v) = 10000 / (6.4 / 2) = 3125
v = atan(3125) = 89.98 도 -> 아래 라인 sin(v) = sin(89.98) = 0.99999
```

$\sin(\psi) = \sin(v) * (\text{interpupillary\_dist} / 2) / \text{cameraRingRadius}$ ; // TODO:  $\sin(v)/*거의 1*/$ 를 왜 곱하나???
 $\psi = \arcsin((6.4 / 2) / 21.8) = \arcsin(0.14678899) = 8.44$ 도(0.14732)
--> TODO: 카메라 2대간의 각도는 약 17도(16.88), 360도를 커버하려면 21대 이상 필요?!

```
fovHorizontalRadians = toRadians(camFovHorizontalDegrees/*137°*/);
vergeAtInfinitySlabDisplacement =
     $\psi / 0.14732 * (\text{float}(\text{camImageWidth} / 1198) / \text{fovHorizontalRadians} / 2.39(1.195 * 2))$ ; // 73.85
--> 1198 폭에서 psi 각도가 차지하는 폭 (1 라이안당 폭 * psi 임)
```

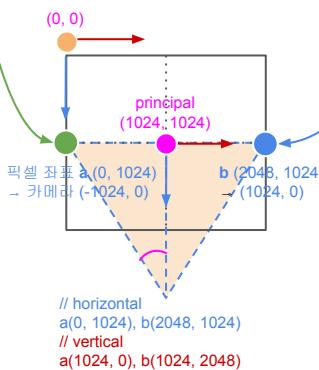


```
const float theta = -M_PI / 2.0f + v + psi; // v 가 거의 M_PI/2 이므로, theta = psi
const float zeroParallaxNovelViewShiftPixels =
     $\text{float}(\text{FLAGS_eqr_width}) * (\theta / (2.0f * M_PI))$ ; // 6300 * (0.14732 / (2 * 3.14)) = 147.78917 px
--> 동공간 거리 절반씩 이동! (전체 폭에서 theta 만큼의 폭을 리턴)
```

# Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)

// FOV(화각) 절반 각도를 리턴!!! (from resolution, forward)

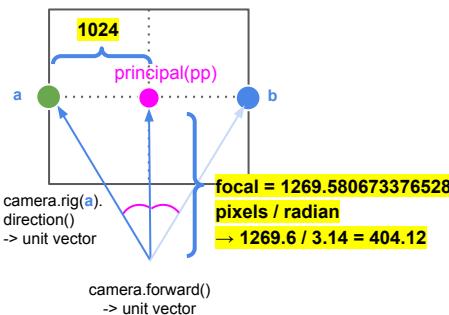
```
// measured in radians from forward
float approximateFov(const Camera& camera, const bool vertical) {
    Camera::Vector2 a = b = camera.principal;
    if (vertical) a.y() = 0; b.y() = camera.resolution.y();
    else a.x() = 0; b.x() = camera.resolution.x();
    return acos(max( // 카메라 방향(forward)과 a, b 벡터 사이의 작은 각도
        camera.rig(a).direction().dot(camera.forward()),
        camera.rig(b).direction().dot(camera.forward())));
}
```



// 2차원 픽셀 좌표의 3차원 rig coordinates(월드 좌표)를 리턴  
using Ray = Eigen::ParametrizedLine<Real, 3>;

```
// compute rig coordinates, returns a ray, inverse of pixel()
Ray Camera::rig(const Vector2& pixel) const {
    // transform from pixel to distorted sensor coordinates
    Vector2 sensor = (pixel - principal).cwiseQuotient(focal); // 월드
    // transform from distorted sensor coordinates to unit camera
    Vector3 unit = sensorToCamera(sensor); // 렌즈 undistort !!!
    // transform from camera space to rig space
    return Ray(position, rotation.transpose() * unit); // R, t
}
```

A dot B =  $\|A\| \|B\| \cos \theta$   
 $\|A\| = \|B\| = 1$  이면  
 $A \cdot B = \cos \theta, \theta = \arccos(A \cdot B)$   
 즉,  $\arccos(\text{camera.rig}(a).\text{direction}().\text{dot}(\text{camera.forward}()))$   
 는 FOV 의 절반 각도임  
 또는  $\arctan(1024/404.12) = \arctan(2.534) = 1.195 = 68.5$ 도  
 실제 스펙은 142도 (vs 137도)



// 3차원 rig coordinates 의 2차원 픽셀 좌표를 리턴

```
// compute pixel coordinates
Vector2 Camera::pixel(const Vector3& rig) const {
    // transform from rig to camera space
    Vector3 camera = rotation * (rig - position);
    // transform from camera to distorted sensor coordinates
    Vector2 sensor = cameraToSensor(camera);
    // transform from sensor coordinates to pixel coordinates
    return focal.cwiseProduct(sensor) + principal; // coefficientWise product(계수끼리 곱, Hadamard product)
}
```

Camera(const dynamic& json)  
> position = deserializeVector<3>(json["origin"]);

setRotation(  
 deserializeVector<3>(json["forward"]),
 ~["up"], ~["right"]));

group, resolution(2048, 2048), principal(1024, 1024) 멤버 등을 설정

void Camera::setRotation(const Vector3& forward, up, right)  
 CHECK\_LT(right.cross(up).dot(forward), 0) << "rotation must be right-handed"; //

음수

rotation.row(2) = -forward; // backward 값을 저장  
 rotation.row(1) = up; // +y is up  
 rotation.row(0) = right; // +x is right

Eigen::AngleAxis<Camera::Real> aa(rotation);  
 rotation = aa.toRotationMatrix(); // Matrix3 rotation; 멤버변수

0: [ 0, -1, 0 ]  
 1: [ 0, 0, 1 ]  
 2: [ 1, 0, 0 ]  
 → "cam1"의 경우

Vector3 Camera::backward() const { return rotation.row(2); }  
 Vector3 Camera::forward() const { return -backward(); }

## Render (Surround360\surround360\_render\source\test\TestRenderStereoPanorama.cpp)

```

int main(int argc, char** argv) {
    renderStereoPanorama()
}

// renderStereoPanorama()

Mat bottomSpherical;
1) prepareBottomImagesThread(const RigDescription& rig, Mat* bottomSpherical) 에서는
if (FLAGS_enable_pole_removal) 이면
    combineBottomImagesWithPoleRemoval(..., bottomImage) 에 하단 카메라 1, 2에 대해
    Pole 마스크 이미지를 지워서 합치고
else
    bottomImage 에 그냥 하단 첫번째 카메라 이미지를 로딩
bottomSpherical 인자에 6300 x (3072 * camera.getFov()/*1.61443(92.5도)*/ / M_PI) 크기로
생성 후, bicubicRemapToSpherical(*bottomSpherical, bottomImage, ...)로 bottomSpherical
에 구좌표계로 투영

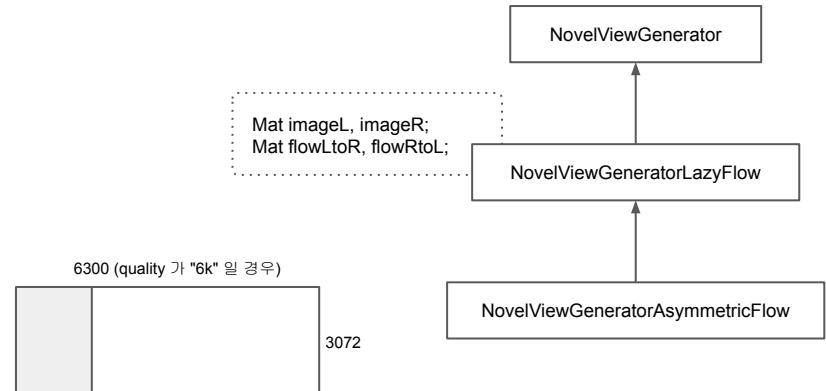
2) prepareTopImagesThread(rig, Mat* topSpherical) 에서 동일한 크기로 구좌표계로 투영

vector<Mat> projectionImages;
3) projectSphericalCamImages(rig, FLAGS_imgs_dir, FLAGS_frame_number,
projectionImages);
side 카메라 이미지를 구좌표계에 투영

Mat sphericalImageL, sphericalImageR;
4) generateRingOfNovelViewsAndRenderStereoSpherical(..., Mat& panolmageL/R, ...)에서
for () projectionImages.size() 만큼 prepareNovelViewGeneratorThread() 쓰레드로
NovelViewGeneratorAsymmetricFlow::prepare()에서
NovelViewGeneratorLazyFlow::imageL/R 저장 ~::flowRtoL에 옵피컬 플로우를 저장
다시 for () projectionImages.size() 만큼 renderStereoPanoramaChunksThread() 쓰레드로
로컬 변수 &panoChunksL[leftIdx]에 저장, panoChunksL 를 가로로 붙어서 panolmageL
전달인자(left/right eye equirect panorama 임)에 저장

5) poleToSideFlowThread 쓰레드에서 사이드와 상하카메라를 연결해
topSphericalWarpedL/R 를 저장, bottomSphericalWarpedL/R 에도 저장

```



**projectionImages[camIdx].create(**  
 FLAGS\_eqr\_height \* vRadians / M\_PI,  
 FLAGS\_eqr\_width \* hRadians / (2 \* M\_PI),  
 CV\_8UC4); --> 회색박스 크기로 생성됨  
 360 도 기준으로 hRadians 각도만큼 썹  
 FLAGS\_eqr\_width 를 뜯어서 이미지로 생성

6) if (FLAGS\_cubemap\_width > 0 && ...) 이면 큐브맵도 저장

7) sphericalImageL/R 을 상하로 연결해서 stereoEquirect 이미지를 생성  
`Mat stereoEquirect = stackVertical(vector<Mat>{sphericalImageL, sphericalImageR});`  
 생성된 stereoEquirect 이미지를 FLAGS\_output\_equirect\_path 파일에 저장  
 (e.g. ~/Desktop/test/render/eqr\_frames / eqr\_000000.png)

# Render

## Remapping

- 한 이미지의 픽셀을 다른 이미지의 다른 위치로 이동시키는 작업  
(It is the process of taking pixels from one place in the image and locating them in another position in a new image).
- 일대일 매핑이 안될 수 있으므로 보간이 필요
- 타겟 이미지의 모든 픽셀의 값을 원본 이미지의 픽셀에서 보간을 사용해서 찾는다!!!  
What remap() does do is, for every pixel in the destination image, lookup where it comes from in the source image, and then assigns an interpolated value.

$$dst(x, y) = \text{src}(\text{map}_x(x, y), \text{map}_y(x, y))$$

e.g.)

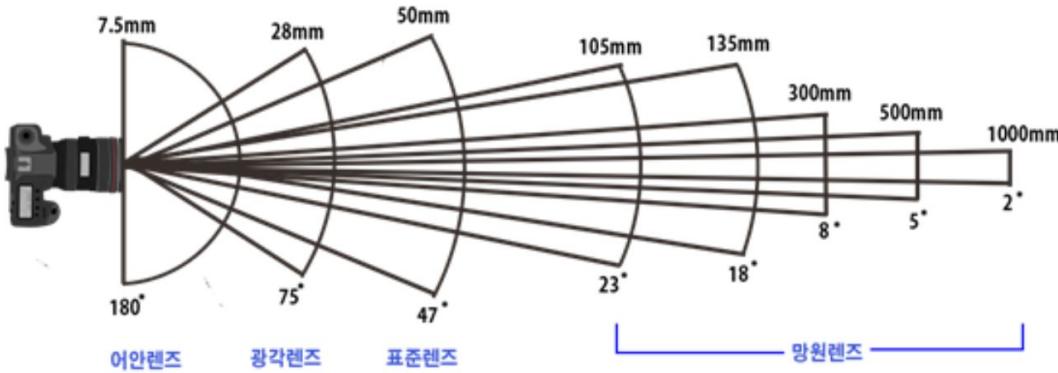
```
remap(tmp, dst, warp, Mat(), CV_INTER_CUBIC, BORDER_CONSTANT);
```

```
void cv::remap(  
    InputArray src,  
    OutputArray dst,  
    InputArray map1, // The first map of either (x,y) points or just x values  
    InputArray map2, // The second map of y values, empty map if map1 is (x,y) points  
    int interpolation,  
    int borderMode = BORDER_CONSTANT,  
    const Scalar & borderValue = Scalar()  
)
```

The function remap transforms the source image using the specified map:

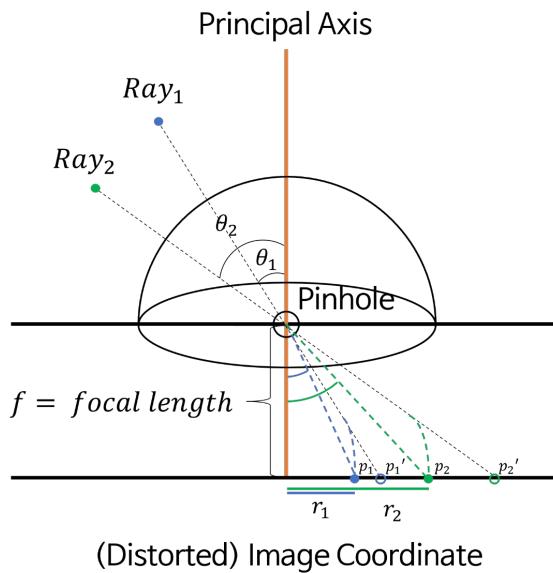
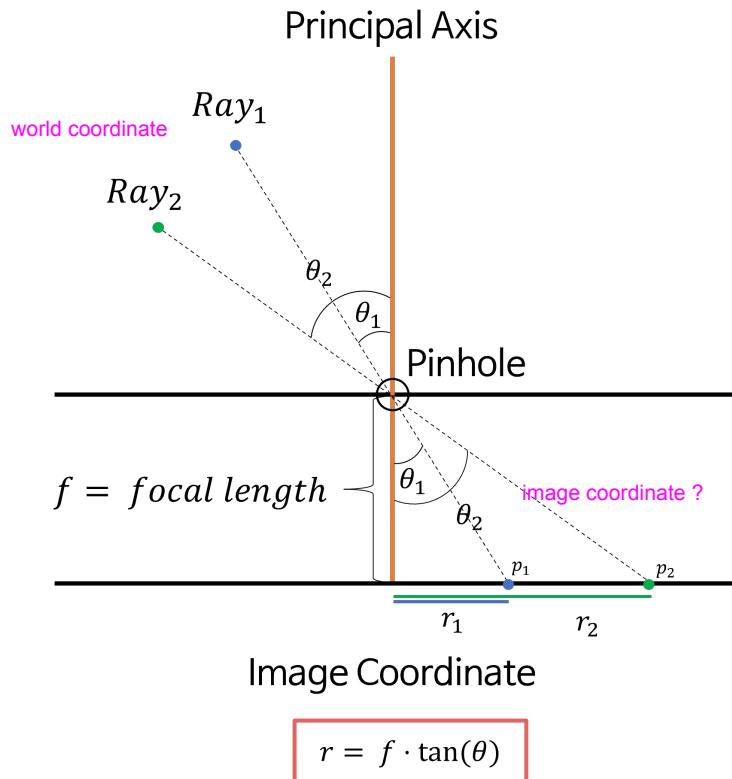
-->  $dst(x, y) = \text{src}(\text{map}_x(x, y), \text{map}_y(x, y))$

--> dst 이미지의 (x, y) 픽셀은 원본 이미지에서  $\text{map}_x(x, y)$ ,  $\text{map}_y(x, y)$ 로 매핑된 위치의 픽셀을 가져온다.

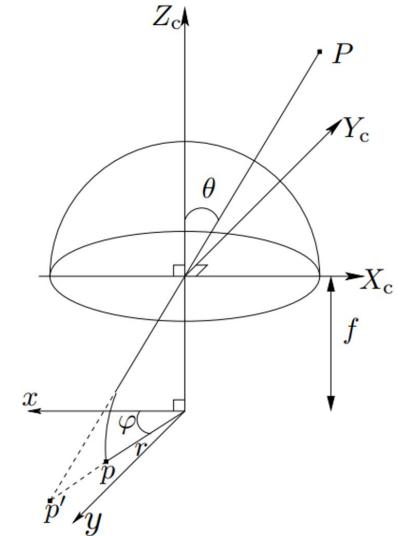


```
cv2.remap(src, map1, map2, interpolation, dst=None, borderMode=None,
           borderValue=None) -> dst
```

- src: 입력 영상
- map1: 결과 영상의 (x, y) 좌표가 참조할 입력 영상의 x좌표.  
입력 영상과 크기는 같고, 타입은 np.float32인 numpy.ndarray.
- map2: 결과 영상의 (x, y) 좌표가 참조할 입력 영상의 y좌표.
- interpolation: 보간법
- dst: 출력 영상
- borderMode: 가장자리 픽셀 확장 방식. 기본값은 cv2.BORDER\_CONSTANT.
- borderValue: cv2.BORDER\_CONSTANT일 때 사용할 상수 값. 기본값은 0.

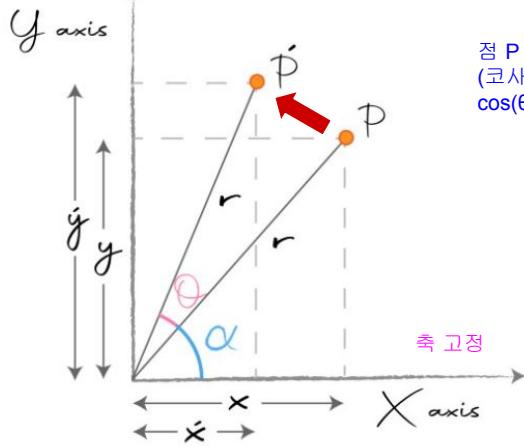


$$\frac{\theta_1}{r_1} = \frac{\theta_2}{r_2} \quad r = f \cdot \theta$$



넓은 화각(넓은 영역), 선명한 영상(많은 빛) 확보를 위해 렌즈를 사용  
렌즈를 사용하면 빛이 직진하지 않게되어 (굴절 때문에) 왜곡이 발생됨  
<https://gaussian37.github.io/vision-concept-calibration/>

## 1) 좌표 변환 (Change of coordinates)을 이용한 회전 (Rotation)



점 P 가  $\theta$  만큼 회전했을 때의 좌표 P'  
(코사인 법칙  
 $\cos(\theta + \alpha) = \cos(\theta) * \cos(\alpha) - \sin(\theta) * \sin(\alpha)$  활용)

$$\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

coordinate transform

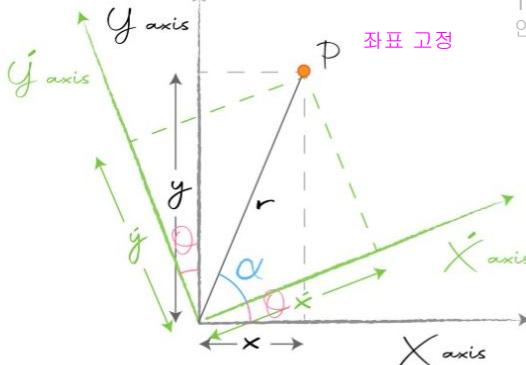
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3차원 평면에서의 회전 변환

## 2) 좌표계(좌표축) 변환 (Change of basis)을 이용한 회전 (Rotation)

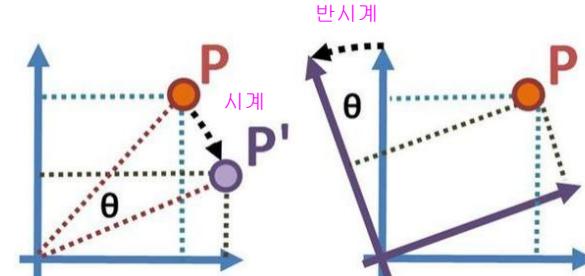


1) 이에서의 x', y' 과는 상관없지만  
연관 짓자면 P' 를 P 로 변환에 해당

$$\begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix}$$

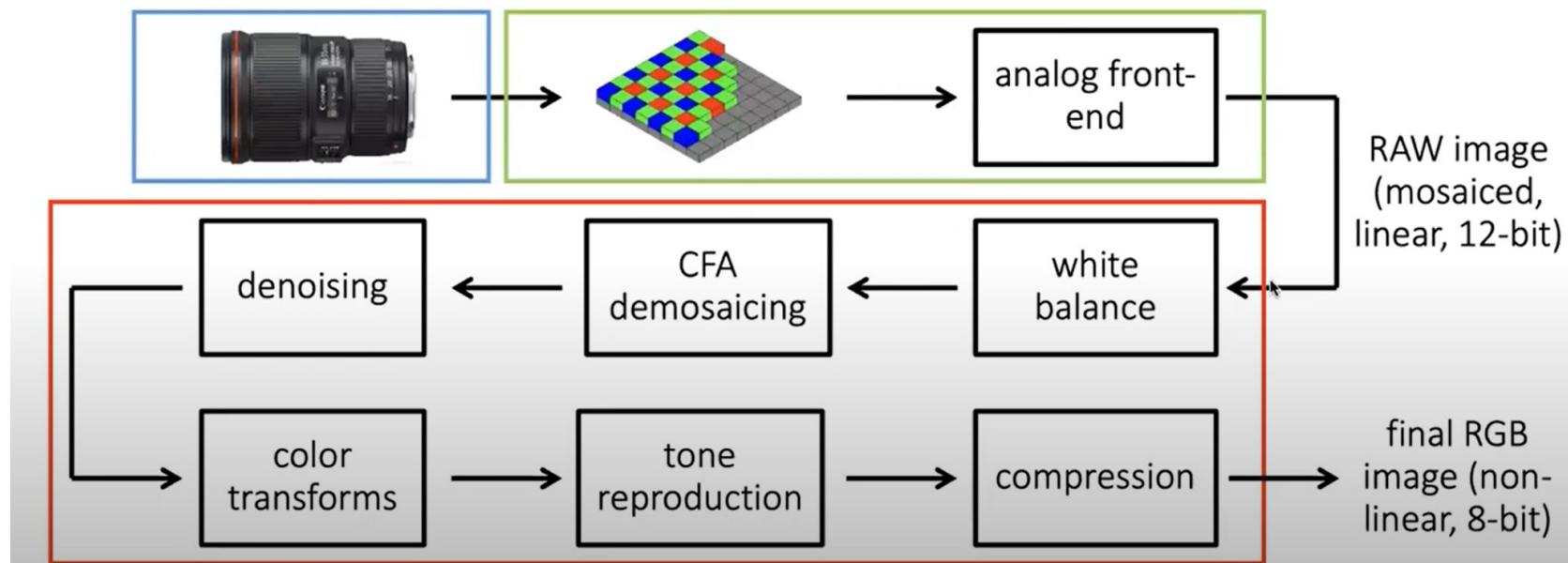
basis transform

$$R = \begin{bmatrix} \cos\alpha \cos\beta & \cos\alpha \sin\beta \sin\gamma - \sin\alpha \cos\gamma & \cos\alpha \sin\beta \cos\gamma + \sin\alpha \sin\gamma \\ \sin\alpha \cos\beta & \sin\alpha \sin\beta \sin\gamma + \cos\alpha \cos\gamma & \sin\alpha \sin\beta \cos\gamma - \cos\alpha \sin\gamma \\ -\sin\beta & \cos\beta \sin\gamma & \cos\beta \cos\gamma \end{bmatrix}$$



# The (in-camera) image processing pipeline

The sequence of image processing operations applied by the camera's image signal processor (ISP) to convert a RAW image into a "conventional" image.



# Quick notes on terminology

- Sometimes the term *image signal processor* (ISP) is used to refer to the image processing pipeline itself.
- The process of converting a RAW image to a “conventional” image is often called *rendering* (unrelated to the image synthesis procedure of the same name in graphics).
- The inverse process, going from a “conventional” image back to RAW is called *derendering*.

# White balancing

Human visual system has *chromatic adaptation*:

- We can perceive white (and other colors) correctly under different light sources.
- Cameras cannot do that (there is no “camera perception”).

White balancing: The process of removing color casts so that colors that we would *perceive* as white are *rendered* as white in final image.



different whites

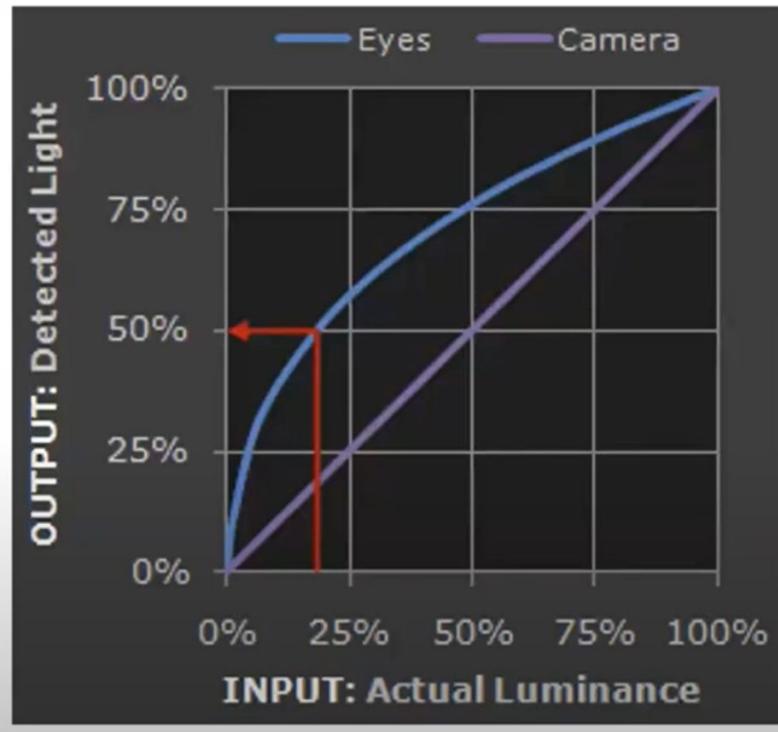


image captured  
under fluorescent



image white-  
balanced to daylight

# Perceived vs measured brightness by human eye



어두운 색에 대해 더 민감해서 어두운 색의 변화는 잘 인지하고  
밝은 빛의 변화는 둔감하다.

We have already seen that sensor response is linear.

Human-eye *response* (measured brightness) is also linear.

However, human-eye *perception* (perceived brightness) is *non-linear*:

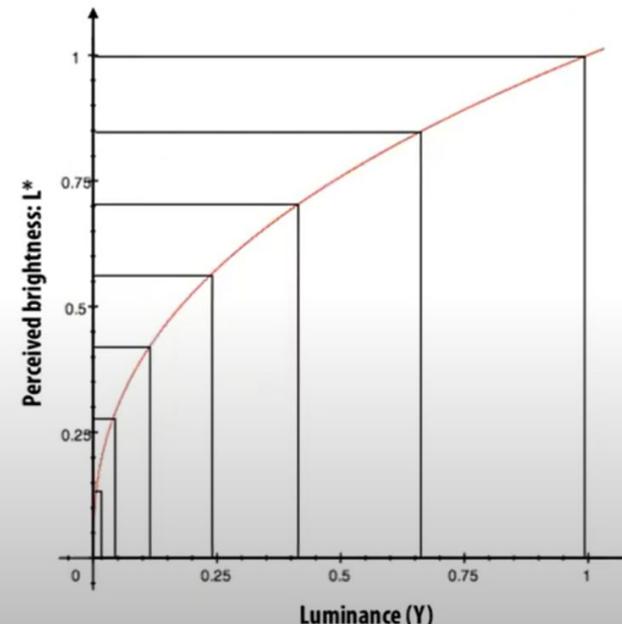
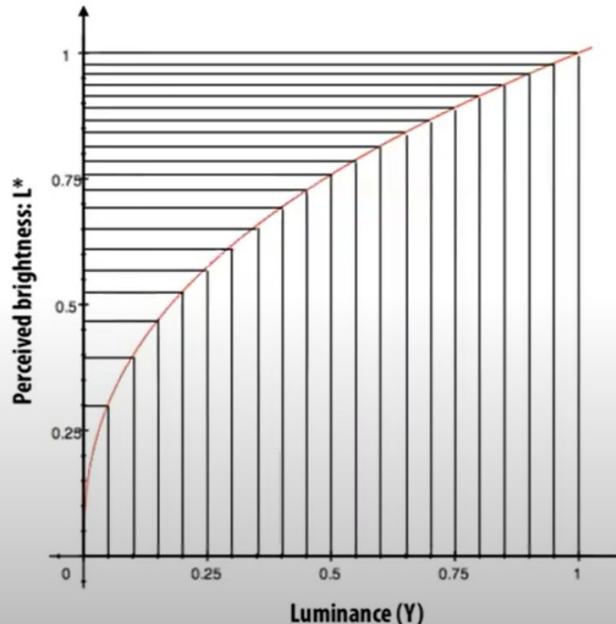
- More sensitive to dark tones.
- Approximately a Gamma function.

# Gamma encoding

After this stage, we perform compression, which includes changing from 12 to 8 bits.

- Apply non-linear curve to use available bits to better encode the information human vision is more sensitive to.

예민한 부분에 좀더 촘촘하게 값을 저장(저장 용량 대비 효율적임)



# Demonstration

original (8-bits, 256 tones)



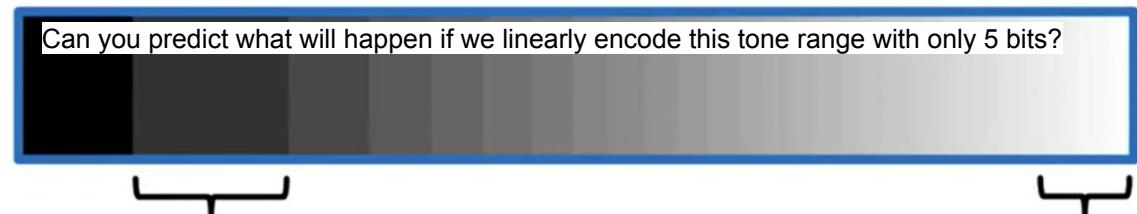
linear encoding (5-bits, 32 tones)

인간의 시각은 베버의 법칙(Weber's law)에 따라  
밝기의 변화에 대해 비선형적으로 반응한다.

(자극의 변화를 느낄 수 있는 최소 변화량은 처음  
자극의 세기에 비례)

따라서, 선형 인코딩시 부드럽지 못하고 단절되  
보임(posterization)

Can you predict what will happen if we linearly encode this tone range with only 5 bits?

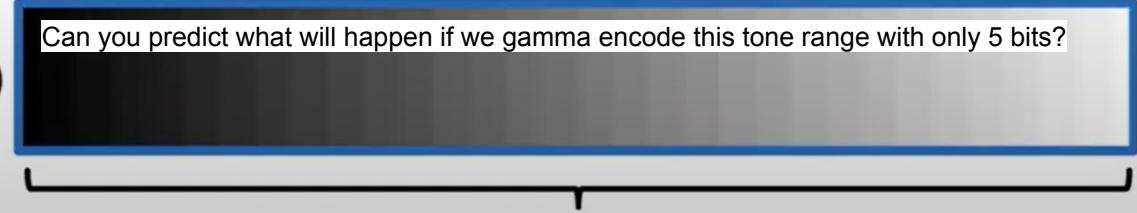


all of this range gets  
mapped to just one tone

all of these tones  
look the same

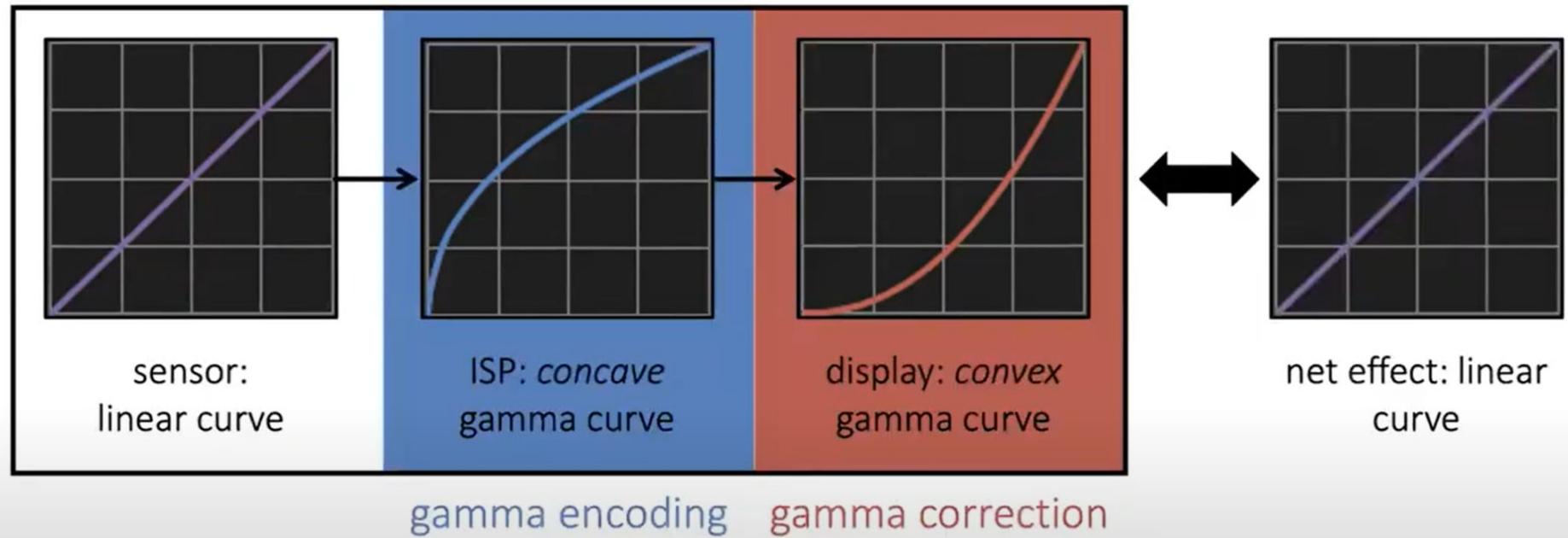
gamma encoding (5-bits, 32 tones)

Can you predict what will happen if we gamma encode this tone range with only 5 bits?



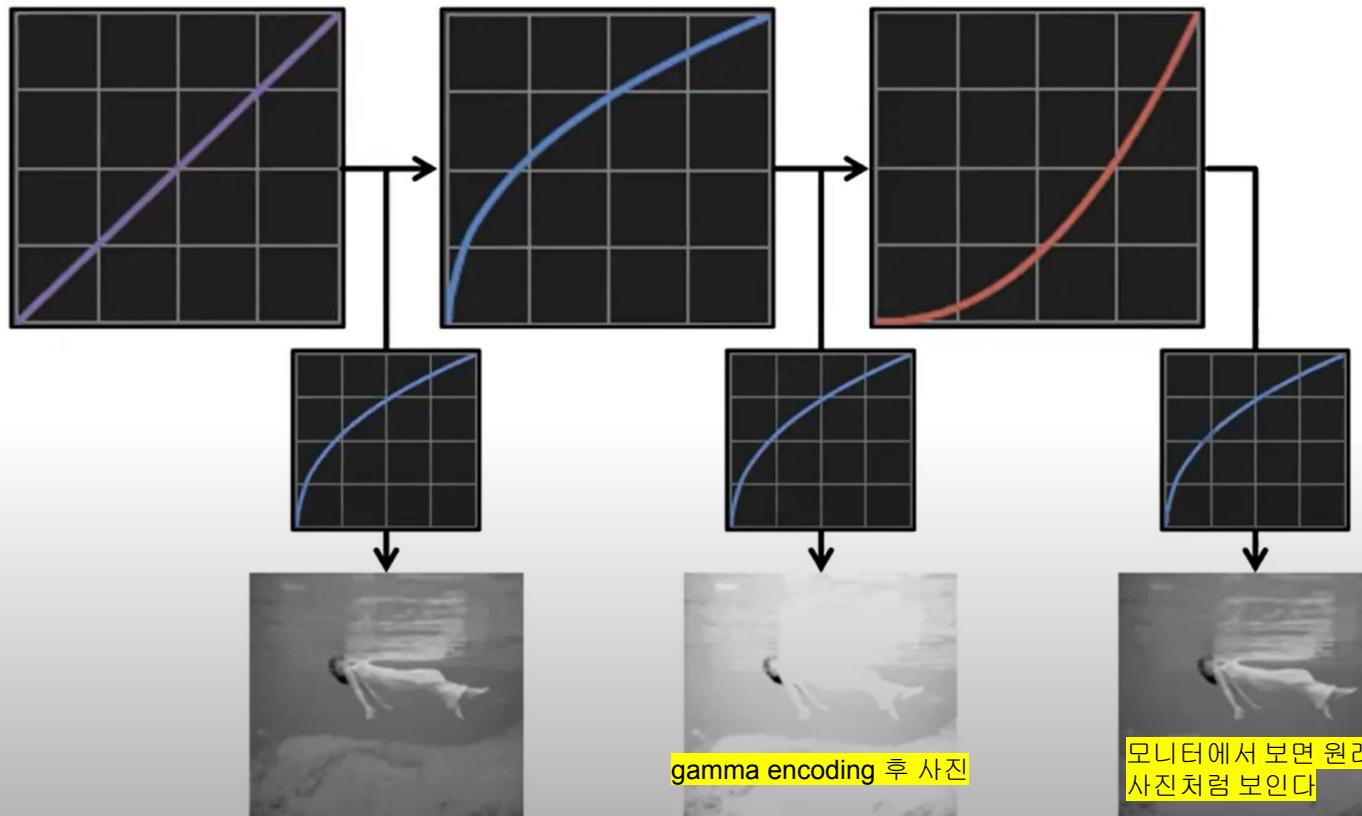
tone encoding becomes a lot  
more perceptually uniform

# Tone reproduction pipeline



모니터가 어두운 값을 낮춰주고, 밝은 값을 유지해서 linear하게 표시된다.

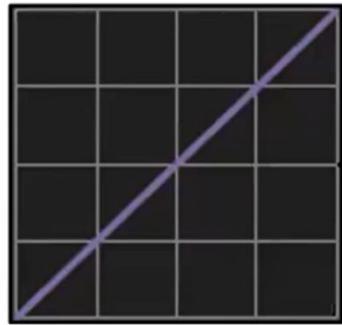
# Tone reproduction pipeline



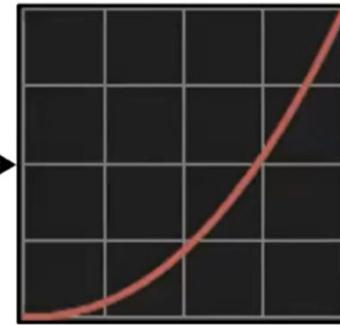
human visual system: *concave*

image a human would see at different stages of the pipeline

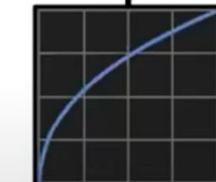
# RAW pipeline



gamma encoding  
is skipped!



display still applies  
gamma correction!



human visual  
system: *concave*  
gamma curve



RAW image appears  
very dark! (Unless you  
are using a RAW viewer)

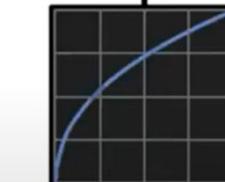


image a human  
would see at  
different stages of  
the pipeline

# Historical note

- CRT displays used to have a response curve that was (almost) exactly equal to the inverse of the human sensitivity curve. Therefore, displays could skip gamma correction and display directly the gamma-encoded images.
- It is sometimes mentioned that gamma encoding is done to undo the response curve of a display. This used to (?) be correct, but it is not true nowadays. Gamma encoding is performed to ensure a more perceptually-uniform use of the final image's 8 bits.

# Gamma encoding curves

The exact gamma encoding curve depends on the camera.

- Often well approximated as  $L^\gamma$ , for different values of the power  $\gamma$  ("gamma").
- A good default is  $\gamma = 1 / 2.2$ .

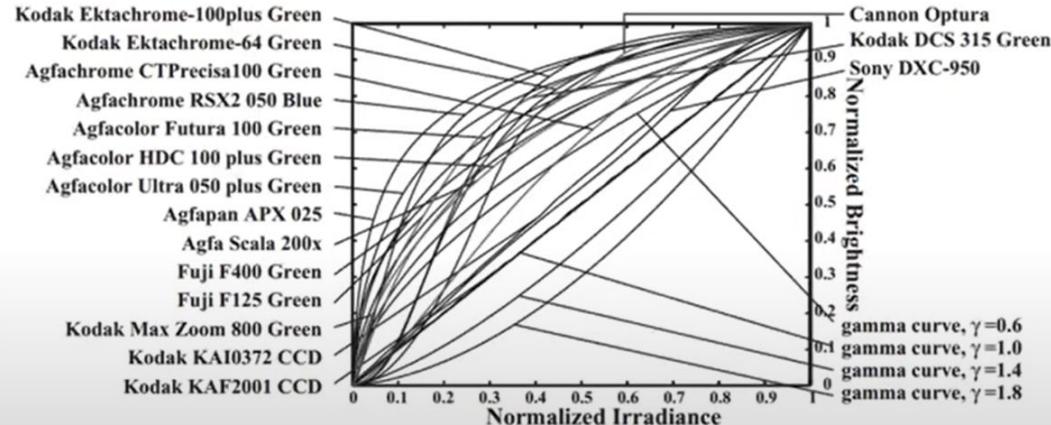
gamma encoding 은 모든 카메라마다 조금씩 다르지만  
기본적인 비슷한 값을 얻고자 한다면 2.2 승



before gamma



after gamma



Warning: Our values are no longer linear relative to scene radiance!

# Do I ever need to use RAW?

Emphatic yes!

- Every time you use a physics-based computer vision algorithm, you *need linear measurements of radiance*.
- Examples: photometric stereo, shape from shading, image-based relighting, illumination estimation, anything to do with light transport and inverse rendering, etc.
- Applying the algorithms on non-linear (i.e., not RAW) images will produce completely invalid results.