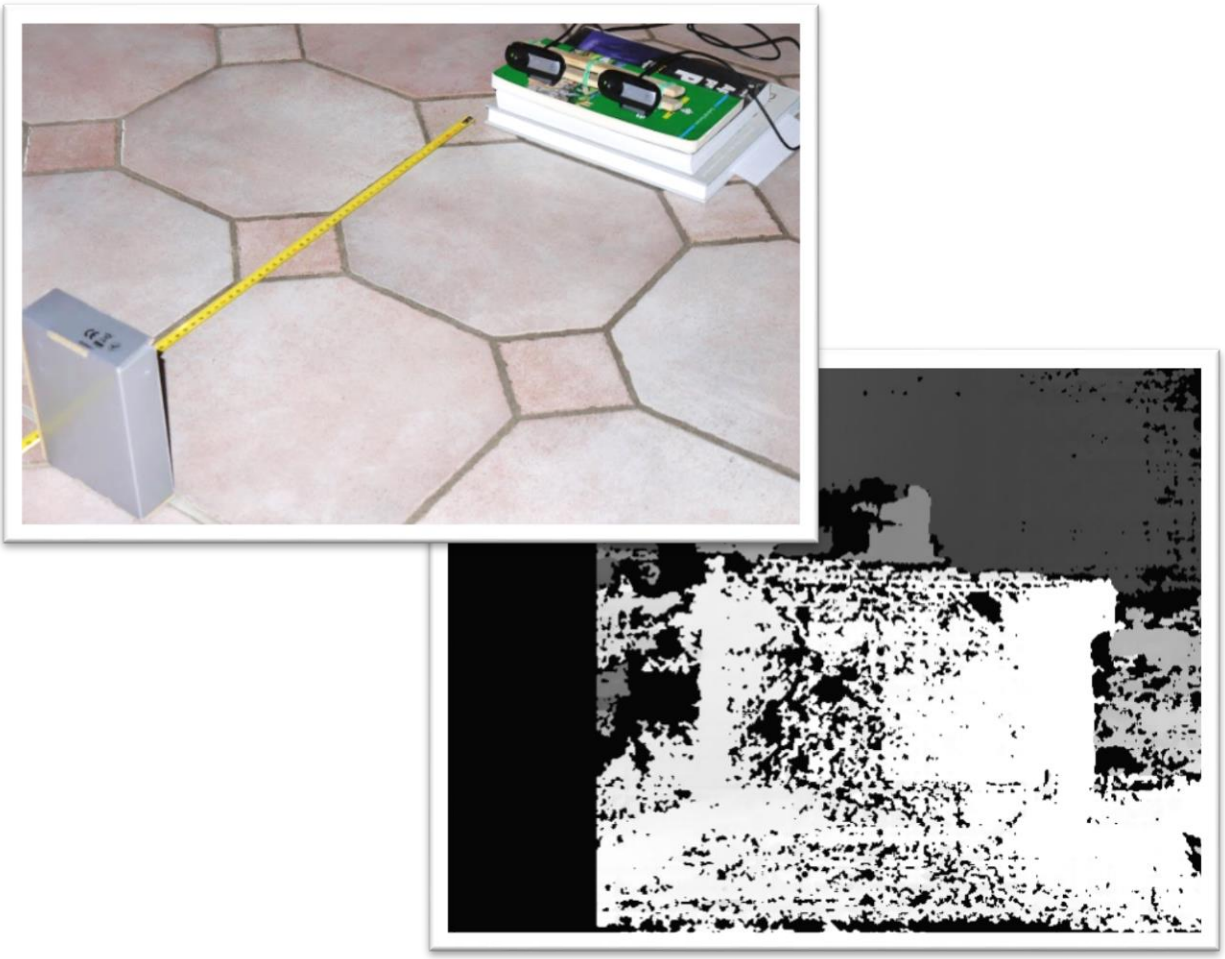
Stereo-Vision



작가:
우르바일러 프레데릭 그리고 부자시노비치 스테판
입학 번호 58566 EIT 입학 번호: 59092 메카트로닉스

이메일: uhfr1011@hs-karlsruhe.de 이메일: vust1011@hs-karlsruhe.de

간병인:Dr.-Ing. 페르디난드 올라프스키

프로젝트 코디네이터 MMT:교수 박사 피터 웨버

프로젝트 코드:17SS_OL_클린룸

사선:2017년 9월 30일

목차

일러스트 목록

코드 디렉토리

1. 소개

스테레오 비전을 이해하려면 먼저 간단한 카메라의 작동 방식, 기본 구성 방식, 조절해야 하는 매개변수를 설명해야 합니다.

A. 스테레오 비전이란 무엇입니까?

입체영상은 동일한 장면에 대한 2개의 이미지를 촬영한 다음 해당 장면의 시차 지도를 작성하는 프로세스입니다. 이 시차 맵을 통해 물체까지의 거리를 측정하고 장면의 3D 맵을 생성할 수 있습니다.

B. 클린룸 로봇의 스테레오 비전

대규모 프로젝트의 목표는 클린룸 전체에서 입자 측정을 수행할 수 있는 클린룸 로봇을 개발하는 것입니다. 문제 없이 측정을 수행하려면 로봇이 충돌 없이 방향을 잡아야 합니다. 이 방향에서는 카메라만 사용하고 물체까지의 거리를 측정하려면 최소한 두 대의 카메라(스테레오 비전)가 필요하기로 결정했습니다. 두 카메라 사이의 거리는 110mm입니다.

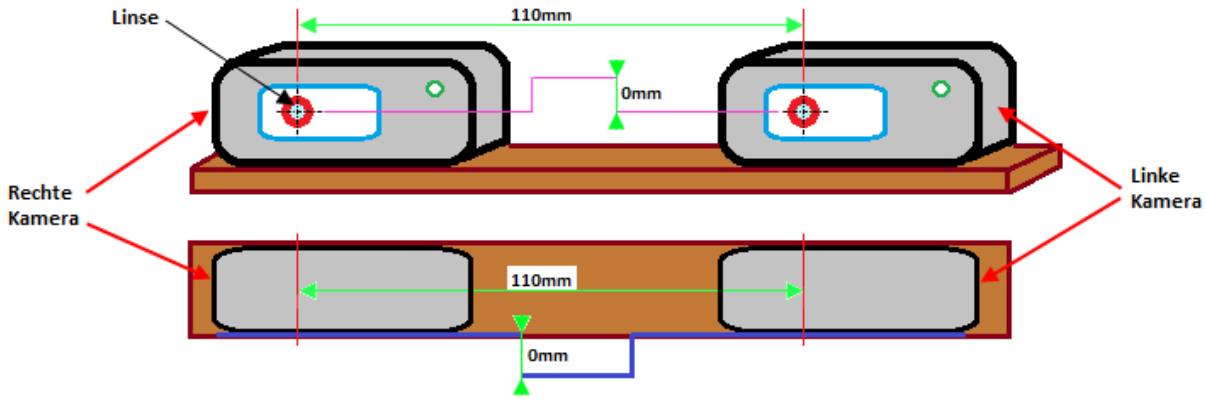


그림 1: 스테레오 카메라의 회로도

이 거리가 클수록 멀리 있는 물체의 장면까지의 거리를 더 잘 평가할 수 있습니다.

이 프로젝트에서는 OpenCV 라이브러리를 사용하는 Python 프로그램을 구현하여 카메라를 보정하고 장면의 객체까지의 거리를 측정했습니다. (부록 참조)

그림 2: OpenCV 및 Python 로고

프레데릭 우르바일러
스테판 부자시노비치

스테레오 카메라는 두 개의 Logitech Webcam C170으로 구성됩니다.

그림 3: Logitech 웹캠 C170의 사진(출처: www.logitech.fr)

비디오 보기는 640x480 픽셀의 이미지에 최적입니다. 초점 거리: 2.3mm. 자체 제작 스테레오 카메라:

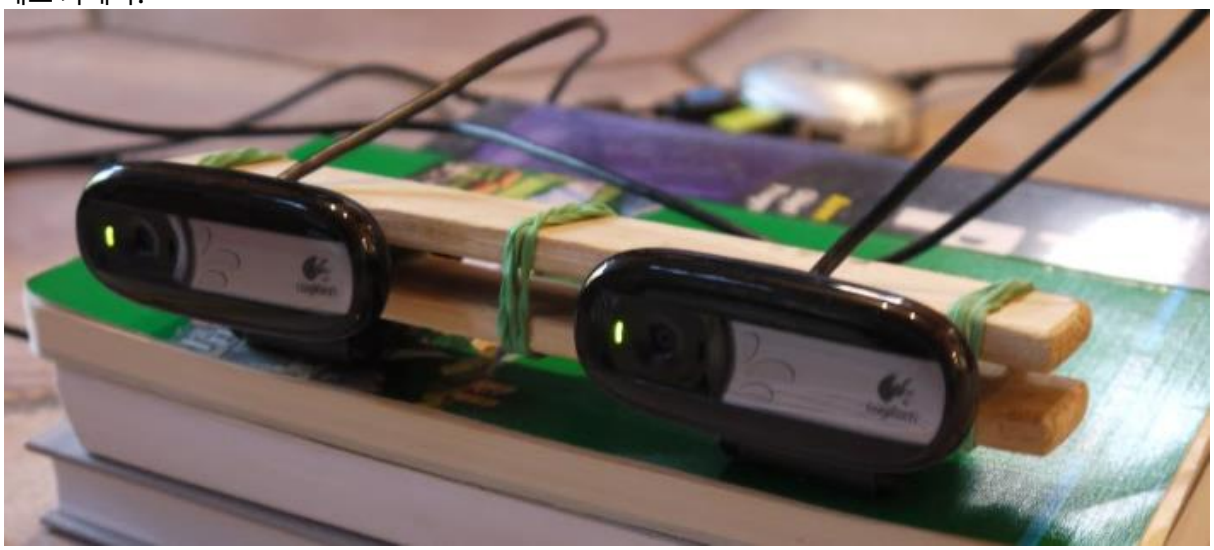


그림 4: 자체 제작한 스테레오 카메라 사진

2. 카메라 모델

카메라는 우리 주변 환경의 광선을 포착합니다. 원칙적으로 카메라는 우리의 눈처럼 작동하며, 주변 환경에서 반사된 광선이 우리 눈에 와서 망막에 수집됩니다.

"핀홀 카메라"는 가장 간단한 모델입니다. 카메라 작동 방식을 이해하는 데 도움이 되는 단순화된 모델입니다. 모델에서 모든 광선은 표면에 의해 차단됩니다. 구멍을 통과한 광선만 기록되어 카메라 표면에 역으로 투사됩니다. 아래 그림은 이 원리를 설명합니다.

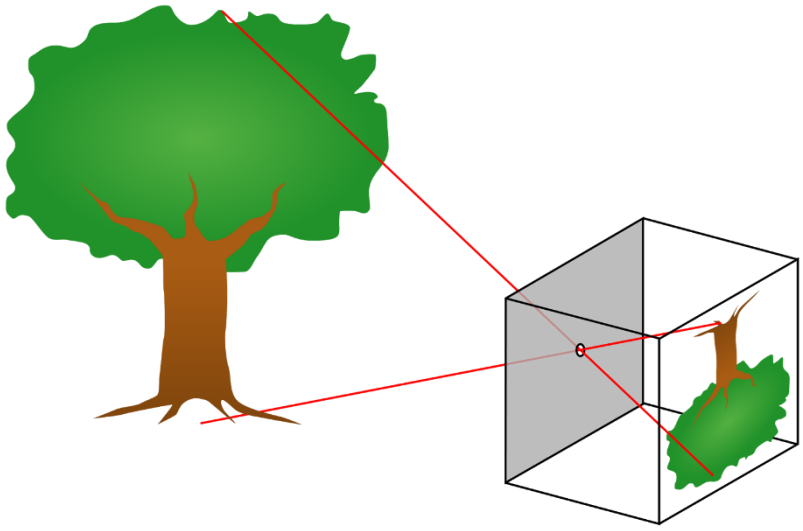


그림 5: 카메라 작동 방식

원천:https://funsizephysics.com/use-light-turn-world-upside/

이 원리는 매우 간단하지만 빠른 노출로 충분한 빛을 포착하는 좋은 방법은 아닙니다. 따라서 렌즈는 더 많은 빛을 한곳에 모으기 위해 사용됩니다. 문제는 이 렌즈가 왜곡을 가져온다는 것입니다.

두 가지 유형의 왜곡을 구별할 수 있습니다.

- 방사형 왜곡
- 접선 왜곡

방사형 왜곡은 렌즈 자체의 모양에서 발생하고 접선 왜곡은 카메라의 기하학적 구조에서 발생합니다. 그런 다음 수학적 방법을 사용하여 이미지를 수정할 수 있습니다.

교정 프로세스를 통해 카메라의 기하학적 모델과 렌즈 왜곡 모델을 형성할 수 있습니다. 이러한 모델은 카메라의 고유 매개변수를 형성합니다.

A. 초점 거리

카메라 표면에 투사되는 이미지의 상대적 크기는 초점 거리에 따라 다릅니다. 핀홀 모델에서 초점 거리는 구멍과 이미지가 투사되는 표면 사이의 거리입니다.

탈레스 정리는 다음을 제공합니다: -x = f * (X / Z) 다음과 같이:

- x: 물체의 이미지(마이너스 기호는 반전되는 이미지에서 나옵니다)
- X: 물체의 크기
- Z: 구멍에서 물체까지의 거리
- f: 초점 거리, 구멍에서 이미지까지의 거리

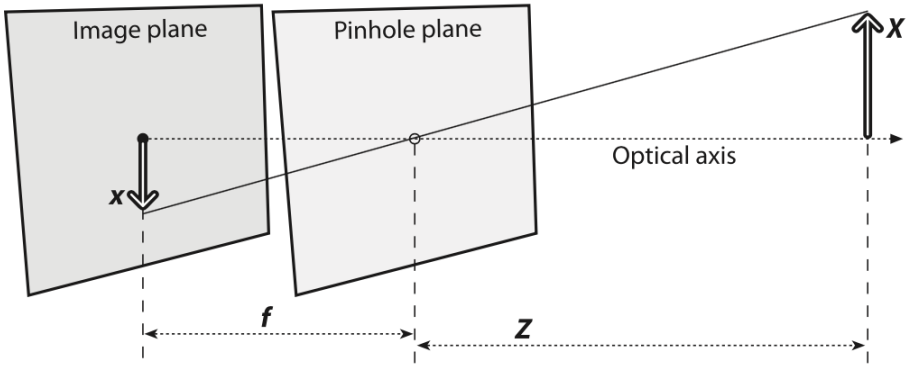


그림 6: 투영된 물체

출처: OpenCV 3 학습 - O'Reilly

렌즈가 완벽하게 중앙에 위치하지 않기 때문에 렌즈의 수평 및 수직 변위에 대해 각각 Cx 및 Cy라는 두 가지 매개변수가 도입됩니다. 이미지 영역이 직사각형이기 때문에 X축과 Y축의 초점 거리도 다릅니다. 이는 표면 위의 물체 위치에 대한 다음 공식을 제공합니다.

$$x_{\text{screen}} = f_x\left(\frac{X}{Z}\right) + c_x, \qquad y_{\text{screen}} = f_y\left(\frac{Y}{Z}\right) + c_y$$

이미지 표면에 투영된 실제 세계의 점은 다음과 같은 방식으로 모델링할 수 있습니다. M은 여기서 고유 행렬입니다.

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

B. 렌즈의 왜곡

이론적으로 포물선형 렌즈를 이용하면 왜곡이 발생하지 않는 렌즈를 만드는 것이 가능합니다. 그러나 실제로는 포물선 렌즈보다 구면 렌즈를 만드는 것이 훨씬 쉽습니다. 앞서 설명한 것처럼 왜곡에는 두 가지 유형이 있습니다. 렌즈 모양에 따른 방사형 왜곡과 카메라 조립 과정에서 발생하는 접선 왜곡입니다.

광학 중심에는 방사형 왜곡이 없으며 가장자리에서 접근할수록 증가합니다. 실제로 이 왜곡은 작게 유지되며 세 번째 항까지 Taylor 확장을 수행하는 것으로 충분합니다. 다음 수식 결과입니다.

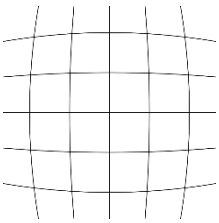


그림 7: 방사형 왜곡(출처: 위키피디아)

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

x와 y는 이미지 표면의 원래 지점의 좌표이며 보정된 지점의 위치를 계산하는 데 사용됩니다.

렌즈가 이미지 표면과 완벽하게 평행하게 구성되지 않았기 때문에 접선 왜곡도 있습니다. 이를 수정하기 위해 p1과 p2라는 두 개의 추가 매개변수가 도입되었습니다.

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$
$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

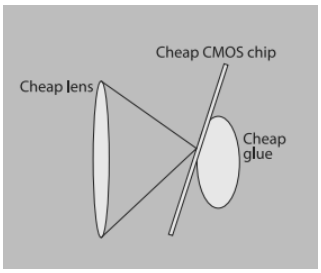


그림 8. 접선 왜곡(출처: Learning OpenCV 3 - O'Reilly)

C. OpenCV를 이용한 교정

OpenCV 라이브러리를 사용하면 특정 기능을 사용하여 고유 매개변수를 계산할 수 있으며, 이 프로세스를 교정이라고 합니다. 이것은 체스판의 다양한 관점을 통해 가능해졌습니다.

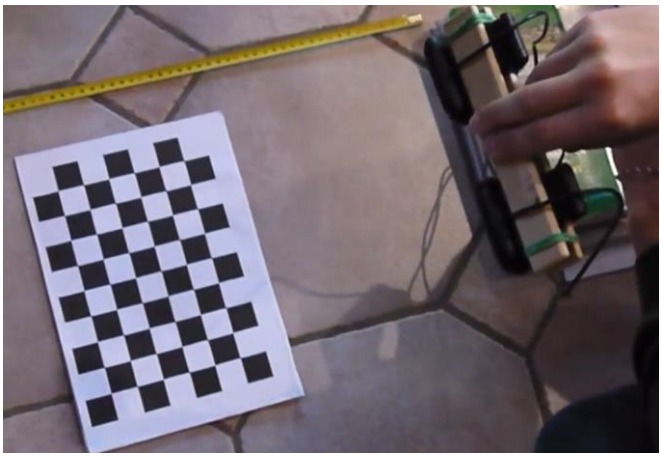


그림 9: 보정을 위해 이미지를 촬영하는 동안의 사진

나중에 교정을 위해 사진을 찍는 프로그램을 "**Take_images_for_calibration.py**”

두 카메라 모두에서 체커보드 모서리가 감지되면 각 카메라마다 감지된 이미지가 있는 두 개의 창이 열립니다. 그런 다음 이미지는 사용자에 의해 저장되거나 삭제됩니다. 모서리가 매우 선명하게 보이는 좋은 사진을 볼 수 있습니다. 이 이미지는 나중에 메인 프로그램의 보정에 사용될 것입니다.”
Main_Stereo_Vision_Prog.py" 사용된. OpenCV는 적절한 보정을 위해 각 카메라마다 최소 10개의 이미지를 보유할 것을 권장합니다. 각 카메라마다 50개의 이미지로 좋은 결과를 얻었습니다.

카메라를 보정하기 위해 Python 코드는 OpenCV 함수를 사용하여 각 카메라의 각 이미지에서 체스판 모서리를 검색합니다.*cv2.findChesssboard코너*

그런 다음 각 이미지의 모서리 위치는 이미지 벡터에 저장되고 3D 장면의 객체 포인트는 다른 벡터에 저장됩니다. 그런 다음 함수에서 이러한 Imgpoint 및 Objpoint를 사용합니다.*cv2.calibeCamera()*출력에서 카메라 매트릭스, 왜곡 계수, 회전 및 이동 벡터를 반환합니다.

기능*cv2.getOptimalNewCameraMatrix()*나중에 함수에서 사용할 정확한 카메라 매트릭스를 얻을 수 있습니다.*cv2.stereoRectify()*사용.

OpenCV로 보정한 후 카메라에 대해 다음과 같은 행렬 M을 얻습니다.

수정되지 않은 매트릭스 M(오른쪽 카메라):

885,439	0	301,366
0	885,849	233,812
0	0	1

매트릭스 M직사각형수정됨(오른쪽 카메라):

871,463	0	303,497
0	869,592	233,909
0	0	1

수정되지 않은 매트릭스 M(왼쪽 카메라):

748,533	0	345,068
0	749,062	228,481
0	0	1

매트릭스 M직사각형수정됨(왼쪽 카메라):

730,520	0	349,507
0	725,714	227,805
0	0	1

3. 스테레오 이미징

가. 설명

스테레오 비전을 사용하면 이미지의 깊이를 인식하고 이미지에서 측정을 수행하며 3D 위치 파악을 수행할 수 있습니다. 이를 위해서는 무엇보다도 두 카메라 사이에서 일치하는 지점을 찾아야 합니다. 이를 통해 카메라와 지점 사이의 거리를 도출할 수 있습니다. 시스템의 기하학적 구조는 계산을 단순화하는 데 사용됩니다.

스테레오 이미징에서는 다음 네 단계가 수행됩니다.

1. 수학적 계산을 통해 방사형 및 접선 왜곡을 제거합니다. 이는 왜곡되지 않은 이미지를 제공합니다.

2. 이미지의 각도와 간격을 수정합니다. 이 단계에서는 두 이미지가 모두 Y축에서 동일 평면상에 있을 수 있으므로 해당 항목을 더 쉽게 검색할 수 있으며 단일 축(즉, X축)에서만 검색하면 됩니다.

3. 오른쪽과 왼쪽 이미지에서 동일한 특징을 찾으세요. 그러면 x축의 이미지 간 차이를 보여주는 시차 맵이 제공됩니다.
4. 마지막 단계는 삼각측량입니다. 삼각측량을 통해 시차 지도를 거리로 변환합니다.

1단계: 왜곡 제거 2단계: 수정

3단계: 두 이미지에서 동일한 특징 찾기 4단계: 삼각측량

B. 삼각 측량

마지막 단계인 삼각측량에서는 두 투영 이미지가 모두 동일 평면에 있고 왼쪽 이미지의 수평 픽셀 행이 왼쪽 이미지의 해당 수평 픽셀 행과 정렬되어 있다고 가정합니다.

이전 가설을 사용하여 이제 다음 그림을 만들 수 있습니다.

점 P는 환경에 있으며 해당 좌표 xl 및 xr을 사용하여 왼쪽 및 오른쪽 이미지의 pl 및 pr에 매핑됩니다. 이를 통해 우리는 새로운 양인 시차(d = xl - xr)를 도입할 수 있습니다. P점에서 멀어질수록 d의 크기는 작아지는 것을 알 수 있다. 따라서 차이는 거리에 반비례합니다.

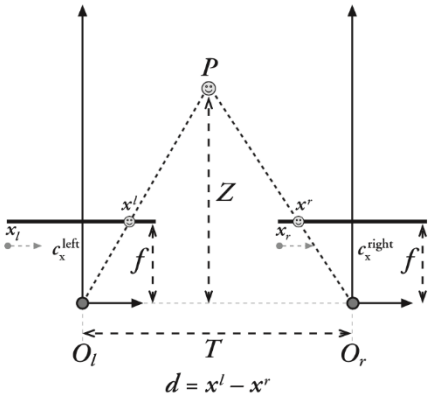


그림 10: 삼각 측량(출처: OpenCV 3 학습 - O'Reilly)

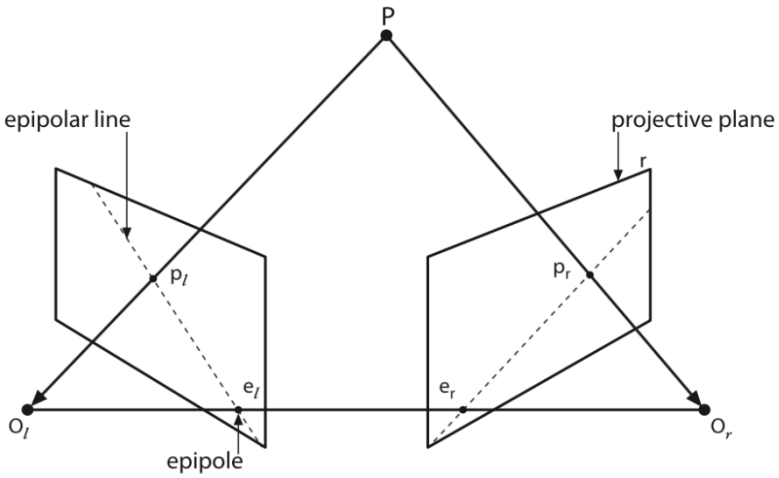
다음 공식을 사용하여 거리를 계산할 수 있습니다. Z=f*T/(xl-xr)

격차와 거리 사이에는 비선형 관계가 있음을 알 수 있습니다. 시차가 0에 가까우면 시차의 작은 차이가 거리의 큰 차이로 이어집니다. 격차가 크면 반대가 된다. 시차의 작은 차이가 거리의 큰 차이로 이어지지는 않습니다. 이것으로부터 우리는 스테레오 비전이 카메라에 가까운 물체에 대해서만 높은 깊이 해상도를 갖는다는 결론을 내릴 수 있습니다.

그러나 이 방법은 스테레오 카메라 구성이 이상적인 경우에만 작동합니다. 그러나 실제로는 그렇지 않습니다. 이것이 바로 왼쪽과 오른쪽 이미지가 수학적으로 평행하게 정렬된 이유입니다. 물론, 카메라는 적어도 대략 물리적으로 평행하게 위치해야 합니다.

이미지를 수학적으로 정렬하는 방법을 설명하기 전에 먼저 에피폴라 기하학을 이해해야 합니다.

C. 에피폴라 기하학



위 그림은 두 개의 핀홀 카메라 모델로 구성된 불완전한 스테레오 카메라 모델을 보여줍니다. 에피폴라 점(el 및 er)은 투영 중심(Ol, Or)의 선을 투영 평면과 교차함으로써 생성됩니다. 선 (pl, el)과 (pr, er)을 에피폴라 선이라고 합니다. 투영 평면에 있는 한 점의 가능한 모든 점의 이미지는 다른 이미지 평면에 있고 에피폴라 점과 원하는 점을 통과하는 에피폴라 선입니다. 이를 통해 점 검색을 전체 평면이 아닌 단일 차원으로 제한할 수 있습니다.

따라서 다음 사항을 요약할 수 있습니다.

- 카메라 뷰의 모든 3D 포인트가 에피폴라 계획에 포함됩니다.
- 한 평면의 형상은 다른 평면의 해당 에피폴라 선에 있어야 합니다(에피폴라 조건).
- 해당 특징의 2차원 검색은 에피폴라 기하학이 알려진 경우 1차원 검색으로 변환됩니다.
- 점의 순서는 유지됩니다. 즉, 두 점 A와 B가 한 평면의 에피폴라 선에서 다른 평면과 동일한 순서로 발견됩니다.

D. 필수 및 기본 매트릭스

에피폴라 선이 계산되는 방법을 이해하려면 먼저 필수 및 기본 행렬(행렬 E 및 F에 해당)을 설명해야 합니다.

필수 매트릭스 E에는 두 카메라가 물리적으로 어떻게 서로 배열되어 있는지에 대한 정보가 포함되어 있습니다. 변환 및 회전 매개변수를 사용하여 첫 번째 카메라를 기준으로 두 번째 카메라의 위치를 설명합니다. 이러한 매개변수는 프로젝트 계획에 사용되므로 매트릭스에서 직접 읽을 수 없습니다. 섹션에서*스테레오 교정* R과 T(회전 행렬과 이동 벡터)를 계산하는 방법을 설명합니다. 행렬 F에는 카메라의 물리적 배열에 대한 필수 행렬 E의 정보와 카메라의 고유 매개변수에 대한 정보가 포함되어 있습니다.

왼쪽 이미지 pl의 투영점과 오른쪽 이미지 pr의 투영점 사이의 관계는 다음과 같이 정의됩니다.

$$피_{트랜스포즈}에프엘=0$$

이 공식이 왼쪽 점과 오른쪽 점 사이의 관계를 완전히 설명한다고 생각할 수도 있습니다. 그러나 3x3 행렬 E는 랭크 2라는 점에 유의하는 것이 중요합니다. 이는 이 공식이 직선의 방정식임을 의미합니다.

따라서 점 간의 관계를 완전히 정의하려면 고유 매개변수를 고려해야 합니다.

우리는 고유 행렬 M을 사용하여 q=Mp임을 기억합니다.
이전 방정식을 대체하면 다음이 제공됩니다.

$$큐_{트랜스포즈}(엠^{-1})^{트}EM_{엘}-큐_{엘}=0$$

당신은 다음을 대체합니다:

$$F=(엠^{-1})^{트}여자\ 이름_{엘}$$

그래서 당신은 다음을 얻습니다:

$$큐_{트랜스포즈}Fq_{엘}=0$$

E. 회전 행렬 및 이동 벡터

이제 필수 행렬 E와 기본 행렬 F에 대해 설명했으므로 회전 행렬과 병진 벡터를 계산하는 방법을 살펴보겠습니다.

우리는 다음 표기법을 정의합니다:

- $피_{엘}$그리고$피_{오른쪽\ 자형}$왼쪽 및 오른쪽 카메라의 좌표계에서 점의 위치를 정의합니다.

- $오른쪽\ 자형_{엘}$그리고$티_{엘}$(또는$오른쪽\ 자형_{오른쪽\ 자형}$그리고$티_{오른쪽\ 자형}$) 카메라에서 왼쪽(또는 오른쪽) 카메라 환경 내 지점까지의 회전 및 변환을 정의합니다.
- R과 T는 오른쪽 카메라의 좌표계를 왼쪽 카메라의 좌표계로 가져오는 회전 및 변환입니다.

그 결과는 다음과 같습니다.

$$피_{엘}=R_{엘}피+티_{엘}그리고\ 피_{오른쪽\ 자형}=R_{오른쪽\ 자형}피+티_{오른쪽\ 자형}$$

당신은 또한 다음을 가지고 있습니다 :

$$피_{엘}=RT(피_{오른쪽\ 자형}-티)$$

이 세 가지 방정식을 사용하여 최종적으로 다음을 얻습니다.

$$R=R_{오른쪽\ 자형}오른쪽\ 자형_{엘}티$$

$$티=티_{오른쪽\ 자형}-RT_{엘}$$

F. 스테레오 정류

지금까지 우리는 "스테레오 교정"이라는 주제를 다루었습니다. 두 카메라의 기하학적 배치에 대한 설명이었습니다. 수정 작업은 두 이미지가 정확히 동일한 평면에 있도록 투사하는 것입니다.

두 이미지에서 점의 대응을 더 무작위로 찾을 수 있도록 에피폴라 선이 수평이 되도록 픽셀 행을 정확하게 배치하고 정렬합니다.
두 이미지를 정렬하는 과정의 결과로 각 카메라에 대해 4개, 8개의 용어가 얻어졌습니다.

- 왜곡 벡터
- 회전 행렬*아르 자형*직사각형, 이미지에 적용해야 함
- 수정된 카메라 매트릭스*중*직사각형
- 수정되지 않은 카메라 매트릭스*중*

OpenCV를 사용하면 Hartley 알고리즘과 Bouguet 알고리즘이라는 두 가지 알고리즘을 사용하여 이러한 용어를 계산할 수 있습니다.

1. 하틀리 알고리즘

Hartley 알고리즘은 두 이미지에서 동일한 점을 찾습니다. 그는 불균형을 최소화하고 에피폴을 무한대로 설정하는 동형어를 찾으려고 노력합니다. 이 방법을 사용하면 각 카메라의 고유 매개변수를 계산할 필요가 없습니다.

이 방법의 장점은 장면의 점을 관찰해야만 보정이 가능하다는 것입니다. 그러나 큰 단점은 이미지의 크기 조정이 없고 상대적인 거리에 대한 정보만 알 수 있다는 것입니다. 물체가 카메라로부터 얼마나 멀리 떨어져 있는지 정확하게 측정할 수는 없습니다.

2. 부게 알고리즘

Bouguet 알고리즘은 계산된 회전 행렬과 변환 벡터를 사용하여 투영된 두 평면을 모두 반 바퀴 회전하여 동일한 평면에 있도록 합니다. 이는 주 광선이 평행하게 정렬되고 평면이 동일 평면에 정렬되지만 행으로 정렬되지 않음을 의미합니다. 이 작업은 나중에 수행됩니다.

이 프로젝트에서는 Bouguet 알고리즘을 사용했습니다.

4.프로그램 작동 방식 스테레오 이미지

이미 언급했듯이 프로그램은 Python으로 코딩되었으며 OpenCV 라이브러리가 사용됩니다. 우리는 Python 언어와 OpenCV 라이브러리를 선택했습니다. 이미 경험이 있고 이에 대한 문서가 많기 때문입니다. 이 결정에 대한 또 다른 주장은 우리가 "오픈 소스" 라이브러리로만 작업하기를 원했다는 것입니다.

이 프로젝트를 위해 두 개의 Python 프로그램이 개발되었습니다.

첫 번째 "**Take_images_for_calibration.py**"는 나중에 두 카메라의 보정(왜곡 보정 및 스테레오 보정)에 사용될 좋은 이미지를 캡처하는 데 사용됩니다.

두 번째 프로그램이므로 메인 프로그램입니다.”**Main_Stereo_Vision_Prog.py**"는 스테레오 이미징에 사용됩니다. 이 프로그램에서는 다음을 사용하여 카메라를 교정합니다.

촬영된 이미지는 시차 지도를 생성하고 실험적으로 발견된 직선 방정식 덕분에 각 픽셀의 거리를 측정할 수 있습니다. 마지막에는 물체의 가장자리를 더 잘 인식하기 위해 WLS 필터가 사용됩니다.

Python 프로그램은 부록에서 찾을 수 있습니다.

A. 사용된 패키지

다음 패키지를 프로그램으로 가져왔습니다.

- Python에서 "cv2"라고 불리는 opencv_contrib(스테레오 함수 포함)이 포함된 OpenCV.3.2.0 버전에는 다음이 포함되어 있습니다.
  영형이미지 처리용 라이브러리 스테레오 비
  영형전 기능
- 넘피.1.12.
  영형행렬 연산에 사용됩니다. (이미지는 행렬로 구성됩니다.)
- openpyxl의 통합 문서
  영형Excel 파일에 데이터를 쓸 수 있는 패키지
- sklearn 0.18.1 라이브러리의 "정규화"
  영형sklearn은 기계 학습을 지원하지만 이 프로젝트에서는 WLS 필터만 사용됩니다.

```
# Package importation
import numpy as np
import cv2
from openpyxl import Workbook # Used for writing data into an Excel file
from sklearn.preprocessing import normalize
```

코드 1: 패키지 가져오기

B. 비디오 루프

카메라를 사용하려면 먼저 활성화해야 합니다. 기능 *cv2.VideoCapture()*각 카메라의 포트 수를 입력하여 두 카메라를 모두 활성화합니다(프로그램에서 두 개의 개체가 생성되고 클래스의 메서드 *cv2.VideoCapture()*사용할 수 있습니다).

```
# Call the two cameras
CamR= cv2.VideoCapture(0)
CamL= cv2.VideoCapture(2)
```

코드 2: OpenCV를 사용한 카메라 활성화

카메라에서 사진을 얻으려면, 방법*cv2.VideoCapture().read()* 사용하면 이 함수가 호출되는 순간 카메라가 보고 있는 장면의 이미지를 출력으로 얻을 수 있습니다. 비디오를 얻으려면 무한 루프에서 이 메서드를 계속해서 호출해야 합니다. 보다 효율적으로 BGR 이미지를 회색 이미지로 변환하는 것이 좋습니다. 이는 다음 기능을 사용하여 수행됩니다.*cv2.cvtColor()* 실행.

```
while True:
    retR, frameR= CamR.read()
    retL, frameL= CamL.read()
    grayR= cv2.cvtColor(frameR,cv2.COLOR_BGR2GRAY)
    grayL= cv2.cvtColor(frameL,cv2.COLOR_BGR2GRAY)
```

코드 3: 이미지 처리

PC에서 영상을 보기 위해 해당 기능을 사용합니다.*cv2.imshow()*이를 사용하면 비디오를 표시할 수 있는 창을 열 수 있습니다.

```
    cv2.imshow('VideoR',grayR)
    cv2.imshow('VideoL',grayL)
```

코드 4: 이미지 표시

무한 루프에서 빠져나오기 위해서는 브레이크(Break)를 사용합니다. 사용자가 스페이스바를 누를 때마다 활성화됩니다. 키를 눌렀다는 인식은 기능 덕분입니다.*cv2.waitKey()*수행.

마지막에는 다음 방법을 사용하여 사용된 두 대의 카메라를 비활성화해야 합니다.
*cv2.VideoCapture().release()*열린 창에는 해당 기능이 포함됩니다. *cv2.destroyAllWindows()*닫은.

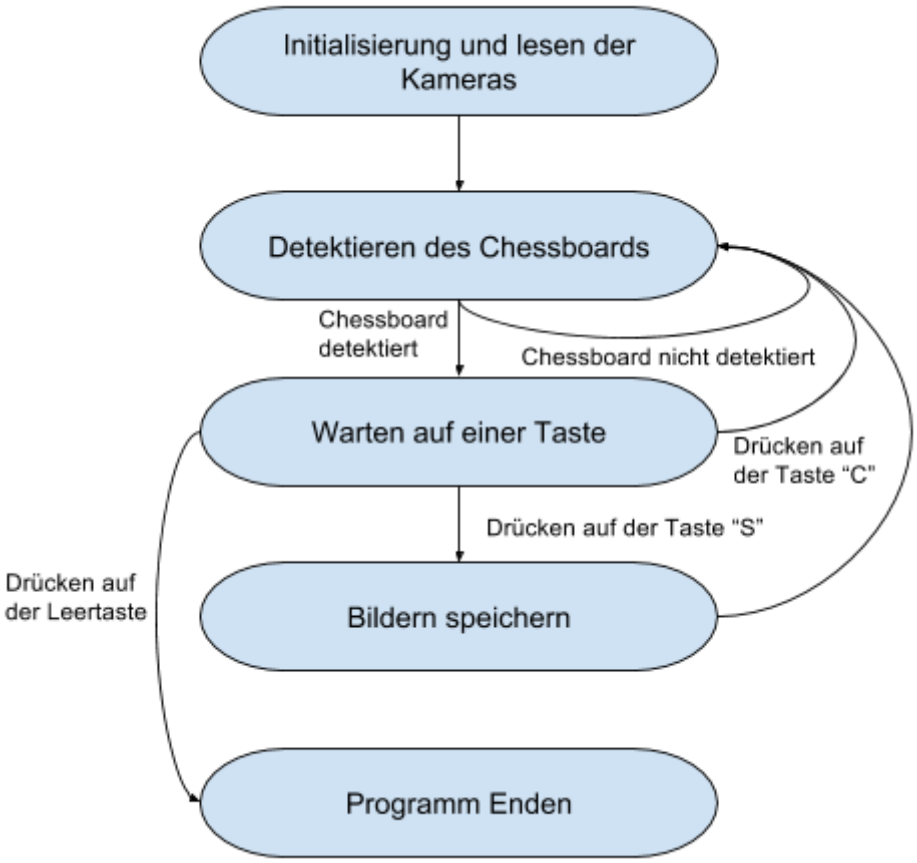
```
    # End the Programme
    if cv2.waitKey(1) & 0xFF == ord(' '):
        break

# Release the Cameras
CamR.release()
CamL.release()
cv2.destroyAllWindows()
```

코드 5: 일반적인 프로그램 종료

C. 프로그램 작동 방식 “ Take_images_for_calibration.py”

이 프로그램이 시작되면 두 카메라가 모두 활성화되고 사용자가 이미지에서 체스판의 위치를 볼 수 있도록 두 개의 창이 열립니다.



*다이어그램 1: "작동 방식"**Take_images_for_calibration.py"***

1. 교정용 벡터

기능*cv2.findChessboardCorners()*정의된 수의 체스판 모서리를 검색하고 다음 벡터가 생성됩니다.

- **imgpointsR**: 오른쪽 이미지(이미지 공간)의 모서리 좌표를 포함합니다.
- **imgpointsL**: 왼쪽 이미지(이미지 공간)의 모서리 좌표를 포함합니다.
- **객체 포인트**: 객체 공간의 모서리 좌표를 포함합니다.

발견된 모서리 좌표의 정밀도가 함수에서 증가합니다. *cv2.cornerSubPix()*사용됩니다.

2. 교정을 위한 이미지 획득

프로그램이 두 이미지 모두에서 체스판 모서리 위치를 인식하면 캡처한 이미지를 평가할 수 있는 두 개의 새 창이 열립니다. 이미지가 흐릿하지 않고 보기에도 좋으면 "s"(저장) 버튼을 눌러 이미지를 저장해야 합니다. 반대의 경우에는 "c"(취소) 키를 누를 수 있습니다.

누르다 그것으로 그만큼 영화 ~아니다 저장됨 이 되다. 와 함께 그만큼 기능 *cv2.draw체스보드코너()*프로그램은 이미지에 체커보드 패턴을 오버레이합니다.

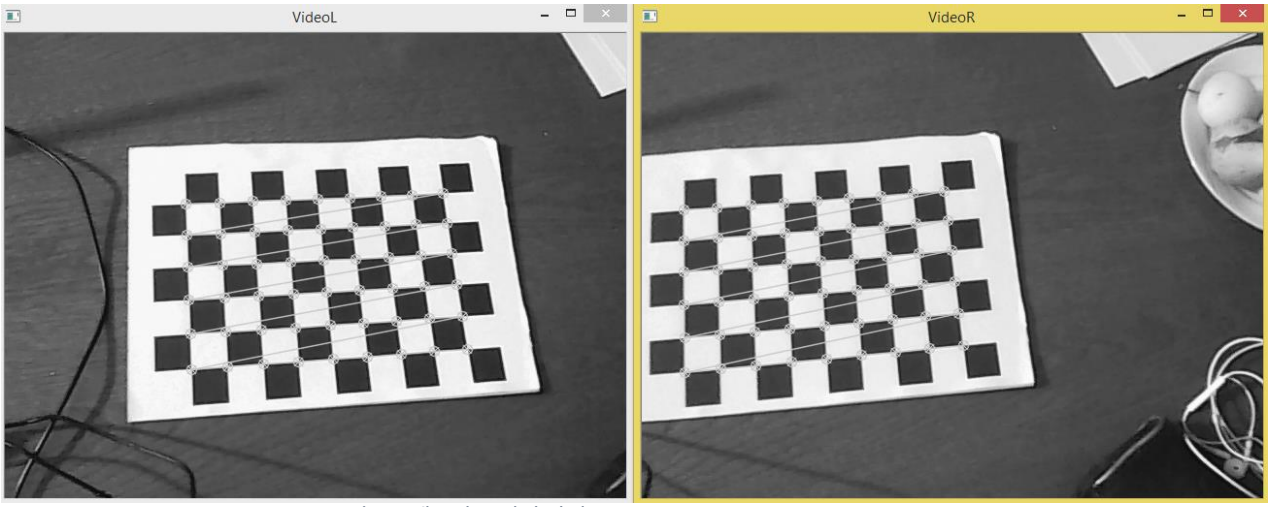


그림 11: 체스판 모서리 감지

D. 프로그램 작동 방식
Main_Stereo_Vision_Prog.py"

초기화하는 동안 카메라는 먼저 개별적으로 보정되어 왜곡을 제거합니다. 그런 다음 스테레오 교정이 수행됩니다(회전 제거, 에피폴라 선 정렬). 이미지는 무한 루프로 처리되고 시차 맵이 생성됩니다.

다이어그램 2: "Main_Stereo_Vision_Prog.py" 작동 방식

1. 왜곡 교정

카메라의 왜곡을 보정하기 위해 "Take_images_for_calibration.py" 프로그램으로 촬영한 이미지가 사용됩니다.
이 보정은 "Take_images_for_calibration.py"를 기반으로 합니다.**이미지포인트**그리고 **객체점**체스판 모서리의 위치가 저장됩니다. 기능 *cv2.calibrateCamera()*새로운 카메라 매트릭스(카메라 매트릭스는 2D 이미지에서 3D 세계의 한 점 투영을 설명함), 왜곡 계수, 각 개별 카메라에 대한 회전 및 변환 벡터를 얻는 데 사용되며, 이는 나중에 각 카메라의 왜곡을 제거하는 데 필요합니다. . 이 기능은 각 카메라에 대한 최적의 카메라 매트릭스를 얻는 데 사용됩니다. *cv2.getOptimalNewCameraMatrix()*사용됩니다(정밀도 증가).

```
#    Right Side
retR, mtxR, distR, rvecsR, tvecsR = cv2.calibrateCamera(objpoints,
                                                        imgpointsR,
                                                        ChessImaR.shape[::-1],None,None)
hR,wR= ChessImaR.shape[:2]
OmtxR, roiR= cv2.getOptimalNewCameraMatrix(mtxR,distR,
                                           (wR,hR),1,(wR,hR))
```

코드 6: Python의 왜곡 교정

프레데릭 우르바일러
스테판 부자시노비치

2. 스테레오 카메라 교정

이 기능은 스테레오 교정에 사용됩니다.*cv2.StereoCalibrate()*이를 사용하면 두 카메라 간의 변환을 계산합니다(한 카메라는 다른 카메라의 참조 역할을 합니다).

기능*cv2.stereoRectify()*두 카메라의 에피폴라 라인을 동일한 평면에 가져올 수 있습니다. 이 변환은 블록 일치를 한 차원에서만 검색하면 되기 때문에 시차 맵을 생성하는 함수의 작업을 더 쉽게 만듭니다. 이 기능을 사용하면 다음 기능에 필요한 필수 매트릭스와 기본 매트릭스도 얻을 수 있습니다.

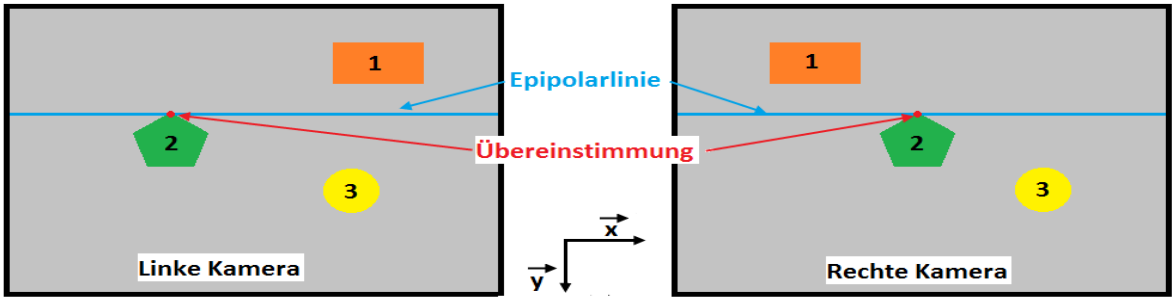


그림 12: 에피폴라 선의 원리

기능*cv2.initUndistortRectifyMap() 어*왜곡이 없는 이미지를 제공합니다. 그런 다음 이러한 이미지는 시차 맵 계산에 사용됩니다.

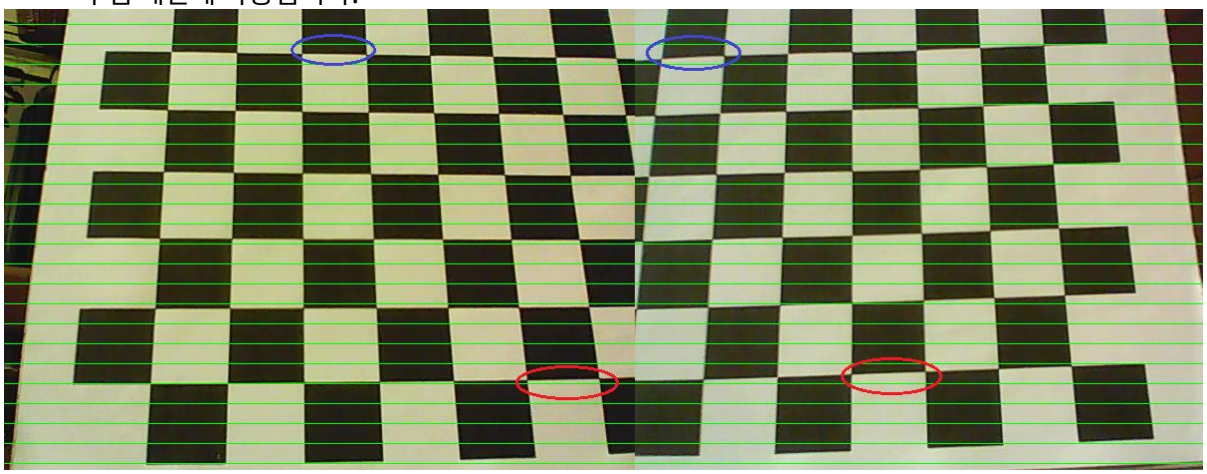


그림 13: 교정하지 않음

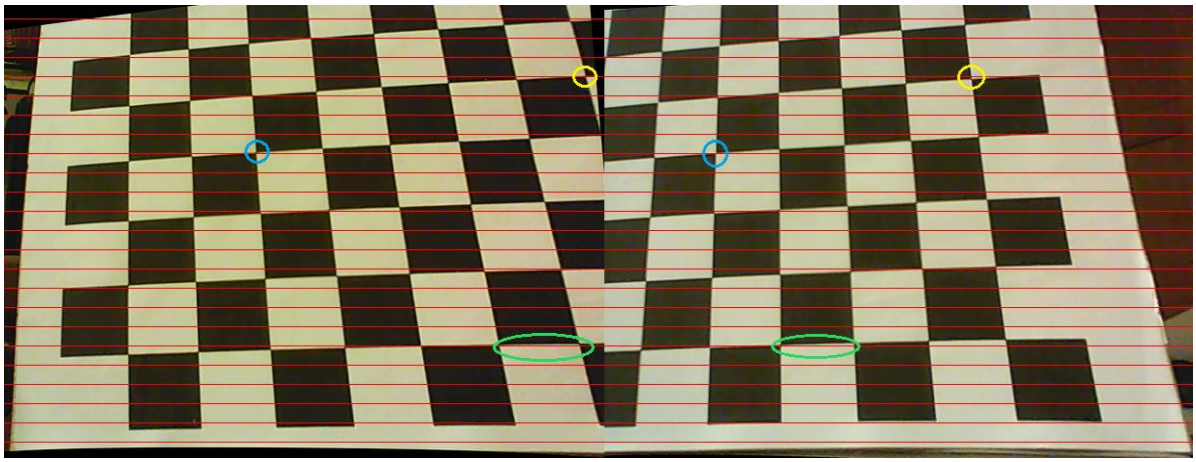


그림 14: 교정 포함

3. 시차 지도 계산

시차 맵을 계산하기 위해 다음 함수를 사용하여 StereoSGBM 개체가 생성됩니다.
*cv2.StereoSGBM_create().*이 클래스는 반전역 매칭 알고리즘(Hirschmüller, 2008)을 사용하여 오른쪽 카메라와 왼쪽 카메라의 이미지 간의 스테레오 매칭을 얻습니다.

반전역 일치 알고리즘의 작동 방식:

다음 장면이 스테레오 카메라에 표시됩니다.

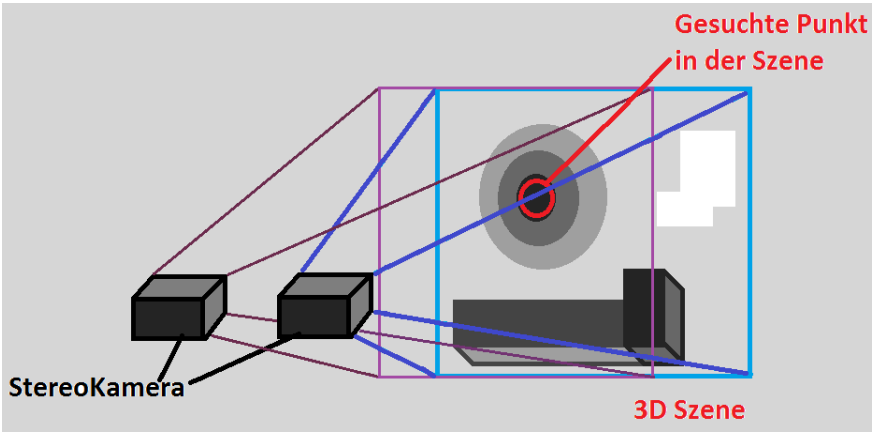


그림 15: 장면을 관찰하는 스테레오 카메라의 예

블록의 크기는 입력 매개변수에 정의됩니다. 블록 크기가 1보다 큰 경우 이러한 블록은 픽셀을 대체합니다. 생성된 SGBM 객체는 참조 이미지의 블록을 일치 이미지의 블록과 비교합니다. 예를 들어, 스테레오 보정이 잘 수행된 경우 참조 이미지의 4행에 있는 블록은 4행에만 있는 일치 이미지의 모든 블록과 비교되어야 합니다. 이러한 방식으로 시차 맵의 계산이 더욱 효율적이 됩니다.

네 번째 행 블록에 대한 이전 예를 사용하여 시차 맵이 작성되는 방법을 설명하겠습니다.

아래 그림에서는 기본 이미지의 네 번째 행, 일곱 번째 열의 블록(4,7)을 일치 이미지의 네 번째 행(동일한 에피폴라 선)의 다른 모든 블록(4,i)과 비교해야 합니다.

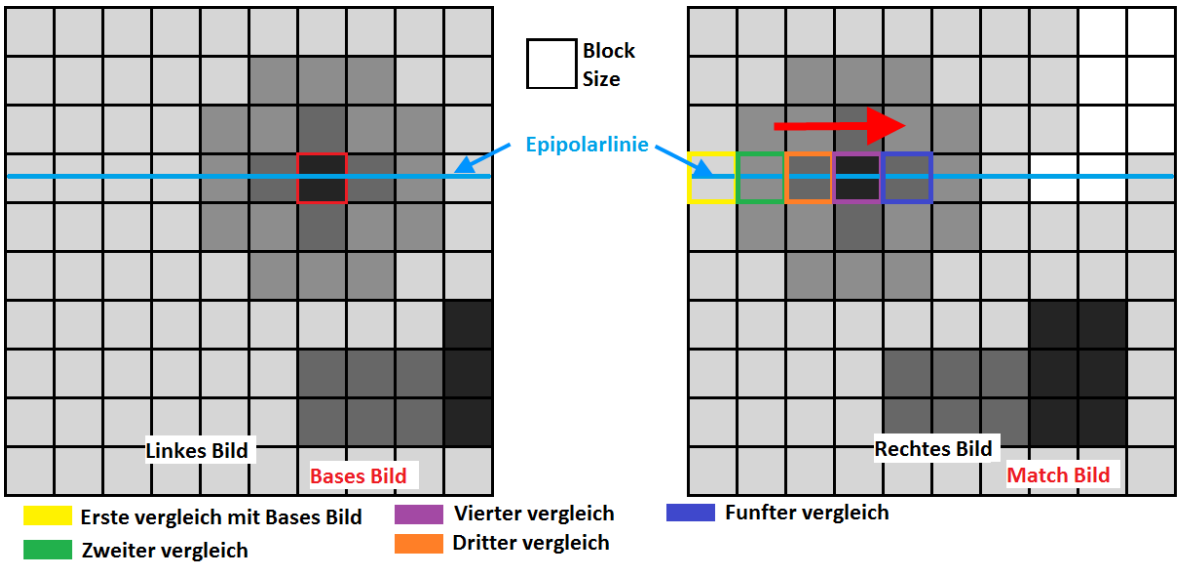


그림 16: StereoSGBM을 사용하여 블록 일치 찾기

참조 블록과 일치 블록 간의 일치가 클수록 환경에서 동일한 지점일 가능성이 높아집니다.

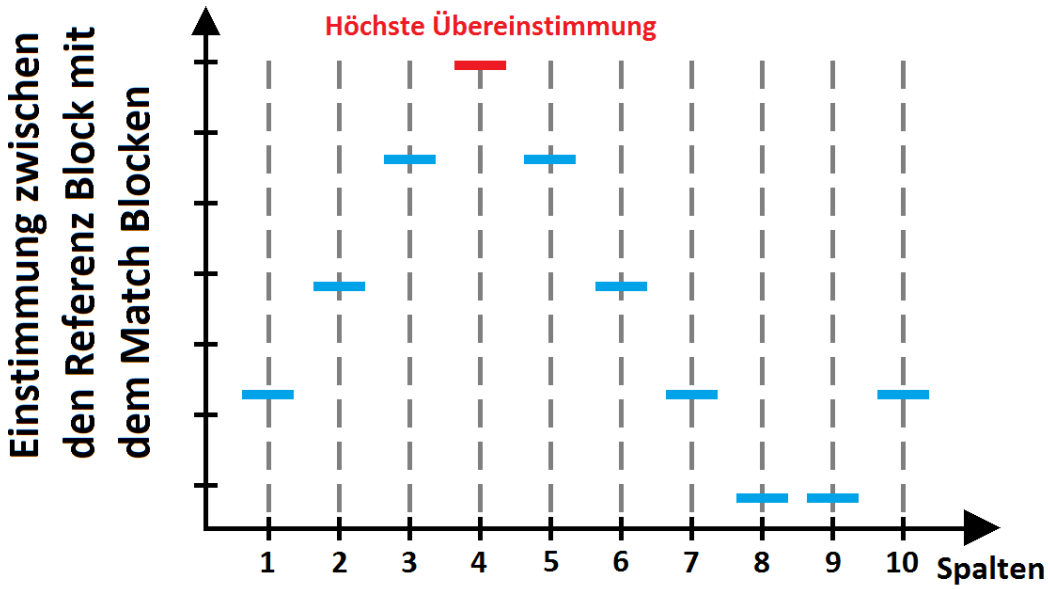


그림 17: StereoSGBM을 사용한 동일한 블록 감지

이 예에서는 참조 블록(4,7)이 일치 블록(4,4)과 일치도가 가장 높은 것을 볼 수 있습니다.

이론적으로는 이와 같이 작동해야 하지만 실제로는 OpenCV에서 기본적으로 한 방향과 동일한 방법을 사용하여 4개의 다른 방향이 더 정확하게 처리됩니다.

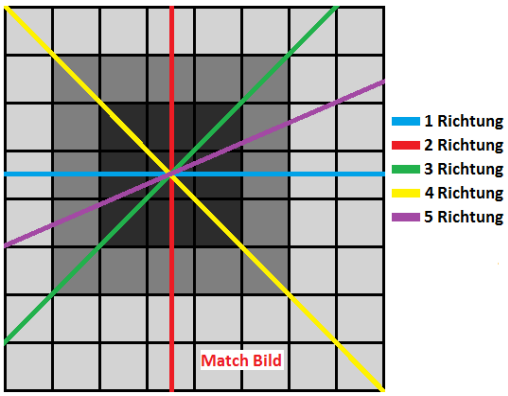


그림 18: OpenCV의 StereoSGBM 알고리즘의 5개 방향 예

시차를 찾기 위해 일치 블록의 좌표에서 참조 블록을 뺀 다음 결과에서 절대 값을 가져오고 이 값이 클수록 객체가 스테레오 카메라에 더 가까워집니다.

프로그램은 보정된 흑백 이미지를 사용하여 시차 맵을 계산합니다. BGR 이미지로 작업하는 것도 가능하지만 이 경우 컴퓨터에 더 많은 시간이 필요합니다. 카드는 스테레오 생성 객체인 cv2.StereoSGBM_create().compute()의 메서드를 사용하여 계산됩니다.

```
# Show the result for the Disparity Map
disp= ((disp.astype(np.float32)/ 16)-min_disp)/num_disp
cv2.imshow('disparity', disp)
```

코드 7: 시차 지도 표시

초기화에서 설정된 매개변수를 사용하여 시차 맵에 대해 다음과 같은 결과를 얻습니다.

```
# Create StereoSGBM and prepare all parameters
window_size = 3
min_disp = 2
num_disp = 130-min_disp
stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
    numDisparities = num_disp,
    blockSize = window_size,
    uniquenessRatio = 10,
    speckleWindowSize = 100,
    speckleRange = 32,
    disp12MaxDiff = 5,
    P1 = 8*3*window_size**2,
    P2 = 32*3*window_size**2)
```

코드 8: StereoSGBM 인스턴스에 대한 매개변수

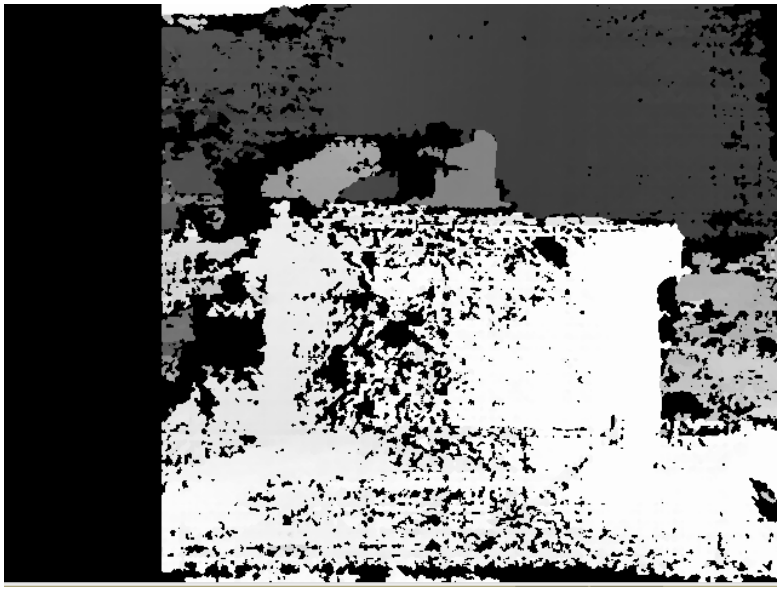


그림 19: 시차 지도 결과

이 시차 맵에는 여전히 많은 노이즈가 있으므로 이를 제거하기 위해 형태학적 필터가 사용됩니다. OpenCV 기능에는 "닫는" 필터가 사용됩니다.*cv2.morphologyEx(cv2.MORPH_CLOSE)*작은 검은 점들을 제거하려면

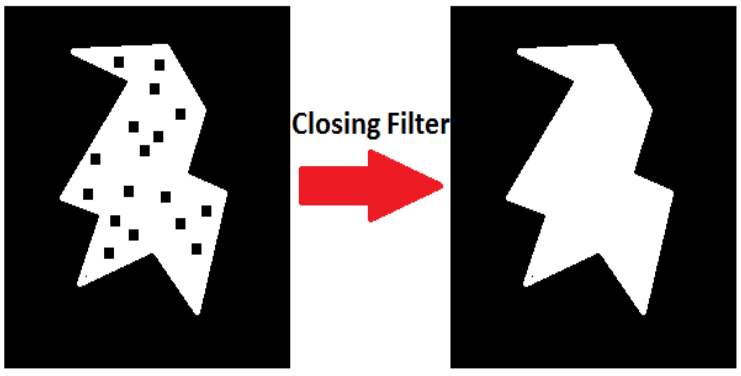


그림 20: 닫는 필터의 예

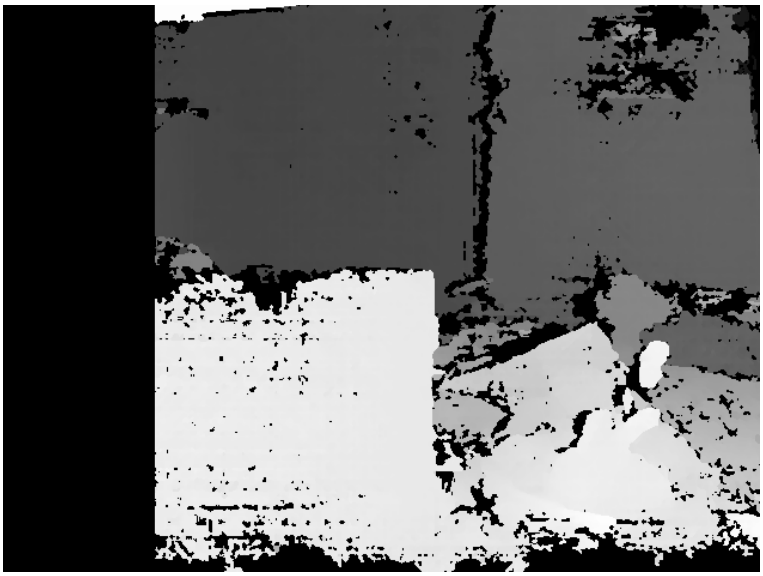


그림 21: 필터 닫기 후 시차 맵 결과

차이점을 더 잘 볼 수 있도록 동일한 장면을 사용한 또 다른 예입니다.

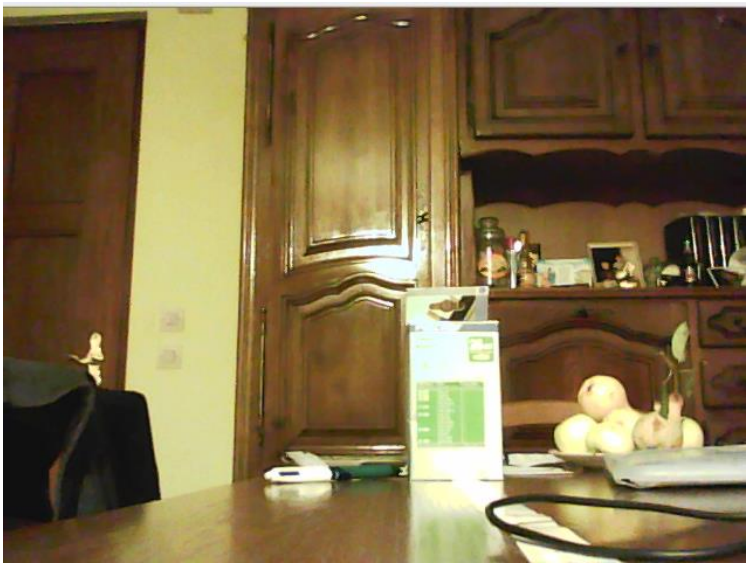


그림 22: 수정 및 보정 없이 왼쪽 카메라에서 본 일반 장면

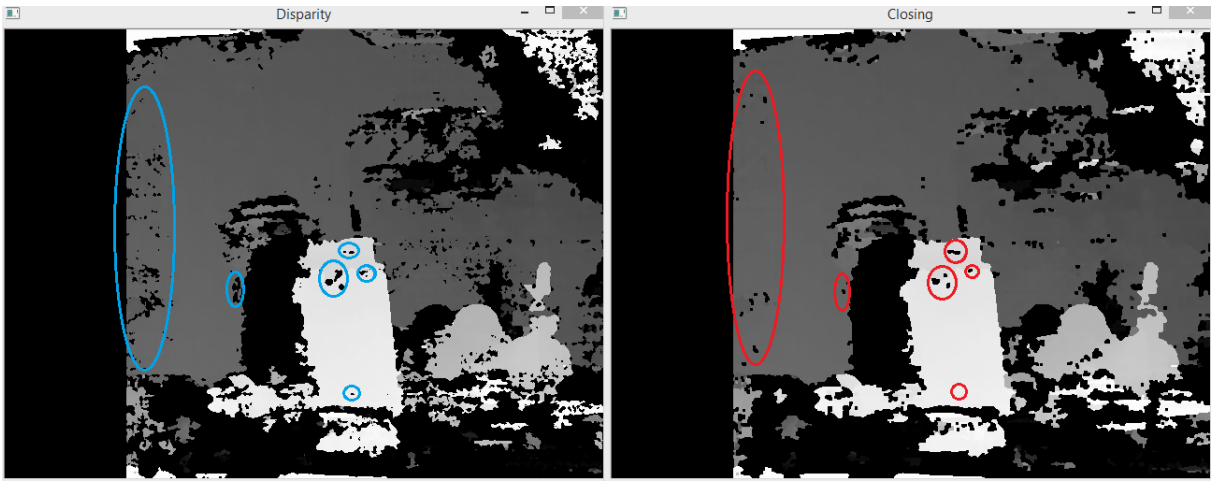


그림 23: 필터를 닫지 않고 필터를 적용한 상단 장면의 시차 맵

4. WLS(Weighted Least Squares) 필터 사용

결과는 나쁘지 않지만 여전히 노이즈로 인해 물체의 가장자리를 인식하기가 매우 어렵기 때문에 WLS 필터를 사용합니다. 먼저 초기화 시 필터의 매개변수를 설정해야 합니다.

```
# FILTER Parameters for the WLS filter
lmbda = 80000
sigma = 1.8
visual_multiplier = 1.0
```

코드 9: WLS 필터의 매개변수

Lambda는 일반적으로 8000으로 설정되는데, 이 값이 클수록 시차 지도의 모양이 참조 이미지의 모양에 더 많이 부착될수록 더 좋은 결과가 나오므로 80000으로 설정했습니다. Sigma는 필터가 물체의 가장자리에서 얼마나 정확해야 하는지 설명합니다.

또 다른 스테레오 객체는 다음을 사용하여 생성됩니다.*cv2.ximgproc.createRightMatcher()*첫 번째를 기반으로합니다. 이 두 인스턴스는 WLS 필터에서 시차 맵을 생성하는 데 사용됩니다.

필터 인스턴스는 다음 함수를 사용하여 생성됩니다.*cv2.ximgproc.createDisparityWLSFilter()* 제조.

```
# Used for the filtered image
stereoR=cv2.ximgproc.createRightMatcher(stereo) # Create another stereo

# Create the WLS filter
wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=stereo)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)
```

코드 10: Python에서 WLS 필터 생성

그런 다음 WLS 필터의 인스턴스를 적용하기 위해 다음 메서드가 호출됩니다. *cv2.ximgproc.createRightMatcher().filter()*, 필터의 값은 다음과 같이 정규화됩니다.*cv2.normalize()* .

```
# Using the WLS filter
filteredImg= wls_filter.filter(displ,grayL,None,dispR)
filteredImg = cv2.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255,
filteredImg = np.uint8(filteredImg)
```

코드 11: Python에서 WLS 필터 구현

더 나은 시각화를 위해 Ocean ColorMap이 사용되었습니다. *cv2.applyColorMap()*. 색상이 어두울수록 물체가 스테레오 카메라에 더 멀리 떨어져 있는 것입니다.

그림 24: 해양 색상맵

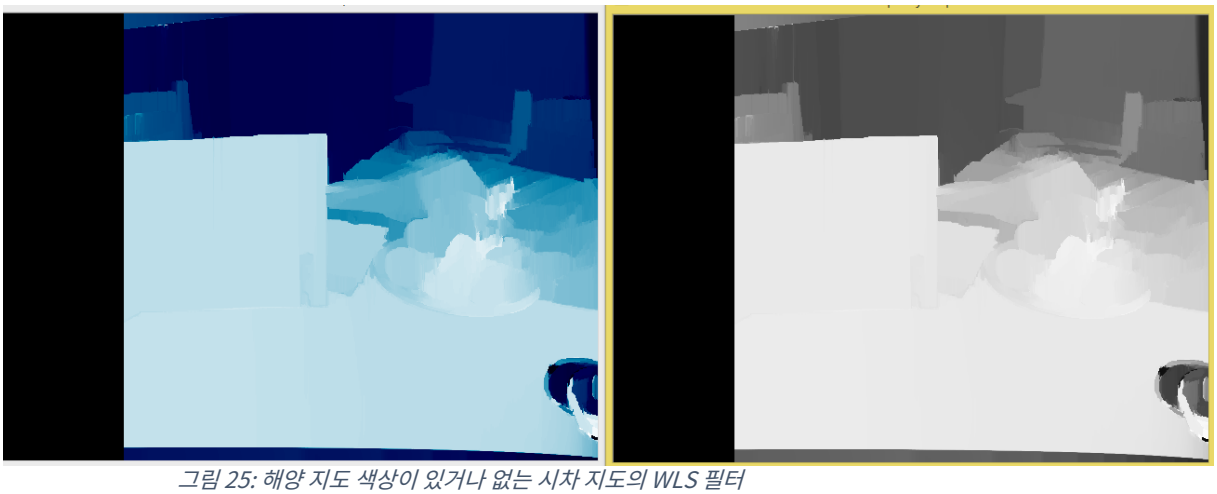


그림 25: 해양 지도 색상이 있거나 없는 시차 지도의 WLS 필터

이러한 방식으로 우리는 가장자리를 잘 표시할 수 있지만 더 이상 충분히 정확하지 않고 더 이상 좋은 시차 값을 포함하지 않는 이미지를 얻습니다(시차 맵은 float32로 인코딩되지만 WLS 결과는 uint8로 인코딩됩니다).
좋은 값을 사용하여 나중에 물체까지의 거리를 측정하기 위해 더블 클릭으로 WLS 필터링된 이미지의 x, y 좌표를 가져옵니다. 이러한 (x, y) 좌표는 시차 맵에서 시차 값을 가져온 다음 거리를 측정하는 데 사용됩니다. 반환된 값은 9x9 픽셀 매트릭스의 불일치 평균입니다.

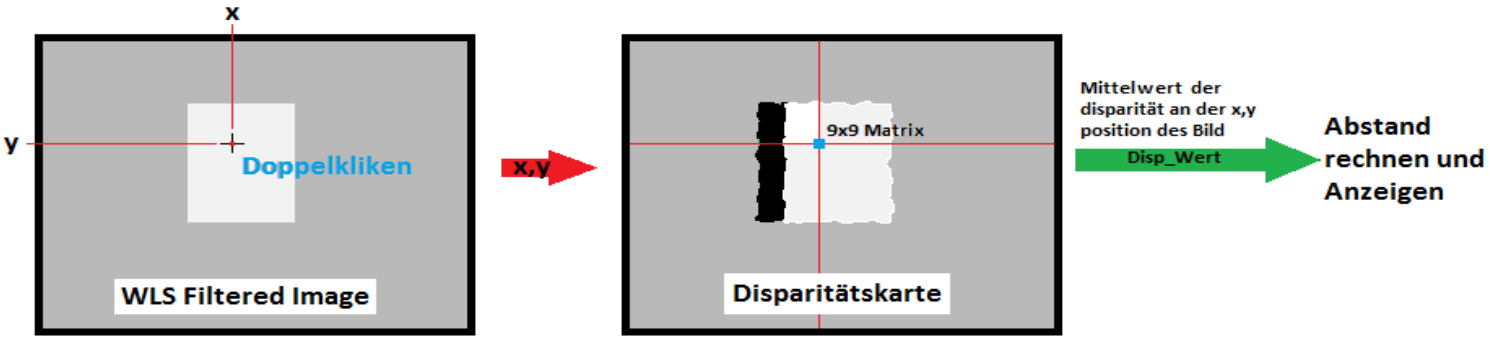
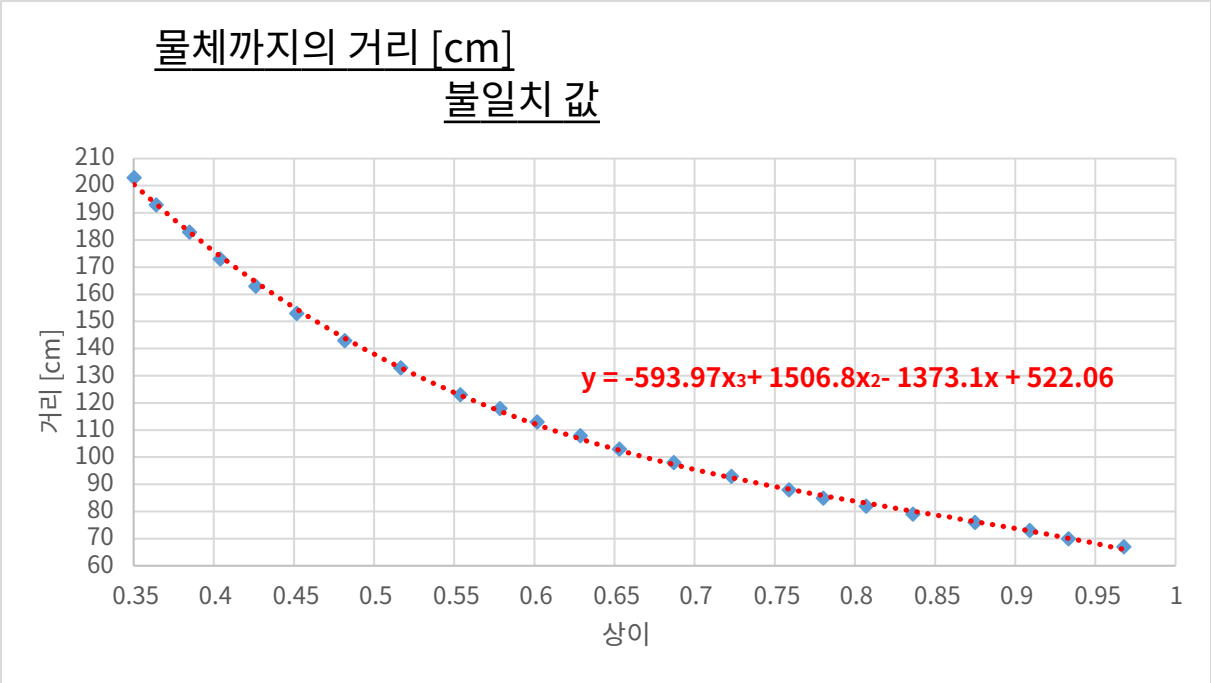


그림 26: WLS 필터와 시차 맵을 사용하여 거리 계산

5. 거리 측정

시차 지도가 생성된 후에는 거리를 결정해야 합니다. 작업은 시차 값과 거리 사이의 관계를 찾는 것으로 구성됩니다. 이를 위해 여러 지점에서 차이 값을 실험적으로 측정하여 회귀를 결정했습니다.



Python 프로그램의 선 방정식

```
# Equation for the distance mesurements
Distance= -593.97*average**(3) + 1506.8*average**(2) - 1373.1*average + 522.06
```

*코드 12: "의 회귀 공식**Main_Stereo_Vision_Prog.py**"*

회귀의 선형 방정식을 얻기 위해 패킹 위치가 사용되었습니다.*openpyxl* Excel 파일에 시차 값을 저장하는 데 사용됩니다. 프로그램에서 값을 저장하게 만든 행은 프로그램에서 주석 처리되었지만 새로운 행 방정식이 필요한 경우 주석 처리를 해제할 수 있습니다.

거리 측정은 좋은 결과를 얻기 위해 67cm~203cm의 거리에만 적용됩니다. 측정의 정밀도는 교정 품질에 따라 달라집니다. 우리의 스테레오 카메라는 +/5 cm의 정밀도로 물체까지의 거리를 측정할 수 있었습니다.

그림 27: 물체와의 거리에 따른 시차의 실험적 측정

6. 개선사항

프로그램의 가능한 개선 사항:

- WLS 필터의 모양을 가져와 시차 지도에 투영합니다. 그런 다음 이 투영을 통해 모양에 있는 모든 불일치 값을 취하고 가장 자주 발생하는 값을 전체 병의 값으로 설정합니다.

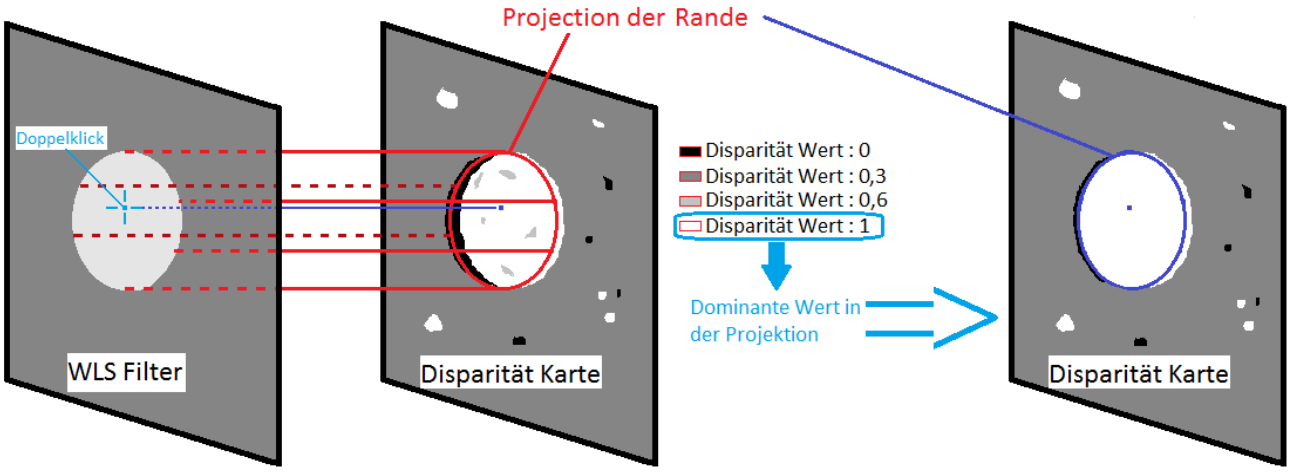


그림 28: 첫 번째 제안으로 가능한 개선 사항

- 시차 맵을 생성하기 전에 보정된 이미지에 양방향 필터를 사용하면 WLS 필터를 사용하지 않는 것이 가능합니다. 이는 확인이 필요하지만 WLS 필터는 물체의 가장자리를 명확하게 인식하는 데에만 사용됩니다. 아마도 더 좋은 방법이 있을 것입니다.

- 시차 맵 생성을 위한 계산 시간을 줄이려면 다음 기능을 사용하여 보정된 이미지의 크기를 줄여야 합니다.*cv2.크기 조정(cv2:INTER_AREA)*, 그런 다음 필수 행렬과 기본 행렬의 값도 비례적으로 감소하는지 확인해야 합니다.

- 깊이 맵을 만드는 것도 도움이 될 수 있습니다.

- 머리의 회전을 실제로 방지할 수 있는 우리보다 더 안정적인 카메라에서는 이러한 방식으로 다시 사용할 수 있도록 스테레오 보정에서 행렬 값을 저장하기만 하면 됩니다. 이렇게 하면 초기화하는 동안 많은 시간을 절약할 수 있습니다.

- GPU에서 프로그램을 실행하면 스테레오 카메라가 움직일 때 더 부드러운 이미지를 얻을 수도 있습니다.

5. 결론

가. 요약

이 프로젝트 작업을 통해 우리는 Python 언어로 작업하고 OpenCV 라이브러리에 대해 더 많이 배울 수 있었습니다. 이번 프로젝트에서는 스테레오 비전이라는 주제로 서로를 신뢰하는 기회를 가졌습니다. 우리 모두에게 큰 관심을 끄는 새로운 주제이자 사용 가능한 다른 거리 측정 기술에 비해 여전히 매우 새로운 주제입니다. TOF(Time of Light) 카메라와 같이 이미지 처리를 사용하여 거리를 측정하는 다른 방법도 있지만 비용이 많이 들고 문서가 거의 없습니다. 우리도 가격 때문에 간단한 카메라를 사용하기로 결정했습니다.

나. 결론

여기서 수행된 프로젝트 작업에서는 개발된 프로그램을 사용하여 시차 지도를 기반으로 거리를 계산하는 것이 가능합니다.

다. 전망

계산된 거리 값이 항상 정확하게 유지되도록 하려면 카메라의 자유로운 움직임을 방지하는 카메라용 새 시스템을 개발해야 합니다. 이는 교정을 더 빠르게 수행하기 위해 매트릭스 값만 사용된다는 것을 의미합니다. 사용된 선 방정식은 항상 더 정확하고 더 적은 노력으로 물체까지의 정확한 거리를 반환합니다.

서지

OpenCV Python 튜토리얼

OpenCV 문서

스택 오버플로

웹사이트 rdmilligan.wordpress.com의 RDMILLIGAN

스테레오 비전: 알고리즘 및 응용 - Stefano Mattoccia, 볼로냐 대학교 컴퓨터 공학부(DISI)

Nassir Navab의 스테레오 매칭과 Christian Unger가 준비한 슬라이드

Oreilly 학습 OpenCV

6. 부록

우리 프로젝트에 대한 비디오:https://youtu.be/xjx4mbZXaNc
Python 프로그램은 "폴더에서 찾을 수 있습니다.**Python_Prog_Stereo_Vision**"를 찾을 수 있습니다.
프로그램으로 구성되어 있습니다**:**

- **Take_images_for_calibration.py**”
- ”**Main_Stereo_Vision_Prog.py**”

두 가지 프로그램을 하나로**.txt**컴퓨터에 Python이 설치되어 있지 않은 경우에도 동일한 폴더에서 버전을 찾을 수 있습니다.