임베디드응용및실습 과제

10주차. 과제 보고서

과 목 명	임베디드응용및실습
학 번	2020161082
이 름	이강산
제 출 일	2025년 11월 10일

과제

lec8 강의노트 마지막의 과제를 수행하고 결과를 제출하시오.

1. OpenCV를 사용하여 라즈베리파이 카메라에서 받은 실시간 영상으로 얼굴 검출

- 얼굴 검출 시 사각형 박스가 표시되도록 함
- 소스 코드 및 나의 얼굴 검출 영상 제출

- 코드

```
import numpy as np
import cv2
import os

haar_path = cv2.data.haarcascades
face_xml_path = os.path.join(haar_path, 'haarcascade_frontalface_default.xml')
eye_xml_path = os.path.join(haar_path, 'haarcascade_eye.xml')

# 수정된 전체 경로로 CascadeClassifier를 로드합니다.
face_cascade = cv2.CascadeClassifier(face_xml_path)
eye_cascade = cv2.CascadeClassifier(eye_xml_path)

if face_cascade.empty():
    print(f"오류: 얼굴 XML 파일 로드 실패. 경로: {face_xml_path}")
    exit()
if eye_cascade.empty():
    print(f"오류: 눈 XML 파일 로드 실패. 경로: {eye_xml_path}")
    exit()

cap = cv2.VideoCapture(0, cv2.CAP_V4L)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

while (True):
    ret, img = cap.read()
    img = cv2.flip(img, -1)

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray, 1.2, 5)
    print("Number of faces detected: " + str(len(faces)))

    for (x, y, w, h) in faces:
        img = cv2.rectangle(img, (x, y), (x + w, y + h), (255, 0, 0), 1)
        roi_gray = gray[y:y + h, x:x + w]
```

```
            roi_color = img[y:y + h, x:x + w]
            eyes = eye_cascade.detectMultiScale(roi_gray)
            for (ex, ey, ew, eh) in eyes:
                cv2.rectangle(roi_color, (ex, ey), (ex + ew, ey + eh), (0, 255, 0), 1)

        cv2.imshow('img', img)
        k = cv2.waitKey(30) & 0xff
        if k == 27:
            break

cap.release()
cv2.destroyAllWindows()
```

- 실행 결과

- 고찰

라즈베리파이의 카메라를 통해 실시간 영상을 입력, 화면 속의 얼굴과 눈을 검출하는 코드입니다.

① 모델 로드: cv2.CascadeClassifier를 사용하여, 사전 학습된 얼굴 검출 (haarcascade_frontalface_default.xml) 및 눈 검출(haarcascade_eye.xml) 모델을 초기화합니다.

② 카메라 초기화: cv2.VideoCapture(0)를 통해 라즈베리파이 카메라를 활성화하고, cap.set을 이용해 처리할 영상의 해상도를 640x480으로 설정합니다.

③ 실시간 프레임 캡처: while(True) 루프에 진입하여, 카메라가 열려있는 동안 cap.read()를 통해 반복적으로 프레임을 1장씩 읽어옵니다.

④ 이미지 전처리: 검출 정확도를 높이기 위해, 읽어온 컬러 프레임(img)을 cv2.cvtColor를 사용해 흑백(gray) 이미지로 변환합니다.

⑤ 얼굴 검출: 흑백 이미지에서 face_cascade.detectMultiScale을 호출하여 얼굴 영역(faces)을 검출합니다.

⑥ 관심 영역(ROI) 설정 및 눈 검출: 검출된 각 얼굴 영역(for (x, y, w, h) in faces:)을 관심 영역(ROI)으로 지정합니다. 그 후, 흑백 ROI (roi_gray) 안에서만 eye_cascade.detectMultiScale을 호출하여 눈 영역(eyes)을 효율적으로 검출합니다.
⑦ 결과 시각화: cv2.rectangle을 사용해 원본 컬러 프레임(img)에 검출된 얼굴(파란색)과 눈(초록색)의 위치에 사각형을 그립니다. 최종 결과 이미지를 cv2.imshow로 화면에 출력하고 , cv2.waitKey로 다음 프레임까지 대기합니다.

모터 제어 예제와 달리, 이 코드는 별도의 스레드를 사용하지 않고 단일 스레드의 while 루프 내에서 순차적으로 모든 작업을 처리합니다. 매 프레임마다 [캡처 -> 흑백 변환 -> 얼굴 검출 -> (얼굴 검출 시) 눈 검출 -> 사각형 그리기 -> 화면 출력]의 과정이 정해진 순서대로 반복됩니다. 검출 알고리즘은 OpenCV에서 제공하는 Haarcascade 모델을 사용했습니다. 이는 detectMultiScale 함수를 통해 이미지 피라미드를 생성하며 객체를 검출하는 방식입니다.

부족한 점으로는, 현재 코드는 단일 스레드에서 영상 처리와 검출을 모두 수행하므로 detectMultiScale 함수의 연산 지연이 발생하면 실시간 영상(imshow)이 멈추거나 랙이 걸리는 현상이 발생할 수 있습니다. 또한, Haarcascade 모델은 정면 얼굴 인식률은 높으나 마스크를 쓰거나 얼굴 각도가 조금만 틀어져도 인식률이 급격히 떨어지는 한계가 있습니다. 모터 제어 예제처럼 스레드를 도입하여, 카메라 캡처/출력 스레드와 얼굴/눈 검출 스레드를 분리한다면, 검출에 시간이 다소 걸리더라도 영상은 부드럽게 재생되도록 성능을 개선할 수 있습니다.

2. 첨부된 4장의 이미지를 라인 트레이서 용도로 얻었다고 가정하고, 4장의 영상에서 노란색 또는 하얀색 선을 추출하여 표기하시오.

- 영상 표기하는 방법은 자유롭게 한다
- 사각형, 라인, 선만 남기고 다 검게 등등...
- 제안 알고리즘을 4장의 영상에 동일하게 적용 시, 성능이 보장되도록 한다
- 영상 크기 변경, 크롭, 컬러 변경 등 자유롭게 할 수 있다
- 소스 코드와 라인 표기된 4장의 영상 제출

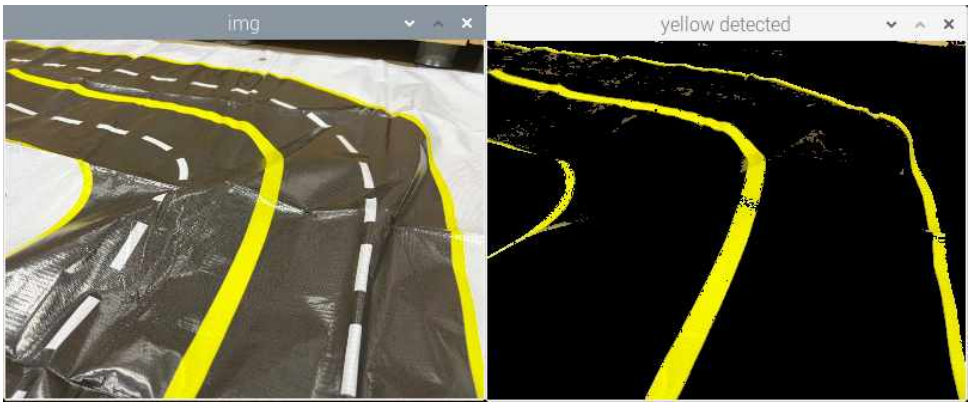
- 코드

```
import cv2
import numpy as np

img = cv2.imread("1.jpg", cv2.IMREAD_COLOR) # 불러올 이미지 파일에 따라 바꿔주기

if img is not None:
    img = cv2.resize(img, (400, 300))
    print('img shape : ', img.shape)

    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    yellow_lower = np.array([20, 100, 100])
```

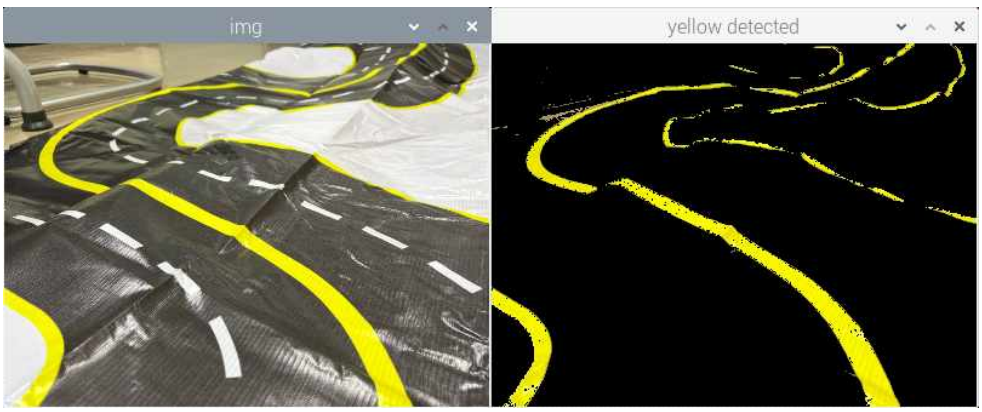
```
    yellow_upper = np.array([30, 255, 255])

    yellow_mask = cv2.inRange(hsv, yellow_lower, yellow_upper)

    result = cv2.bitwise_and(img, img, mask=yellow_mask)

    img_1ch = img.copy()
    img_1ch[:, :, 0] = 0
    img_1ch[:, :, 2] = 0

    cv2.imshow("img", img)
    cv2.imshow("yellow detected", result)

    cv2.waitKey(0)
    cv2.destroyAllWindows()

else:
    print("Img file not found")
```

- 실행결과

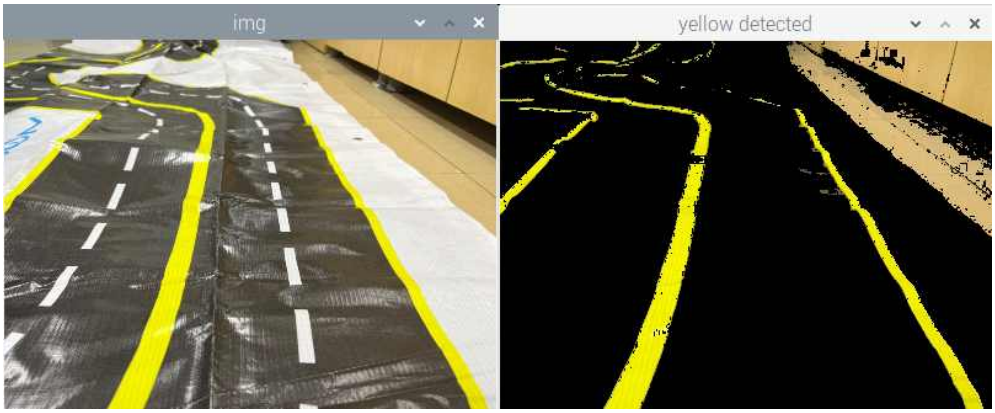
- 1.jpg



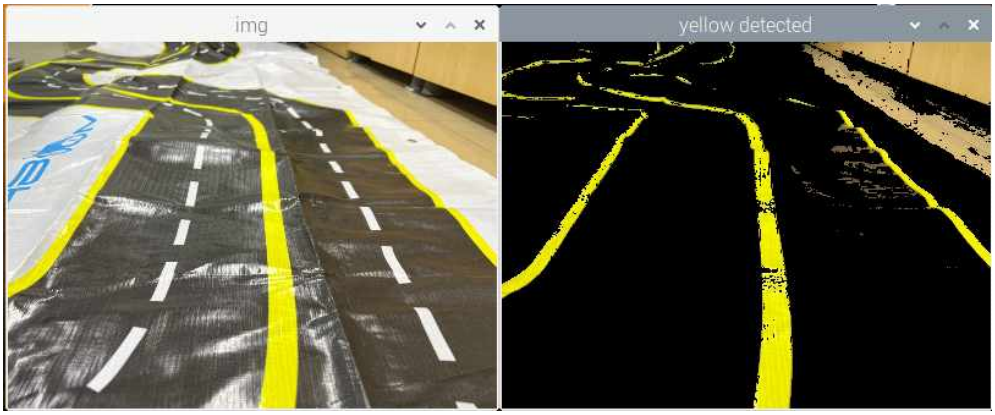
- 2.jpg



- 3.jpg



- 4.jpg



- **고찰**

지정된 이미지 파일("4.jpg")을 읽어와, HSV 색상 공간을 이용해 사용자가 정의한 '노란색' 범위에 해당하는 영역만 검출하여 화면에 출력하는 코드입니다.

① 이미지 로드: cv2.imread를 사용하여 정적 이미지 파일을 컬러 모드로 불러옵니다.

② 예외 처리: if img is not None: 구문을 통해 파일 로드에 실패했을 경우(None 반환 시) else문의 오류 메시지를 출력하고, 성공 시에만 메인 로직을 실행합니다.

③ 이미지 전처리: cv2.resize를 이용해 입력 이미지를 400x300 픽셀 크기로 강제 조절하여 연산 속도와 화면 출력 크기를 표준화합니다.

④ 색상 공간 변환: cv2.cvtColor를 사용해 BGR 색상 공간의 원본 이미지를 HSV(색상, 채도, 명도) 색상 공간의 이미지(hsv)로 변환합니다.

⑤ 색상 범위 정의: np.array를 활용해 검출하려는 노란색 계열의 HSV 최솟값(yellow_lower, H:20)과 최댓값(yellow_upper, H:30)을 정의합니다.

⑥ 마스크 생성: cv2.inRange 함수를 호출하여, HSV 이미지에서 ⑤에서 정의한 노란색 범위에 속하는 픽셀은 흰색(255), 속하지 않는 픽셀은 검은색(0)으로 구분하는 바이너리 마스크(yellow_mask)를 생성합니다.

⑦ 결과 추출: cv2.bitwise_and 연산을 수행하여, 원본 컬러 이미지(img)에서 yellow_mask가 흰색인 영역(즉, 노란색 영역)의 픽셀 값만 원본 그대로 추출한 result 이미지를 생성합니다.

⑧ 결과 시각화: cv2.imshow를 사용해 원본 이미지("img")와 노란색만 검출된 최종 결과물("yellow detected")을 별도의 창에 출력하고, cv2.waitKey(0)로 사용자의 키 입력을 대기합니다.

이 코드는 별도의 스레드나 while 루프 없이, 단일 스레드에서 순차적으로 모든 작업을 처리합니다.

메인 if 블록이 True일 경우, [이미지 리사이즈 -> HSV 변환 -> 범위 정의 -> 마스크 생성 -> Bitwise 연산 -> 화면 출력]의 과정이 정해진 순서대로 한 번 실행되고 사용자의 키 입력을 기다린 뒤 종료됩니다. 핵심 검출 알고리즘으로는 OpenCV의 HSV 색상 공간 변환 및 inRange 함수를 사용했습니다. 이는 BGR이나 RGB 공간에서는 조명이나 음영에 따라 값 변동이 심해 검출이 어려운 특정 색상을, 색상(H) 채널을 기준으로 비교적 일정하게 분리해낼 수 있는 고전적이면서도 효과적인 방식입니다.

부족한 점으로는, yellow_lower와 yellow_upper로 정의된 HSV 범위가 고정값이라는 점입니다. 이 범위는 파일('1.jpg', '2.jpg', '3.jpg', '4.jpg')의 특정 노란색에는 잘 동작할 수 있으나, 조명 환경이 다르거나 다른 톤의 노란색(예: 개나리색, 레몬색)은 검출하지 못할 수 있습니다. 또한, inRange 방식은 극단적인 그림자나 하이라이트 영역에서는 색상이 왜곡되어 검출 성능이 저하될 수 있습니다. 성능을 개선하기 위해, cv2.createTrackbar와 같은 GUI 도구를 도입하여 HSV 범위를 실시간으로 조절하며 최적의 임계값을 찾도록 코드를 확장할 수 있습니다. 또는, 실시간 영상 처리를 위해 이 로직을 cv2.VideoCapture로 연 카메라의 while 루프 안으로 옮겨 적용할 수도 있습니다.