

OpenCV + 라즈베리파이 + 아두이노 자율주행 RC카 만들기



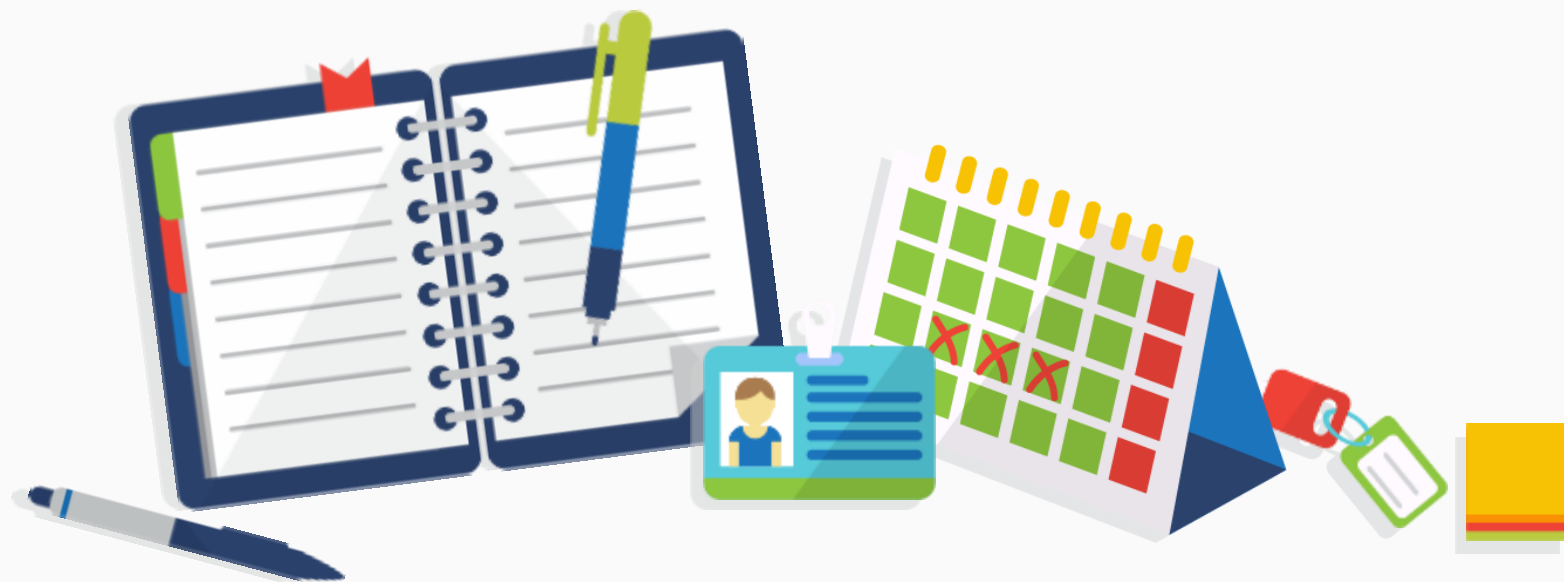
김도연, 김아영, 이나겸



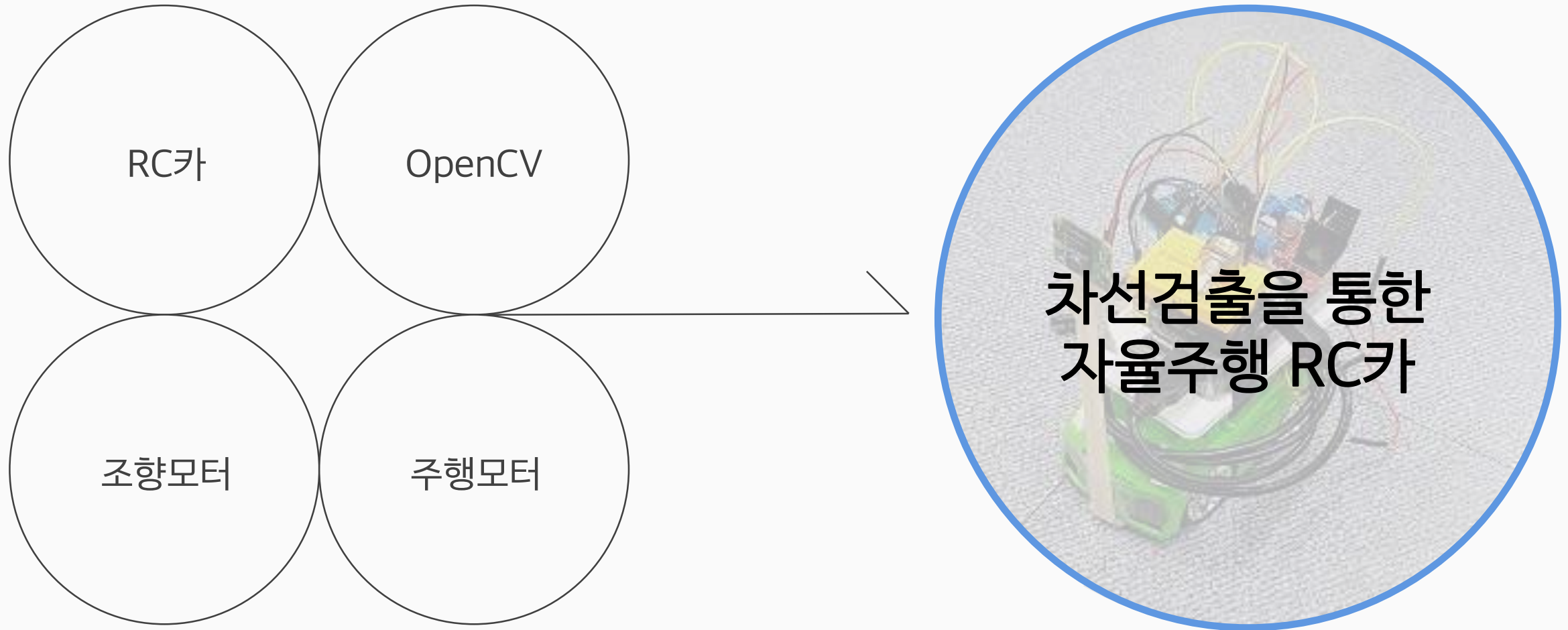
INDEX

OpenCV + 라즈베리파이 + 아두이노
자율주행 RC카 만들기

제작개요	01
관련이론	02
부품사양	03
제작과정	04
오류개선	05
영상 및 관리	06



01. 제작개요



02. 관련이론 - Canny Edge

Canny Edge

```
import cv2                # opencv 사용
import numpy as np

def grayscale(img):        # 흑백이미지로 변환
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

def canny(img, low_threshold, high_threshold):    # Canny 알고리즘
    return cv2.Canny(img, low_threshold, high_threshold)

def gaussian_blur(img, kernel_size):              # 가우시안 필터
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

image = cv2.imread('solidWhiteCurve.jpg')        # 이미지 읽기
height, width = image.shape[:2]                  # 이미지 높이, 너비

gray_img = grayscale(image)                      # 흑백이미지로 변환
blur_img = gaussian_blur(gray_img, 3)            # Blur 효과
canny_img = canny(blur_img, 70, 210)             # Canny edge 알고리즘 (추천 1:2, 1:3)

cv2.imshow('result',canny_img)                   # Canny 이미지 출력
cv2.waitKey(0)
```

출처 : <https://blog.naver.com/windowsub0406/220894160982>

01 Canny Edge 검출 알고리즘

엣지(Edge) 검출은 영상 요소를 기술하기 때문에 중요한 시각 정보를 갖는다.

첫번째, 가우시안 필터링을 하여 영상을 부드럽게 한다.

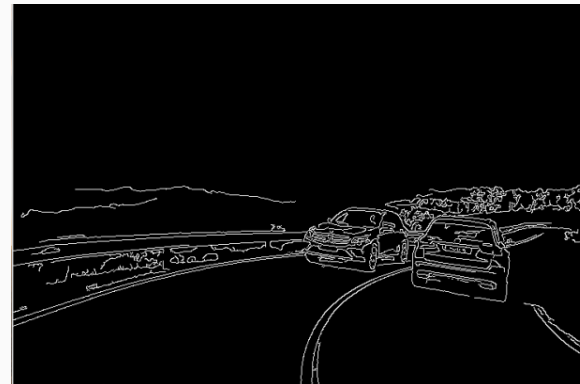
두번째, Sobel 연산자를 사용하여 기울기 벡터의 크기를 계산한다.

세번째, 가느다란 엣지를 얻기 위해 3*3 창을 사용하여 기울기 벡터 방향에서
기울기 크기가 최대값인 화소만 남기고 나머지는 0으로 억제한다.

네번째, 연결된 엣지를 얻기위해 두 개의 임계값을 사용한다.

높은 값의 임계값을 사용하여 기울기 방향에서 낮은 값의 임계값이 나올 때까지
추적하며 엣지를 연결하는 히스테리시스 임계값 방식을 사용한다.

02 실행 결과 영상



02. 관련이론 - Hough Transform (1)

Hough Transform

```
def grayscale(img): # 흑백이미지로 변환
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

def canny(img, low_threshold, high_threshold): # Canny 알고리즘
    return cv2.Canny(img, low_threshold, high_threshold)

def gaussian_blur(img, kernel_size): # 가우시안 필터
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

def region_of_interest(img, vertices, color3=(255,255,255), color1=255): # ROI
    셋팅
    mask = np.zeros_like(img) # mask = img와 같은 크기의 빈 이미지
    if len(img.shape) > 2: # Color 이미지(3채널)라면 :
        color = color3
    else: # 흑백 이미지(1채널)라면 :
        color = color1
    # vertices에 정한 점들로 이뤄진 다각형부분(ROI 설정부분)을 color로 채움
    cv2.fillPoly(mask, vertices, color)
    # 이미지와 color로 채워진 ROI를 합침
    ROI_image = cv2.bitwise_and(img, mask)
    return ROI_image
```

```
def draw_lines(img, lines, color=[0, 0, 255], thickness=2): # 선 그리기
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap): # 허프 변환
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]),
minLineLength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)

    return line_img

def weighted_img(img, initial_img,  $\alpha$ =1,  $\beta$ =1.,  $\lambda$ =0.): # 두 이미지 오버랩 하기
    return cv2.addWeighted(initial_img,  $\alpha$ , img,  $\beta$ ,  $\lambda$ )
```

02. 관련이론 - Hough Transform (2)

Hough Transform

```
image = cv2.imread('solidWhiteCurve.jpg') # 이미지 읽기

height, width = image.shape[:2] # 이미지 높이, 너비

gray_img = grayscale(image) # 흑백이미지로 변환

blur_img = gaussian_blur(gray_img, 3) # Blur 효과

canny_img = canny(blur_img, 70, 210) # Canny edge 알고리즘

vertices = np.array([(50,height),(width/2-45, height/2+60), (width/2+45, height/2+60), (width-50,height)], dtype=np.int32)

ROI_img = region_of_interest(canny_img, vertices) # ROI 설정

hough_img = hough_lines(ROI_img, 1, 1 * np.pi/180, 30, 10, 20) # 허프 변환

result = weighted_img(hough_img, image) # 원본 이미지에 검출된 선 오버랩

cv2.imshow('result',result) # 결과 이미지 출력

cv2.waitKey(0)
```

출처 : <https://blog.naver.com/windowsub0406/220894462409>

01 Hough Transform 검출

$x\cos\theta + y\sin\theta = r$ 로 어떠한 점을 지나는 직선을 구할 수 있다.

X, y는 변수, θ 와 r은 상수이다.

점들이 놓여있는 평면에서 어떤 직선 위에 있는 점들의 개수를 파악 할 수 있는지가 핵심이다.

어떤 선에서 교점이 생겼다는 것은 n개의 곡선이 한 점에서 만나면 n개의 점이

하나의 직선위에 있다는 것을 뜻한다.

해당 교점의 좌표를 가지고 xy평면에 직선을 그리면 n개의 점을 가지고 직선을 그릴 수 있다.

이런 과정을 통해 어떤 선 위에 점이 몇 개 올라와 있는지 그 개수를 파악할 수 있다.

02 실행 결과 영상



02. 관련이론 - ROI

ROI (Region Of Interest)

def roi(img, vertices):

```
#blank mask:
mask = np.zeros_like(img)      # np.zeros = np array의 값을 0으로 초기화 하는 것
#filling pixels inside the polygon defined by vertices with the fill color
cv2.fillPoly(mask, vertices, 255) # cv2.fillPoly = fillPoly : 오브젝트, 면을 생성한다.
# mask : 마스크, vertices : 위에서 받아온 값, 255 : 영역을 정하는 것 (색)
# 영역을 설정할 때 한개의 값만 쓰면 RGB 세개 다 똑같이 255로 설정한 것과 똑같다.
#returning the image only where mask pixels are nonzero
masked = cv2.bitwise_and(img, mask)
# and연산을 해서 겹치는 부분을 제외한 나머지는 날린다.
return masked
```

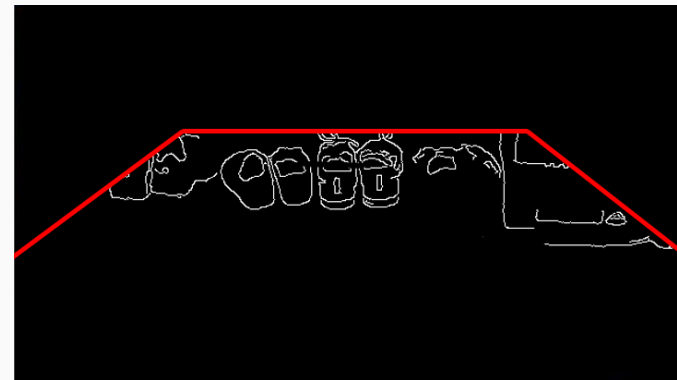
```
Vertices      =      np.array([[0,360],[0,240],[160,120],[480,120],[640,240],[640,320]],
dtype=np.int32)
# 리스트 안에 리스트는 2차원 배열이다. 튜플로 점을 정하고 있다.
# 리스트를 만들어서 쓰면 그것을 지정한 것도 하나의 리스트여야 안의 리스트를 볼 수 있다.
# 그래서 [vertices] 라고 리스트로 만들어줬다.
Roi_img = roi(dst, [vertices])
```

출처 : <https://blog.naver.com/windowsub0406/220894645729>

01 ROI (Region Of Interest)

ROI는 관심영역에만 집중하여 이미지 처리의 연산량을 줄이는 것이다.
카메라를 켜고 차선을 비추고 주행을 해보면 차선이 나타나는 부분은 중심 부근에서 사다리꼴로 계속 찍히는 것을 알 수 있었다.
그렇기 때문에 numpy 배열로
라즈베리파이에서 코드를 돌리면 프로세스를 어마하게 잡아먹는 것을 확인할 수 있었다.
카메라도 같이 딜레이가 걸려 느리게 움직이는 것을 알 수 있었다.
차선이 있을만한 구간을 정하고 그 안에서 차선을 찾는 것이 더 효율적이라고 생각되었다.

02 실행 결과 영상

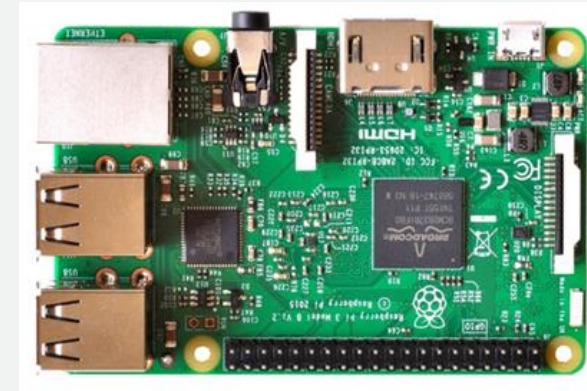


03. 부품사양



01. 아두이노

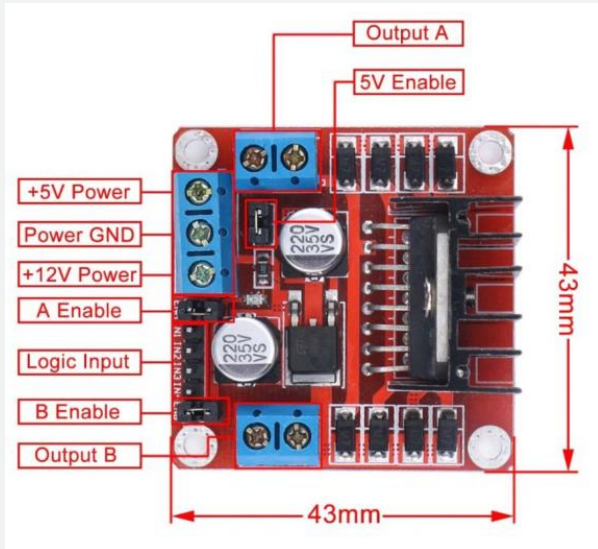
- MCU : ATmega328P
- 작동전압 : 5V
- 입력전압 (권장값) : 7-12V
- 입력전압 (한계치) : 6-20V
- 디지털 I/O 핀 : 14개 (6개는 PWM 제공)
- 아날로그 입력 핀 : 6개
- I/O 핀당 DC 전류 : 40mA
- 플래시 메모리 : 32Kb
0.5Kb가 부트로더에 의해 사용됨
- SRAM : 2Kb (ATmega328P)
- EEPROM : 1Kb (ATmega328P)
- Clock Speed : 16Mhz



02. 라즈베리파이

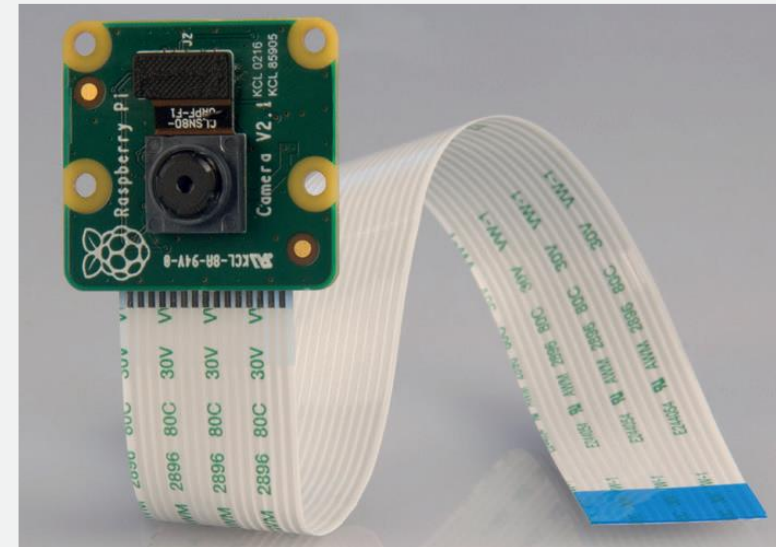
- 프로세서 : Broadcom BCM2387 칩셋
1.2GHz 쿼드 코어 ARM Cortex-A53
- CSI-2 : PI 카메라 연결용 카메라포트
- 마이크로 SD 슬롯
- 전원 : 마이크로 USB 소켓 5V / 2.1A
- 802.11 b / g / n 무선 LAN
- GPU : 듀얼 코어 VideoCore IV®
멀티미디어 Co-Processor
- GPIO : 40pin

03. 부품사양



03. L298N 모터드라이버

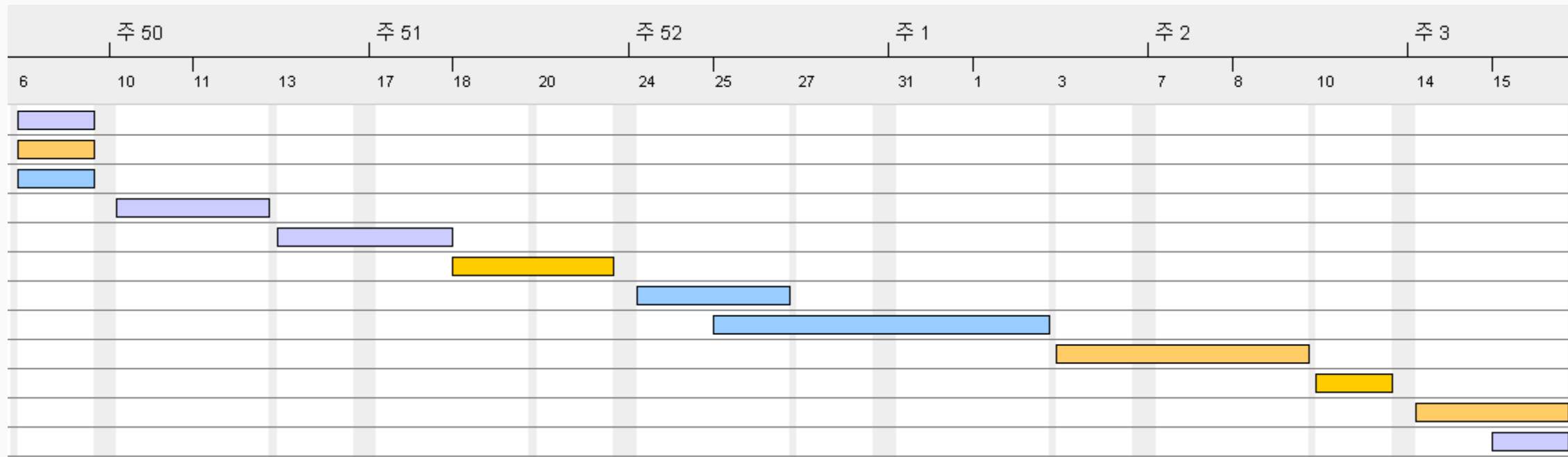
- 입력전압 : DC 3.2V ~ 40V
- Driver
L298N Dual H Bridge DC Motor Driver
- 공급전원 : DC 5V ~ 35V
- 피크전류 : 2A
- 동작전류범위 : 0 ~ 36mA



04. 파이카메라

- 최대 이미지 전송 속도
1080p : 30fps (encode and decode)
720p : 60fps
- 해상도 : 8-megapixel
- 렌즈 크기 : 1/4"(인치)
- 치수 : 23.86 × 25 × 9 mm

04. 제작과정

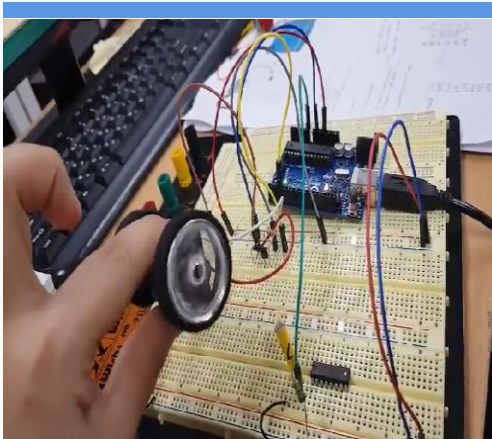


주행모터 제어	18. 12. 6	18. 12. 6
OpenCV 설치	18. 12. 6	18. 12. 6
조향모터 제어	18. 12. 6	18. 12. 6
앞바퀴, 뒤바퀴 조합	18. 12. 10	18. 12. 11
모터드라이버로 바퀴제어	18. 12. 13	18. 12. 17
차선인식	18. 12. 18	18. 12. 20

조향모터	18. 12. 24	18. 12. 25
파이썬 조향각도	18. 12. 25	19. 1. 1
OpenCV 코드간소화	19. 1. 3	19. 1. 8
코드 함수화	19. 1. 10	19. 1. 10
카메라 딜레이 해결	19. 1. 14	19. 1. 15
뒷바퀴 모터드라이버	19. 1. 15	19. 1. 15

주행모터 조향모터 OpenCV

04. 제작과정



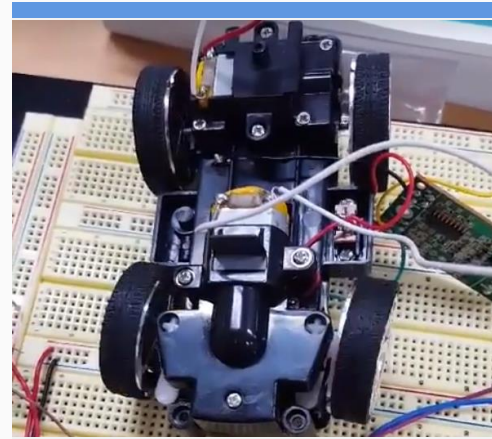
01. 모터제어

- DC모터로 속도 제어 (뒷바퀴)
- DC모터로 방향 제어 (앞바퀴)
- 모터 드라이버로 바퀴 제어



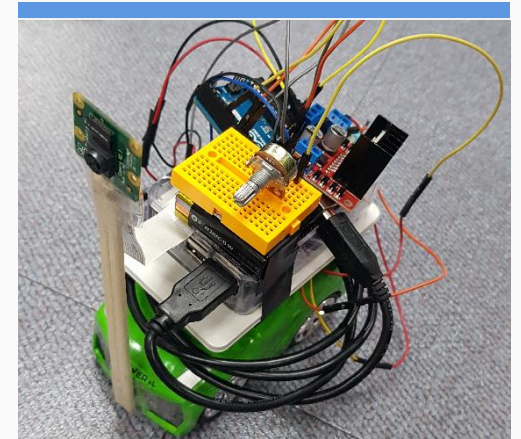
02. OpenCV

- 영상처리에 필요한 라이브러리 사용
- Canny, Hough, ROI 차선인식
- 라즈베리파이 + OpenCV + 파이썬



03. 아두이노

- 라즈베리파이로부터 값 받기
- 받은 값으로 모터 조향제어
- 코드 함수화 및 간소화

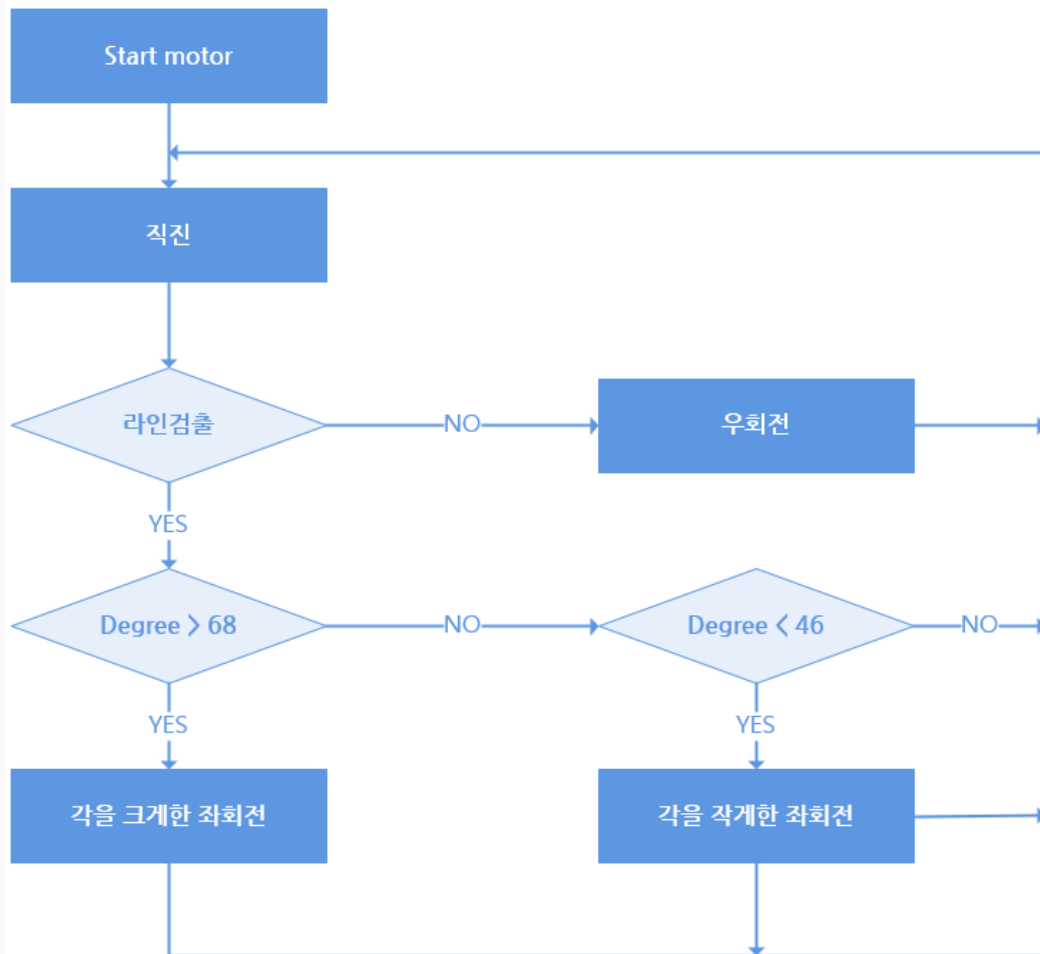


04. 외형제작

- 카메라 위치 선정
- 메인 부품 위치 선정
- 카메라 각도 조절

04-1. 개념설계

01 알고리즘

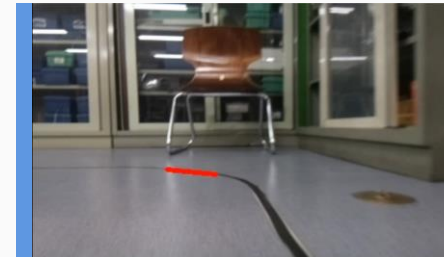


02 알고리즘 동작설명



라인 미 검출 시, 우회전

오른쪽 라인을 잡고 주행하기 때문에 라인을 검출하지 못할 시 조향모터를 조금씩 틀어서 오른쪽 라인을 검출하도록 한다.



각을 크게 한 좌회전

68도보다 각이 클 경우 DC모터의 회전속도와 지연시간을 조절하여 크게 회전할 수 있게 만들어서 원만하게 좌회전을 한다.

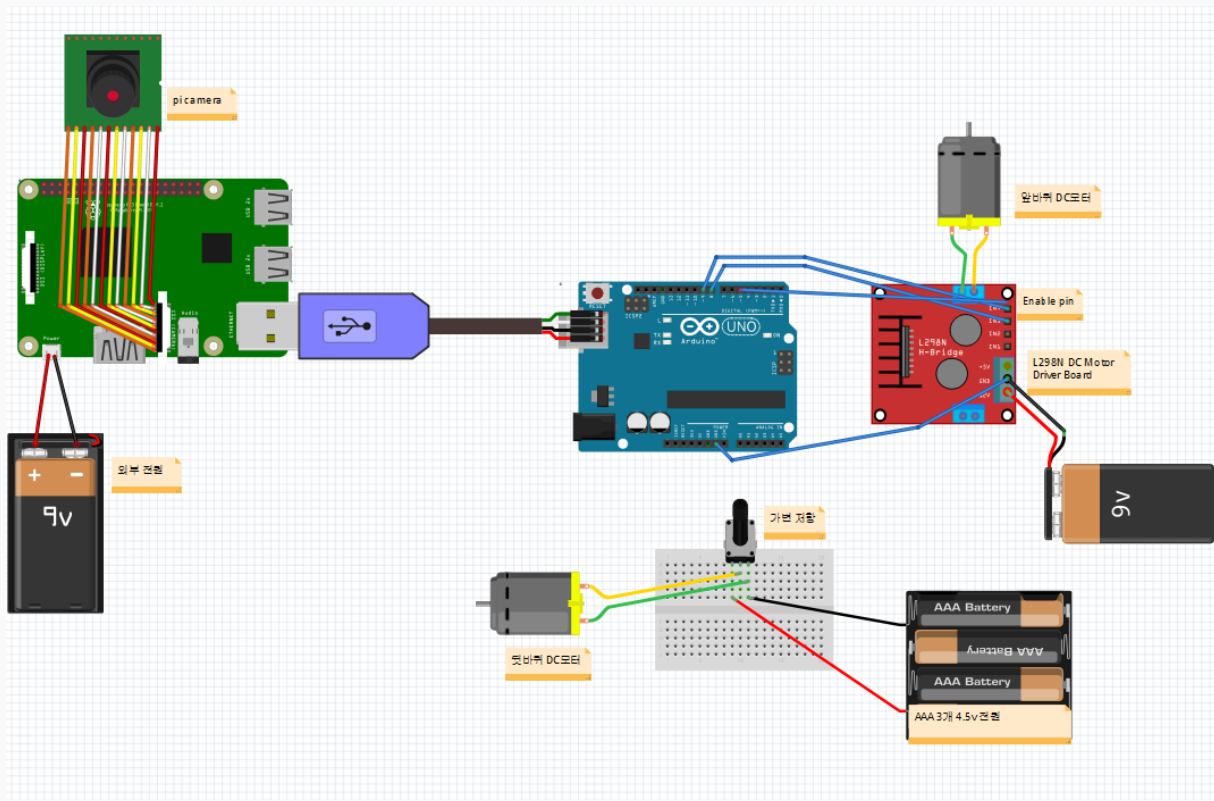


각을 작게 한 좌회전

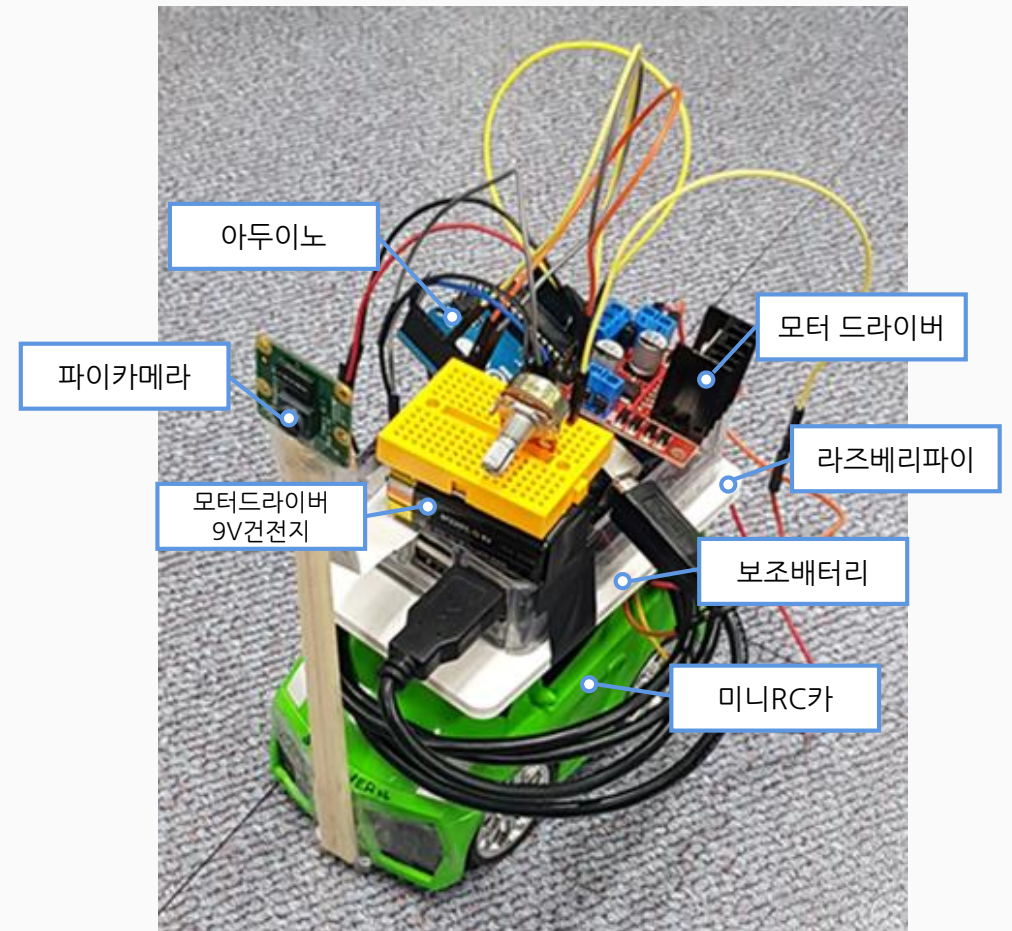
68도보다 작고, 46도보다 작으면 선은 검출하지만 중심에서 벗어난다. 좌측으로 조향모터를 조금씩 틀어서 트랙을 벗어나지 않도록 한다.

04-2. 상세설계

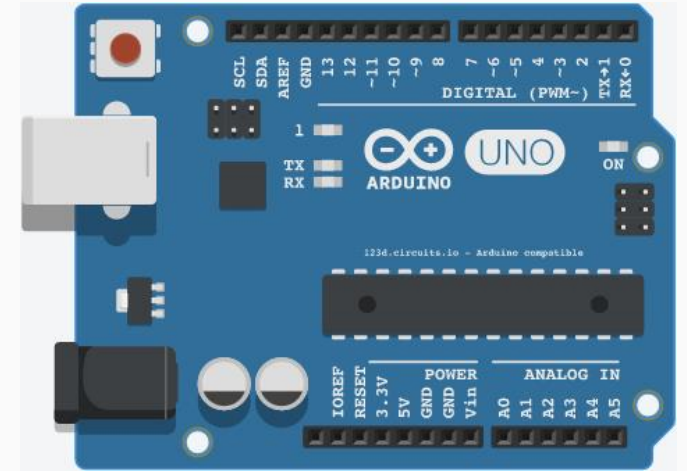
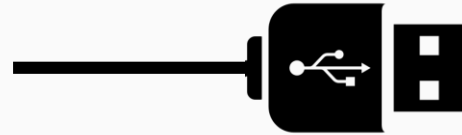
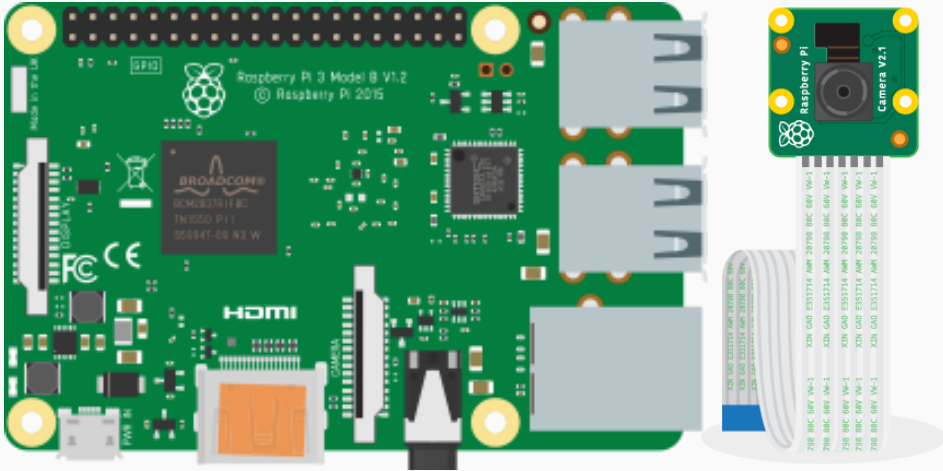
01 부품 구성도



02 작품사진



04-3. 동작설명




OpenCV + Python

```
void FromRasp(int turn)
{
    switch(turn)
    {
        case 1:    RightM(80); break;
        case 2:
            if(numberLeft==3){    numberLeft = 0;
            RightM(110);}
            else    BigLeftM();
            break;
        case 3:    LeftM();    break;
    }
}
```

Motor

```
if degree > 68:
    serialFromArduino.write(b'2')
elif degree < 46:
    serialFromArduino.write(b'3')
else:
    serialFromArduino.write(b'1')
```

05. 오류개선



실시간 신호처리로
좀 더 빨라진
자율주행 자동차

01 카메라 속도 개선

- 카메라를 켜두면 delay가 쌓여서 3초정도의 delay 발생
- 파이썬 코드의 모든 delay를 삭제
- 코드를 함수화, 간소화를 하여 CPU 부하를 줄여서 카메라 속도를 개선

02 아두이노 오버플로우 현상 해결

- 단위테스트를 통해 라즈베리파이에는 이상이 없는 것을 확인
- 아두이노의 Serial.parseInt가 1초동안 값을 모아서 원하는 출력이 나오지 않음
- setTimeout(30)으로 설정해서 데이터 수신 대기 시간을 0.03으로 변경

03 카메라 높이 조절

- 카메라의 위치가 초반과 많이 달라졌음을 인지
- 카메라의 위치를 고정하고자 파이카메라를 켜고 위치를 변경
- 이전과 같은 소스코드였음에도 주행을 완주하는 것을 확인

06. 영상 및 형상관리

01 자율주행 RC카 자동차 시점 / 외부 시점 영상



02 자율주행 자동차 형상관리 - GITLAB



논의 및 고찰

1. 뒷바퀴 제어 문제

가변저항으로 뒷바퀴를 제어해서 완주시키고 모터 드라이버로 제어를 시도하였더니 무게와 전력부족으로 주행할 수 없었다. 외형제작 시 무게를 고루 분포해서 만들었다면 해결할 수 있었을 것이다.

2. 카메라 문제

카메라의 각도와 높이의 고정이 어려워서 외부적인 요인에 의해 위치가 계속 변경되어 실험할 때 무엇이 문제인지 확인이 어려웠다. 카메라를 좀 더 확실하게 고정해서 실험을 진행한다면 좀 더 원하는 결과물을 만들기까지 시간이 단축되었을 것이다.

3. 하드웨어적인 문제

앞바퀴를 DC모터로 제어했을 때 각도의 제어가 제한적이었다. 서보모터나 스텝모터를 사용하여 좀 더 정밀하게 각도가 제어 된다면 좌회전이나 우회전 시 delay를 사용하지 않아도 정확한 값으로 회전을 할 수 있을 것이다.

