[두산로보틱스] 지능형 로보틱스 엔지니어

터토봇(TURtle auTOnomy roBOT)

협동3)디지털 트윈 기반 서비스 로봇 운영 시스템 구성

B-4 터토봇

[팀원] 이세현, 강인우, 이형연

[멘토] 김루진

K-Digital Training

목 차

K-Digital Training

01 프로젝트 개요

1

프로젝트 주제 및 선정 배경, 기획의도

차선 인식 기반으로 자율주행 하는 터틀봇에 매니퓰레이터를 결합해, 이동 중 작업 수행이 가능한 모바일 서비스 로봇 구현

2

프로젝트 내용

- 컴퓨터 비전 기반 차선 (가이드 라인) 인식

-자율 주행 로봇 제어

- Aruco marker 활용 매니퓰레이터 조작

3

활용 장비 및 재료

- Turtlebot3 Waffle + 매니퓰레이터
- NVIDIA Jetson
- LOGITECH C270 웹캠
- RVIZ
- Python
- Ubuntu 22.04

4

프로젝트 구조

- 실시간 카메라 영상에서 차선을 검출, 주행 방향 판단
- 매니퓰레이터의 경로 계획 및 작업 수행

-ROS2 기반 터틀봇 주행 및 시스템 구성

5

활용방안 및 기대 효과

- 제어된 실내 환경에서 AGV 등의 기술을 활용한 다양한 서비스 로봇 응용 가능
- 모바일 매니퓰레이터를 통해 다양한 작업 환경에서 물체 조작 등 복합 업무 수행으로 확장성 확보

02 프로젝트 팀 구성 및 역할

프로젝트 팀 구성 및 역할

훈련생	역할	담당 업무
이세현	팀장	시뮬레이션 환경 제작, GUI 제작, 차선 검출 알고리즘 구현
강인우	팀원	시뮬레이션 태스크 , 이미지 전처리, RO2기반 로봇제어 시스템 구현
이형연	팀원	교통 표지판 인식 모델 구축, aruco mark 인식 기반 매니퓰레이터 조작

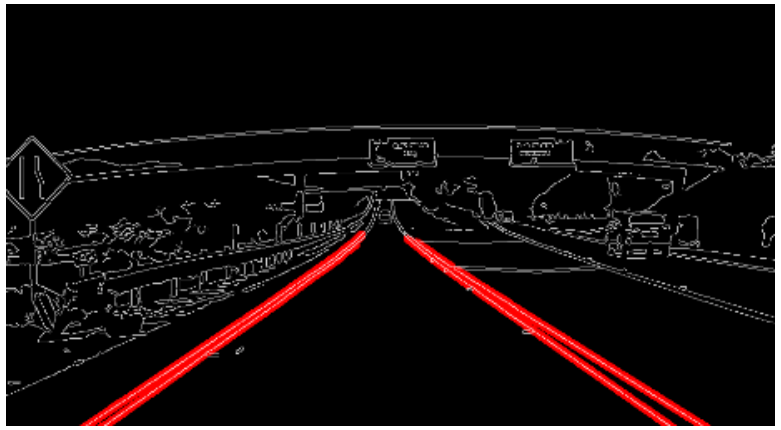
프로젝트 수행 절차 및 방법

프로젝트 수행 절차

구분	기간	활동	비고
사전 기획	6.9(월)	프로젝트 기획 및 기능 설계	구현 기능 설정
시뮬레이션	6.10(화) ~ 6.13(금)	가제보 시뮬레이션 주행 테스트	
기능 구현	6.16(월)~6.19(목)	차선 인식, 터틀봇 제어, 매니퓰레이터 제어	
통합 및 테스트	6.19(목)	통합 테스트, 리팩토링	주행 테스트
결과물 도출	6.19(목)	시연 영상 제작, 산출물 작성	

04 프로젝트 수행 경과

수행 경과



1. 차선 검출

2. 터틀봇 제어

3.아루코 마커 감지
및 매니퓰레이터 조작

04 프로젝트 수행 경과

▸ 이미지 전처리

차선 마스크 추출 및 정제

오탐지 제거

외부 조명, 반사, 그림자 영향 최소화.

- CLAHE 히스토그램 균일화 및 밝기/대비 적용
- 히스토그램 클리핑 기반 선형 스트레칭

HSV 색상 공간 추출

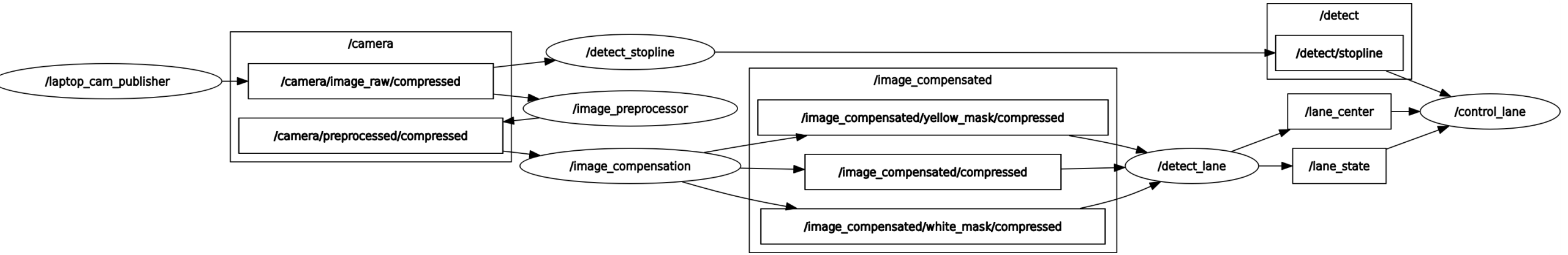
흰색, 노란색 차선 분리

컨투어 분석 적용

노이즈 감소, 가장 큰 윤곽선 추출.

프로젝트 수행 경과

노드, 토픽 그래프



04 프로젝트 수행 경과

▸ 이미지 발행

```
self.cap = cv2.VideoCapture(0)
self.cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc('M', 'J', 'P', 'G'))
self.cap.set(cv2.CAP_PROP_FPS, 25)
self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

self.cap.set(cv2.CAP_PROP_EXPOSURE, -6)

width = self.cap.get(cv2.CAP_PROP_FRAME_WIDTH)
height = self.cap.get(cv2.CAP_PROP_FRAME_HEIGHT)
self.get_logger().info(f"Laptop camera resolution: {width} x {height}")

if not self.cap.isOpened():
    self.get_logger().error("노트북 카메라 열기 실패!")
    exit(1)
else:
    self.get_logger().info("노트북 카메라 오픈 성공")

# 타이머 주기 (0.2초 → 5fps 정도)
self.timer = self.create_timer(0.2, self.publish_image)
```

이미지 발행 노드

- 카메라 선택
- 코덱 설정
- 프레임 설정 1280 x 720
- 발행 주기 설정

프로젝트 수행 경과

이미지 전처리

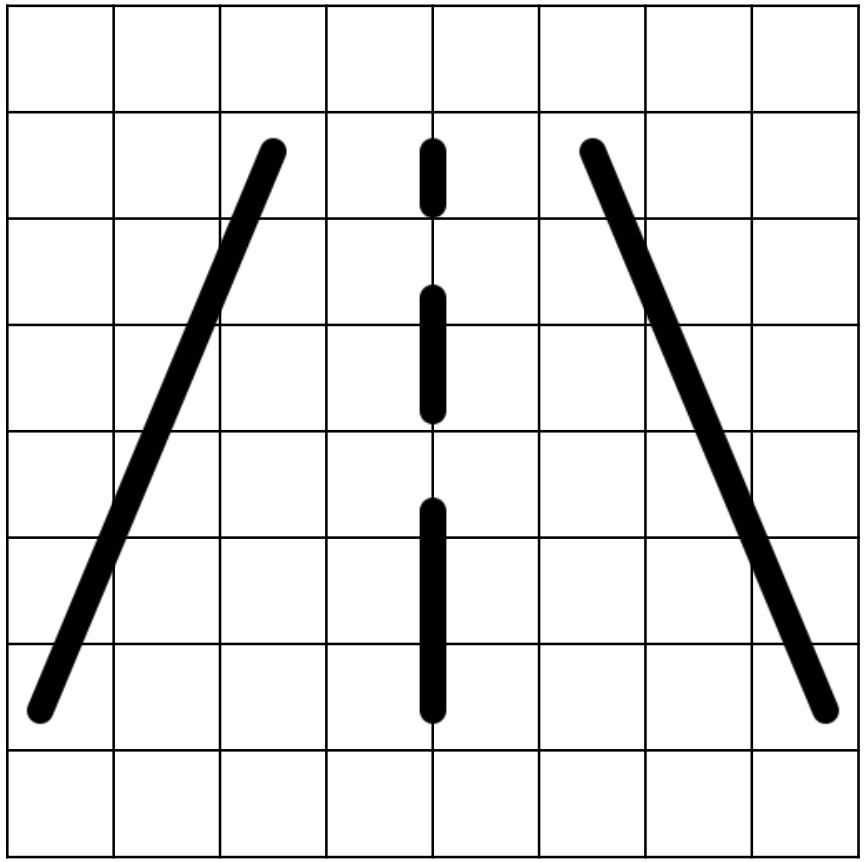
```
if self.use_preprocessing:
    self.get_logger().info('전처리 적용됨: HSV V채널 CLAHE + 밝기/대비 조절')

    # HSV로 변환 후 V 채널 CLAHE 적용
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)

    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    v_eq = clahe.apply(v)

    hsv_eq = cv2.merge((h, s, v_eq))
    processed = cv2.cvtColor(hsv_eq, cv2.COLOR_HSV2BGR)

    # 추가 밝기/대비 조절
    processed = cv2.convertScaleAbs(processed, alpha=0.9, beta=-50)

else:
    processed = frame
    self.get_logger().info('전처리 미적용 (원본 그대로)')
```

이미지 전처리 노드

- CLAHE 히스토그램 균일화
- 대비, 밝기 조절

프로젝트 수행 경과

이미지 전처리



Contrast Limited Adaptive Histogram Equalization (CLAHE)

1. HSV 변환, V 채널 분리.

2. V 채널 타일 분할, 각 타일별 히스토그램 평활화

3. 클리핑으로 과증폭 제한, 타일 경계 보간

3. HSV 채널 병합, BGR로 재변환.

프로젝트 수행 경과

이미지 전처리

```python
def keep_largest_contour(self, mask):
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if not contours:
        return np.zeros_like(mask)
    largest = max(contours, key=cv2.contourArea)
    result = np.zeros_like(mask)
    cv2.drawContours(result, [largest], -1, 255, thickness=cv2.FILLED)
    return result

def cbImageCompensation(self, msg_img):
    # 이미지 디코딩
    if self.sub_image_type == 'compressed':
        np_image_original = np.frombuffer(msg_img.data, np.uint8)
        cv_image_original = cv2.imdecode(np_image_original, cv2.IMREAD_COLOR)
    else:
        cv_image_original = self.cvBridge.imgmsg_to_cv2(msg_img, 'bgr8')

    clip_hist_percent = self.clip_hist_percent

    # 대비 보정 진행 (그레이스케일 히스토그램 기반)
    cv_image_compensated = np.copy(cv_image_original)
    hist_size = 256
    gray = cv2.cvtColor(cv_image_compensated, cv2.COLOR_BGR2GRAY)
    if clip_hist_percent == 0.0:
        min_gray, max_gray, _, _ = cv2.minMaxLoc(gray)
    else:
        hist = cv2.calcHist([gray], [0], None, [hist_size], [0, hist_size])
        accumulator = np.cumsum(hist)
        total_max = accumulator[hist_size - 1]
        clip_hist_percent_adjusted = clip_hist_percent * (total_max / 100.0) / 2.0
        min_gray = 0
        while accumulator[min_gray] < clip_hist_percent_adjusted and min_gray < hist_size - 1:
            min_gray += 1
        max_gray = hist_size - 1
        while accumulator[max_gray] >= (total_max - clip_hist_percent_adjusted) and max_gray > 0:
            max_gray -= 1
    input_range = max_gray - min_gray
    alpha = (hist_size - 1) / input_range if input_range != 0 else 1.0
    beta = -min_gray * alpha
    self.get_logger().info(
        f"[Clip {clip_hist_percent}] min_gray: {min_gray}, max_gray: {max_gray}, alpha: {alpha:.3f}, beta: {beta:.3f}")
    cv_image_compensated = cv2.convertScaleAbs(cv_image_compensated, alpha=alpha, beta=beta)
```

이미지 전처리 노드

- 히스토그램 기반 선형 스트레칭
- 이미지 컨투어
  (동일 색, 픽셀을 가진 한 영역만 추출)

프로젝트 수행 경과

이미지 전처리

1. CLAHE 단독 사용

지역(타일) 기반 명암 대비 향상
밝기 불균형이 심한 환경(터널, 그림자)에서 부드럽고 자연스럽게 개선
밝기 변화가 심한 차선 구간에서 디테일 보존에 효과적

➤ 장점: 국소 명암 대비 개선, 세부 구조 잘 보임

➤ 단점: 전역적인 대비 향상은 미흡할 수 있음

2. 히스토그램 클리핑 기반 선형 스트레칭 단독 사용

전체 이미지에서 극단값 제거 후 선형적으로 확대
전역 밝기 및 대비를 강하게 보정
➤ 장점: 한 장의 이미지에서 명암이 넓게 퍼졌을 때 빠르고 간결한 대비 향상,
이미지 명확도 증가

➤ 단점: 노이즈와 밝기 왜곡이 발생할 수 있으며, 그림자·글레어(반사광)에 민감

성능 개선 시각 요약

항목	CLAHE만	히스토그램만	CLAHE + 히스토그램
지역 대비 향상	👍	✕	✓
전역 대비 향상	✕	👍	✓
노이즈 억제	👍	✕	✓
그림자· 굴레어 대응	👍	✕	✓
차선 검출 안정성	중간	낮음	**높음**

프로젝트 수행 경과

이미지 전처리

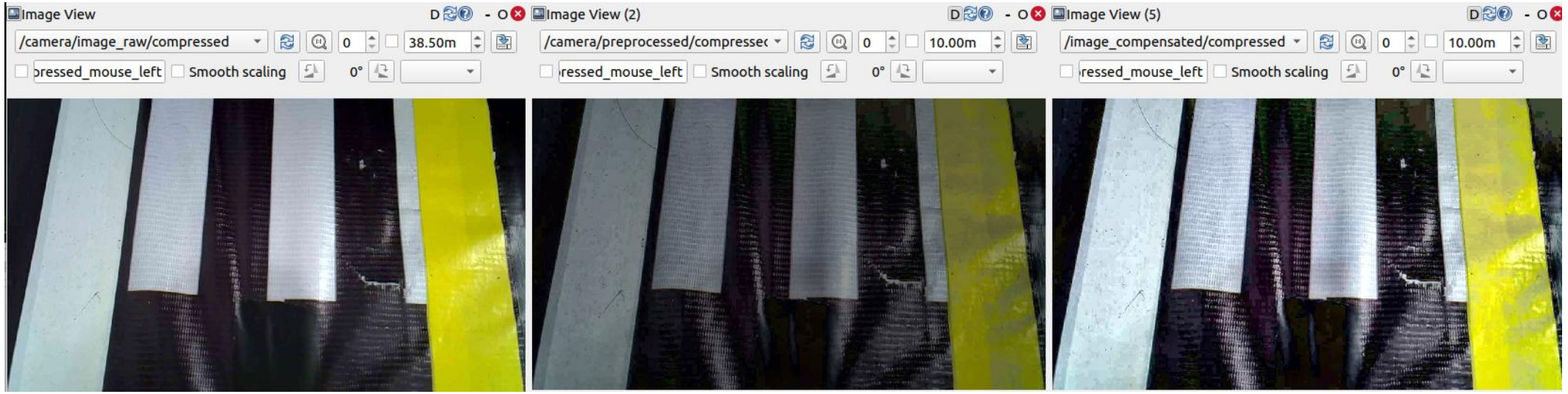
3. CLAHE → 히스토그램 스트레칭 결합

✅ 결합 순서:

CLAHE로 국소적 균일화 → 스트레칭으로 전역 대비 확장

항목	개선 내용
✅ 차선 인식률	어두운 영역 및 반사광 영역에서 라인 인식률 향상
✅ 밝기 안정성	그림자/글레어를 안정적으로 억제, 디테일 유지
✅ 적용 범용성	실내외, 야간 등 다양한 환경에서 균일한 성능 유지
✅ 정밀도	BEV 변환 및 마스크 후처리에 더 나은 입력 제공

< 원본 이미지 > ➕ → < CLAHE 히스토그램 + 밝기/대비 적용 > ➕ → < 히스토그램 기반 선형 스트레칭 >



04 프로젝트 수행 경과

▸ 이미지 전처리

```
# 원본 이미지의 차선 영역 4개 꼭짓점 (Source points)
src = np.float32([
(180, 400),
(70, 720),
(1230, 720),
(1140, 400)
])

# 변환 후 결과 이미지의 4개 꼭짓점 (Destination points)
dst = np.float32([
    (0, 0),
    (0, img_size[1]),
    (img_size[0], img_size[1]),
    (img_size[0], 0)
])
M = cv2.getPerspectiveTransform(src, dst)
Minv = cv2.getPerspectiveTransform(dst, src)
```

이미지 전처리 노드

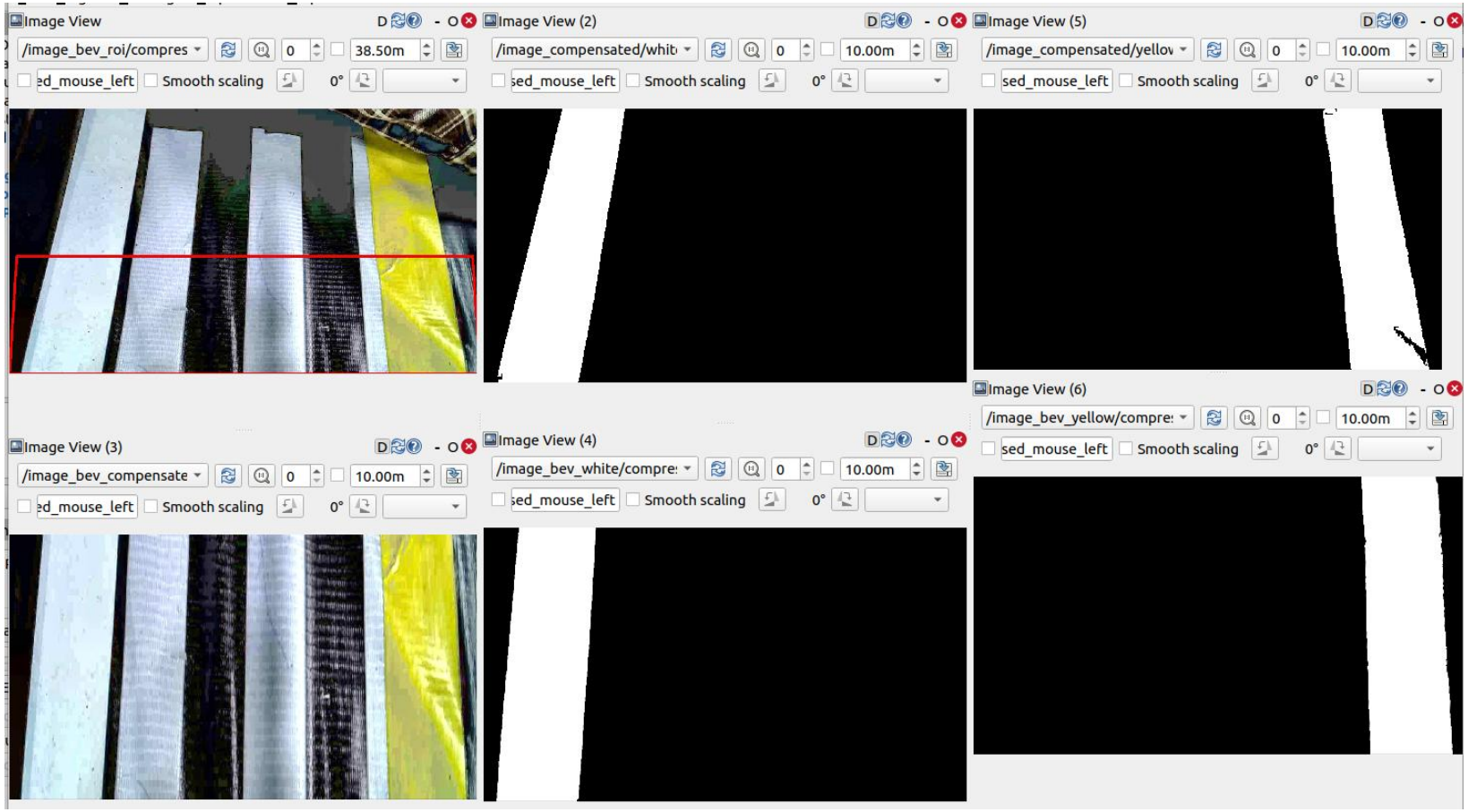
- 관심 영역(src) 및 변환 영역(dst) 정의
- 원근 변환 행렬 M 계산
  - 원본 이미지 → BEV 이미지
- 역원근 변환 행렬 Minv 계산
  - BEV 이미지 → 원본 이미지

04 프로젝트 수행 경과

▸ 이미지 전처리

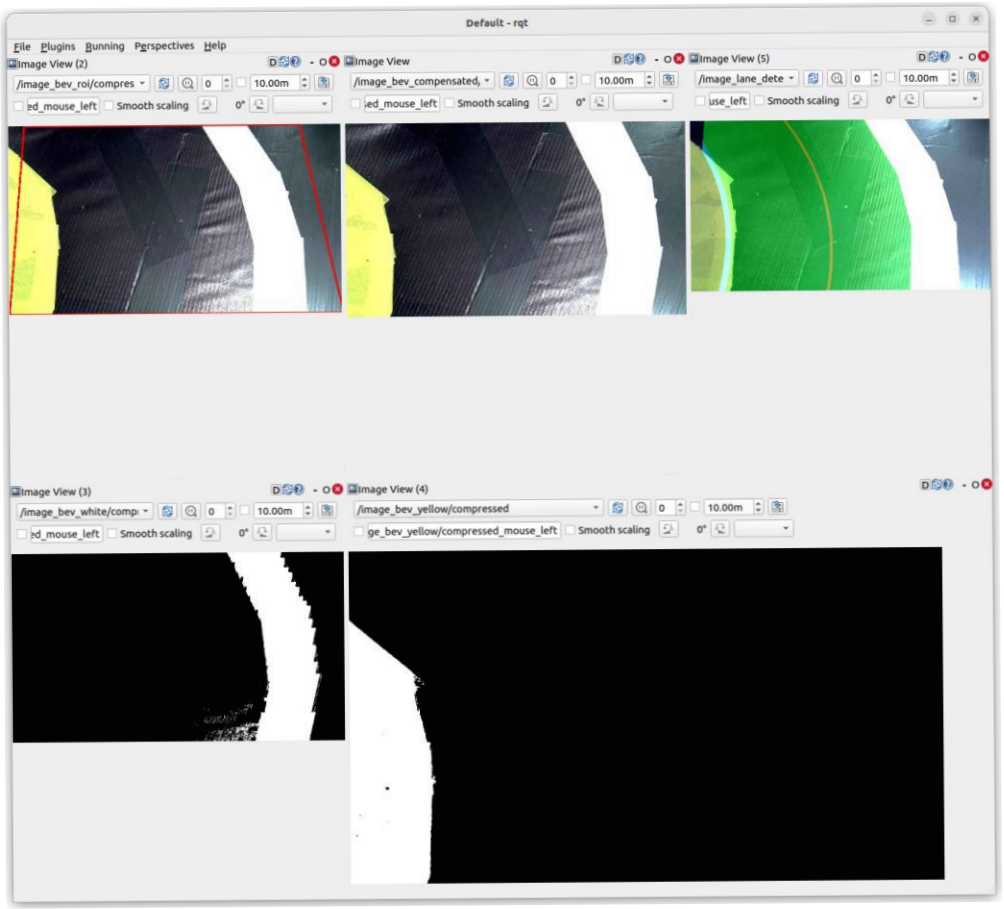
원본 영역에서 마스크 추출

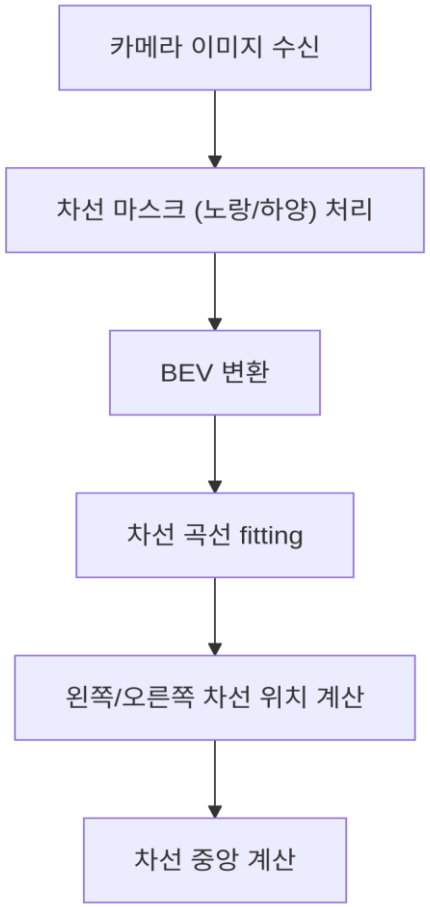
BEV 영역에서 마스크 추출



프로젝트 수행 경과

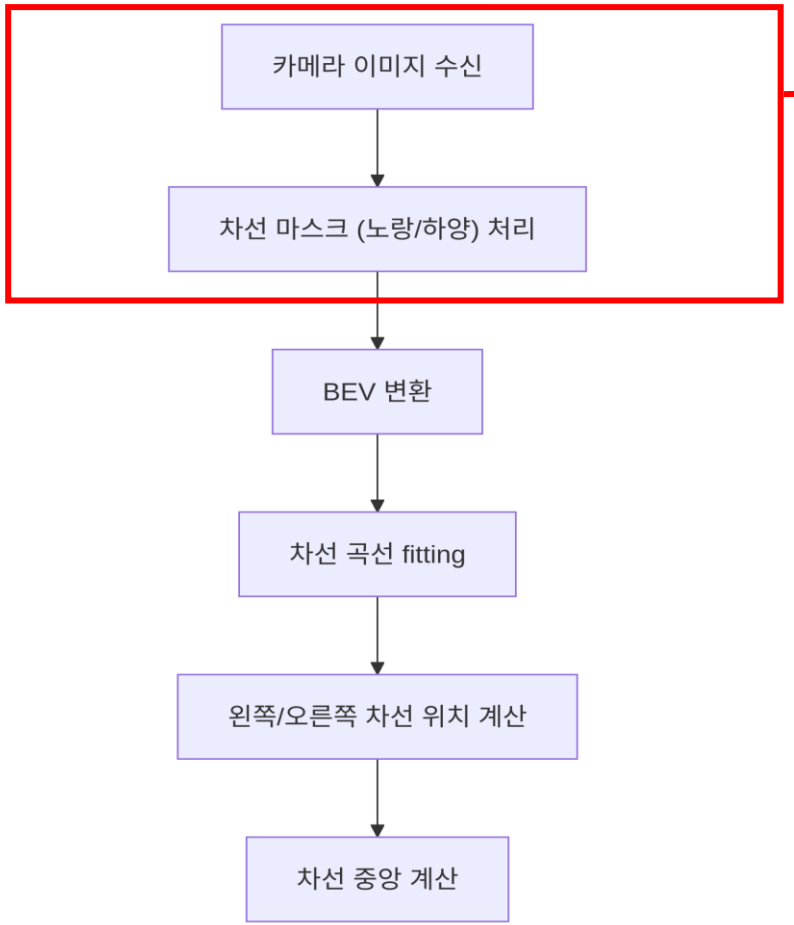
라인 검출





프로젝트 수행 경과

라인 검출 – 토픽 이미지 동기화



이미지 동기화

/image_compensated/compressed:
원본 보정 이미지

/image_compensated/white_mask/compressed:
흰색 차선 마스크

/image_compensated/yellow_mask/compressed:
노란색 차선 마스크

- **이미지 동기화를 통해**
  ➡️ 정확도, 시각적 일관성, 상태 판단 안정성 향상

- 비동기식 수신일 경우
  마스크 이미지와 원본 이미지 간의 불일치 현상 → **불안정한 판단 유발**

04 프로젝트 수행 경과

라인 검출 – 토픽 이미지 동기화

```
# message_filters로 동기화 설정
sub_compensated = message_filters.Subscriber(self, CompressedImage, '/image_compensated/compressed')
sub_white_mask = message_filters.Subscriber(self, CompressedImage,
'/image_compensated/white_mask/compressed')
sub_yellow_mask = message_filters.Subscriber(self, CompressedImage,
'/image_compensated/yellow_mask/compressed')

self.time_synchronizer = message_filters.ApproximateTimeSynchronizer(
    [sub_compensated, sub_white_mask, sub_yellow_mask],
    queue_size=10,
    slop=0.1
)
self.time_synchronizer.registerCallback(self.sync_callback)
```

- 3개 모두 같은 이미지 시점에서 처리된 결과
  따라서 **한 프레임 기준으로 묶여야 의미 있는 조합**
  ➡️ message_filters.ApproximateTimeSynchronizer()로 동기화 설정 및 sync_callback() 내에서 한꺼번에 처리

```
def sync_callback(self, msg_compensated, msg_white, msg_yellow):
  # 이미지 동기화 후 디코딩
        # 1. 이미지 디코딩
    img_compensated = self.cvBridge.compressed_imgmsg_to_cv2(msg_compensated)
    mask_white = self.cvBridge.compressed_imgmsg_to_cv2(msg_white)
    mask_yellow = self.cvBridge.compressed_imgmsg_to_cv2(msg_yellow)
        ###  3채널 -> 1채널 이미지 보장
    if len(mask_white.shape) == 3:
        mask_white = cv2.cvtColor(mask_white, cv2.COLOR_BGR2GRAY)
    if len(mask_yellow.shape) == 3:
        mask_yellow = cv2.cvtColor(mask_yellow, cv2.COLOR_BGR2GRAY)


        # 2.ROI(BEV영역) 처리 / 3.BEV 변환 / 4.차선 피팅 / 5.시각화 및 최종 발행.
```

- Yellow_mask를 예시로 들어서
  msg_yellow는 동기화된 메시지, 최근 도착한 메시지 중 timestamp가 다른 2개 이미지와 일치된 것을 선택.

  해당 메시지는 아직 CompressedImage 타입,
  내부적으로 .data 필드에 JPEG/PNG 바이너리 이미지

  따라서 이미지 디코딩 진행 후 BEV, 차선 피팅 등 적용

라인 검출 – 차선 피팅

호출부

```
left_fitx, self.left_fit = self.fit_from_lines(self.left_fit, warped_white_mask)  # w
right_fitx, self.right_fit = self.fit_from_lines(self.right_fit, warped_yellow_mask)
```

라인 피팅

```
def fit_from_lines(self, lane_fit, image):
        """이전 차선 위치 주변에서 새로운 차선을 찾습니다."""
        nonzero = image.nonzero()
        nonzeroy, nonzerox = np.array(nonzero[0]), np.array(nonzero[1])
        margin = 100

        lane_inds = ((nonzerox > (lane_fit[0] * (nonzeroy ** 2) + lane_fit[1] * nonzeroy + lane_fit[2] - margin)) &
                     (nonzerox < (lane_fit[0] * (nonzeroy ** 2) + lane_fit[1] * nonzeroy + lane_fit[2] + margin)))

        x, y = nonzerox[lane_inds], nonzeroy[lane_inds]

        try:
            new_fit = np.polyfit(y, x, 2)
        except TypeError:
            new_fit = lane_fit

        ploty = np.linspace(0, image.shape[0] - 1, image.shape[0])
        fitted_x = new_fit[0] * ploty ** 2 + new_fit[1] * ploty + new_fit[2]

        return fitted_x, new_fit
```

차선 곡선 피팅 (fit_from_lines)

- 마스크 이미지에서 비어 있지 않은 픽셀 추출
- 이전 차선 근처에 있는 픽셀만 선택 (±margin)
- np.polyfit()으로 2차 곡선 피팅 수행
- 새로운 차선 곡선의 x좌표 생성 → 시각화 및 중심 계산에 사용

04 프로젝트 수행 경과

▶ 라인 주행 – control_lane

주행 판단

```python
def callback_lane_state(self, msg):
    # 상태 전이
    self.prev_lane_state = self.curr_lane_state
    self.curr_lane_state = msg.data


    if self.prev_lane_state in [1, 3] and self.curr_lane_state == 0:
        self.get_logger().warn(f"Lost lane after only {self.prev_lane_state} side. Using last valid center.")
        self.hold_center_temporarily()
    elif self.prev_lane_state == 2 and self.curr_lane_state == 0:
        self.get_logger().warn("Both lanes lost. Stopping.")
        self.stop_vehicle()
    elif self.prev_lane_state == 0 and self.curr_lane_state == 2:
        self.get_logger().info("Recovered lane after losing both lanes. Resuming drive.")
        self.state = self.STATE_RUN

def hold_center_temporarily(self):
    # 이전 center 값으로 약한 유지 조향
    error = self.last_valid_center - 640
    Kp = 0.002
    twist = Twist()
    twist.linear.x = 0.03
    twist.angular.z = -Kp * error
    self.pub_cmd_vel.publish(twist)
```

/lane_state 정의

2	양쪽 차선 인식됨
1	왼쪽만 인식됨
3	오른쪽만 인식됨
0	둘 다 없음

1 -> 0	이탈 판단, 기존 주행 경로로 편향
3 -> 0	이탈 판단, 기존 주행 경로로 편향
2 -> 0	정지 (후진 적용 예정)

프로젝트 수행 경과

라인 주행 – control_lane

초기값 설정

```
# 상태 변수 초기화
### 상태
# STATE_RUN = 0
# STATE_STOP = 1

self.state = self.STATE_RUN
self.prev_lane_state = 2  # 초기에는 정상 주행 상태라고 가정
self.curr_lane_state = 2
self.last_valid_center = 640.0  # 마지막 유효한 center 값 저장
```

주행

```
def callback_follow_lane(self, desired_center):
    # lane_state에 의해 주행이 방해되면 아무 동작도 하지 않음
    if self.state != self.STATE_RUN:
        return

    center = desired_center.data
    self.last_valid_center = center  # 마지막 유효한 center 값 저장
    error = (center - 640) * 0.5

    Kp = 0.0025
    Kd = 0.007

    angular_z = Kp * error + Kd * (error - self.last_error)
    self.last_error = error

    twist = Twist()
    # 선형 속도: error에 따라 속도를 조정 (최대 0.05 제한)
    twist.linear.x = min(self.MAX_VEL * (max(1 - abs(error) / 640, 0) ** 2.2), 0.025)
    twist.angular.z = -max(angular_z, -0.3) if angular_z < 0 else -min(angular_z, 0.3)
    self.pub_cmd_vel.publish(twist)
```

주행 제어 연계

프로젝트 수행 경과

터틀봇 제어

```
if self.state != self.STATE_RUN:
    return

center = desired_center.data
error = (center - 640) * 0.5

Kp = 0.0025
Kd = 0.007

angular_z = Kp * error + Kd * (error - self.last_error)
self.last_error = error


twist = Twist()
# Linear velocity: adjust speed based on error (maximum 0.05 limit)
twist.linear.x = min(self.MAX_VEL * (max(1 - abs(error) / 640, 0) ** 2.2), 0.02)
twist.angular.z = -max(angular_z, -0.3) if angular_z < 0 else -min(angular_z, 0.3)
self.pub_cmd_vel.publish(twist)
```

중심 오차 기반 PD제어

- /lane_center 기준으로 중심 오차 계산
- PD 제어로 회전 속도(angular.z) 결정
- 오차 기반으로 전진 속도(linear.x) 조절
- /cmd_vel로 속도 명령 전송

※오차가 커지면?
(차선 중심과 화면중심 사이가 멀어지면?)

→회전 속도(angular.z)가 커져 방향을 더 많이 틀고
전진 속도(linear.x)는 느려져 천천히 이동하며 조향을 보정함

프로젝트 수행 경과

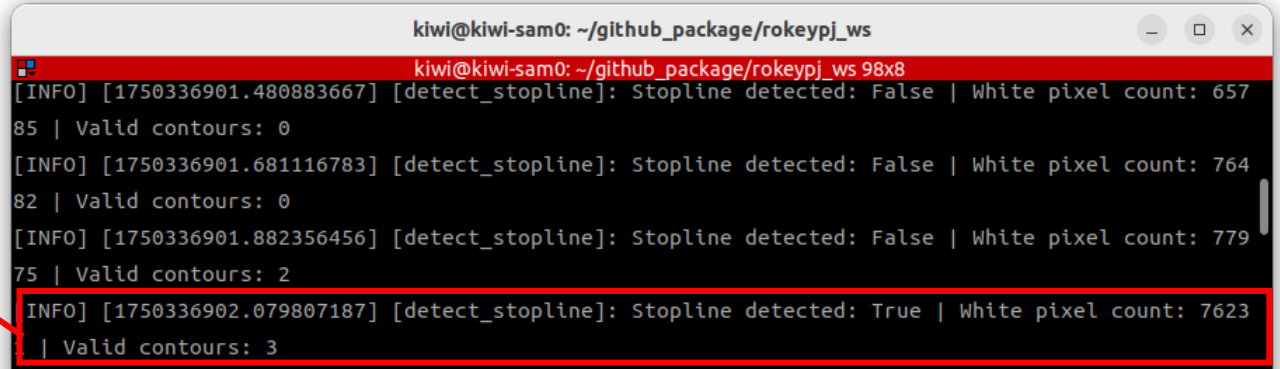
터틀봇 제어

```
# Grayscale 변환 + 이진화
gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
_, binary = cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY)

# 전체 흰색 픽셀 수 계산
white_pixel_count = cv2.countNonZero(binary)

# 윤곽선 검출
contours, _ = cv2.findContours(binary, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# 유효한 수직 띠 개수 세기
valid_contours = 0
for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)
    if h > 20 and w > 5 and h > w:
        valid_contours += 1
```

횡단보도 검출

- 조건1. 흰색 픽셀 수 > 50000
- 조건2. 수직 띠 개수 > = 3

조건 만족
→검출 토픽 발행



04 프로젝트 수행 경과

- Aruco marker 인식

```
# ArUco 디텍터 생성
detector = cv2.aruco.ArucoDetector(self.aruco_dict,
parameters)

# 그레이스케일 변환
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# 마커 감지
corners, ids, _ = detector.detectMarkers(gray)
```

인식

- 흑백 이미지로 변환하여 마커 인식률 향상
- Cv2.aruco.ArucoDetector()로 감지기 생성
- detectMarkers() 함수로 마커 감지

- 감지한 내용을 토대로 마커 인식
  corners : 마커의 꼭짓점 좌표(4개)
  ids : 감지한 마커 ID 리스트

프로젝트 수행 경과

- Aruco marker 인식

```python
# 마커의 4개 코너 좌표를 객체 포인트로 사용
object_points = np.array([
    [0, 0, 0],
    [self.marker_size, 0, 0],
    [self.marker_size, self.marker_size, 0],
    [0, self.marker_size, 0]
], dtype=np.float32)

# 마커 위치 추정
success, rvec, tvec = cv2.solvePnP(
    object_points,
    corners[i],
    self.camera_matrix,
    self.dist_coeffs
)
 # 거리 계산 (정확한 3D 거리)
 distance = np.linalg.norm(tvec)
```

위치 추정

<입력 값>
- Object_points : 마커의 실제 좌표
- Corners[i] : 이미지에서 검출된 2D 좌표
- Camera_matrix, dist_coeffs : 카메라 내부 파라미터

<출력 값>
- rvec : 회전 벡터 (마커 자세)
- tvec : 이동 벡터 (카메라 → 마커까지의 3D 위치)

<거리 계산>
- tvec의 크기 → 마커까지의 3D 거리
- norm 함수를 사용하여 거리 계산

프로젝트 수행 경과

- Aruco marker 인식

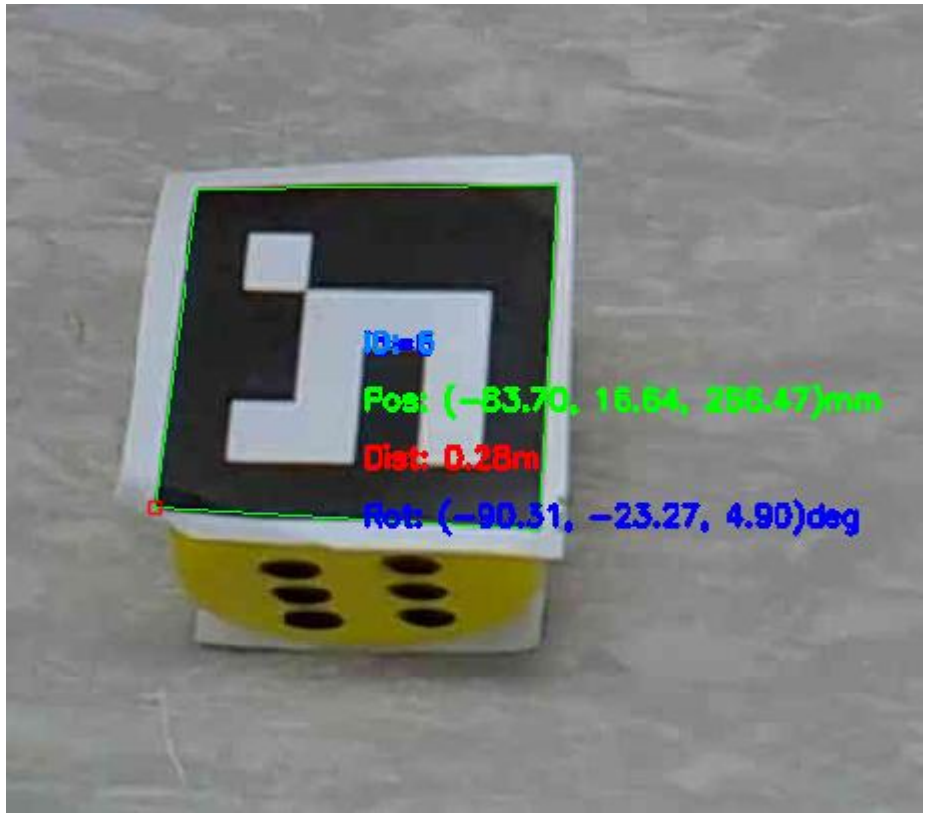
```
rotation_matrix, _ = cv2.Rodrigues(rvec)
yaw = np.arctan2(rotation_matrix[1,0],
rotation_matrix[0,0]) * 180 / np.pi
pitch = np.arctan2(-rotation_matrix[2,0], sy)
* 180 / np.pi
roll = np.arctan2(rotation_matrix[2,1],
rotation_matrix[2,2]) * 180 / np.pi
```

자세(roll, pitch, yaw) 계산

- rvec -> Rodrigues 변환을 통해 회전 행렬로 변환
- 회전 행렬로부터 오일러 각(yaw/pitch/roll)을 계산
- 이 각도들은 마커가 카메라에 대해
  어떤 방향으로 회전했는지 표현

프로젝트 수행 경과

- Aruco marker 인식 결과



04 프로젝트 수행 경과

- pick_and_place

```
def marker_callback(self, msg):
    if self.processing or len(msg.markers) == 0:
        return

    marker = msg.markers[0]
    self.marker_pose = marker.pose.pose

    self.log_info(f"마커 위치 수신: x={self.marker_pose.position.
    x:.3f}, y={self.marker_pose.position.y:.3f}, z={self.
    marker_pose.position.z:.3f}")

    self.processing = True
    self.execute_pick_and_place()
```

작업 시작

- detected_makers 토픽 구독
- 가장 먼저 인식된 마커 1개 선택
- 인식된 마커 좌표 값 저장
- pick_and_place 작업 시퀀스 시작

프로젝트 수행 경과

- pick_and_place

```python
def send_moveit_request(self, cmd, target, callback=None):
    req = MoveitControl.Request()
    req.cmd = cmd
    if cmd == 0:
        req.waypoints = target
    else:
        req.posename = target
    future = self.moveit_client.call_async(req)
    future.add_done_callback(...)
```

MoveIt 서비스 요청

- cmd 값을 통해 조작 방식 지정
  (1 : 로봇팔, 2: 그리퍼)
- 비동기 서비스 호출 방식 사용
- 각 스텝 완료 후 콜백으로 다음 작업 실행

04 프로젝트 수행 경과

- pick_and_place

작업 순서

1. 그리퍼 열기
2. Aroco mark 위로 이동
3. 집을 수 있는 위치로 이동
4. 그리퍼 닫기
5. 위로 올리기
6. 우측 컨베이어 상단 위치로 이동
7. 우측 컨베이어에 적재 위치로 이동
8. 그리퍼 열기
9. 위로 올리기
10. Lane tracking 위치로 복귀

K-Digital Training

05 자체 평가 의견

- 자체 평가

사전 기획 대비 완성도 : 9/10

잘한 부분과 아쉬운 점

- 역할 분배를 통해 차선 인식 및 주행 관련 핵심 기능 구현 완료
- 계획했던 일부 태스크 미구현

개선점 및 보완할 점

- 작업 수행 안정성 및 신뢰성 향상
- 사용자 인터페이스 추가
- 실제 환경 시나리오 기반 테스트

경험 및 성과

- **실환경 대응 능력 확보 :** 실제 환경에서 발생하는 비전 인식 오류 및 제어 이슈들을 식별하고, 이를 해결하기 위한 여러 방법을 실험적으로 적용함.