2026. 1. 5

자율주행 경진대회

캡스톤 디자인 2 중간발표

LIMO Pro

문정석, 박건우

CONTENTS

코스 별 알고리즘간의 전환 알고리즘 부재 및 코스 알고리즘의 미완성

코스 각각의 미션 수행 불가능 + **코스 간의 연결 주행 불가능**

각 코스 별 연결 알고리즘 부재	인식 및 주행 알고리즘 문제	잘못된 알고리즘 parameter
• Launch 파일 독립적으로 미션 수행 가능	• 커브 구간에서 Line detecting 알고리즘 불안정 • 신호등 인식 알고리즘 수정 필요	• 하드 코딩된 주차 알고리즘 • 보행자, 로타리, 장애물 미션 parameter 수정 필요

전체 랩타임 최적화는 모든 코스가 안정적으로 동작한 이후에 가능
중간 단계에서는 랩타임 단축이 아닌 안정적인 주행을 목표

배경

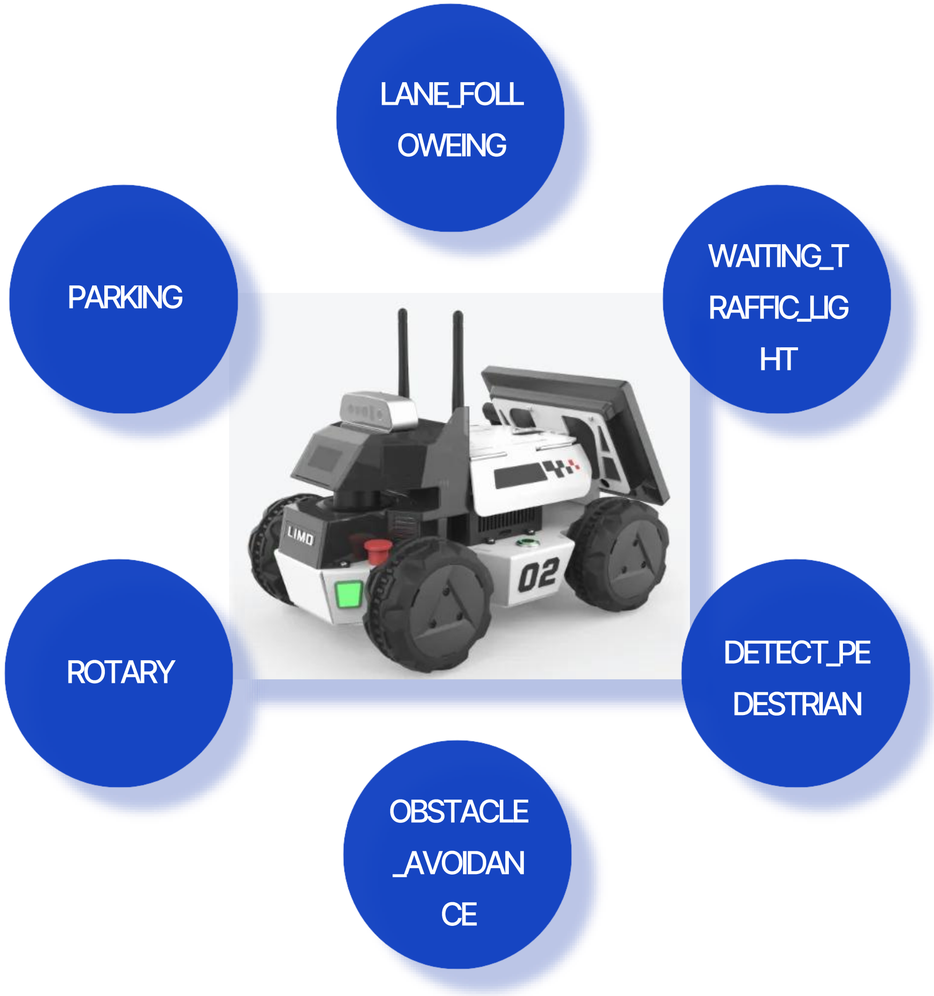
- 개별 코스의 신뢰성 미확보 시 연결 알고리즘 검증 불가
- 랩타임 최적화를 위한 반복 가능 안정성 확보

목표

- 각 코스 별 미션 수행 가능한 알고리즘 완성
- 코스 간 전환이 가능한 연결 알고리즘(State 기반 구조) 구현
- 단일 실행으로 전체 주행이 가능한 통합 실행 구조 확보

State 전환 구조

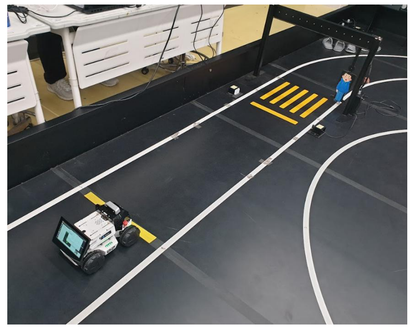
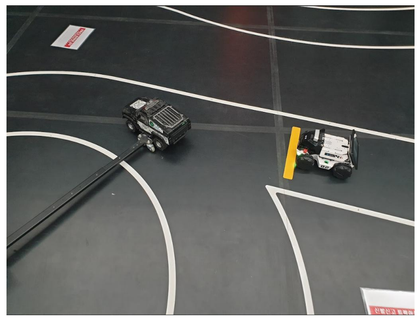
State: 현재 주행 상황에 따라 차량이 수행해야 할 하나의 명확한 행동 단위

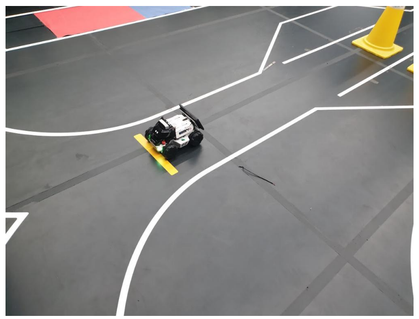


각 State는 하나의 명확한 목적만 수행

알고리즘 수정 용이

State 전환 조건 선정: 노란 선 인식



- 각 구간 시작 전 노란 선 존재
- 센서 이벤트 기반 state 전환

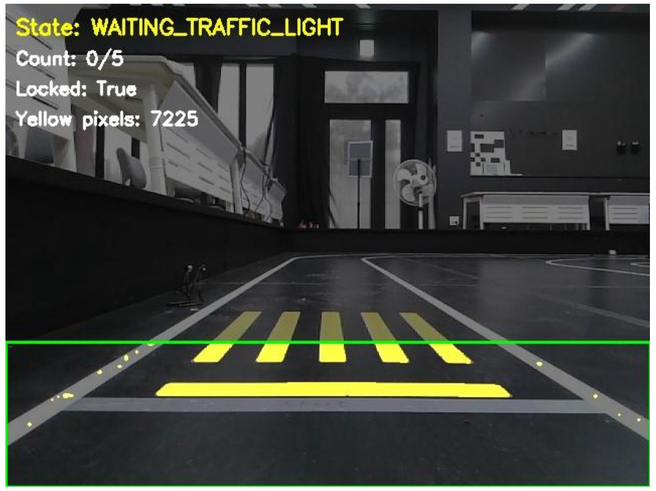
성공

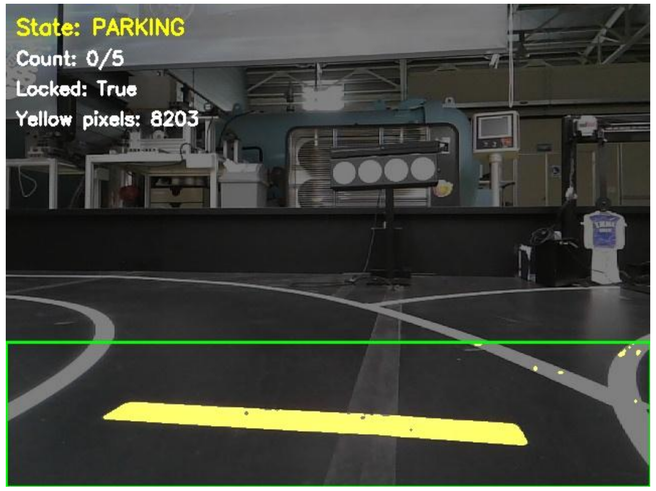
이상적인 주행 가정하에 모든 구간 100%

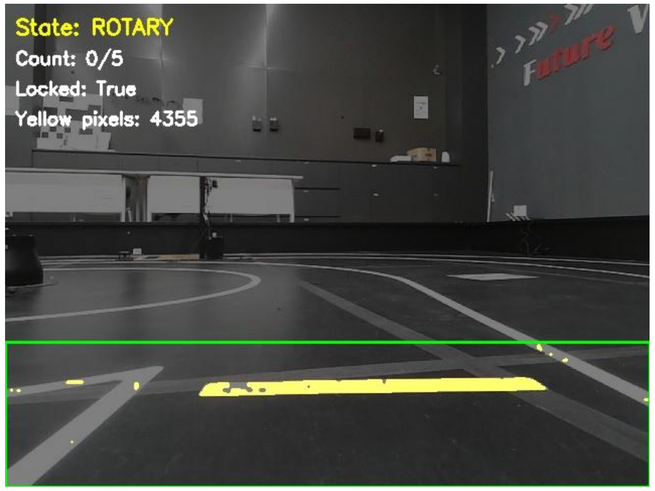
실패

차선을 완전히 벗어나서 주행하는 경우
다른 미션의 노란색선들이 검출될 수 있음

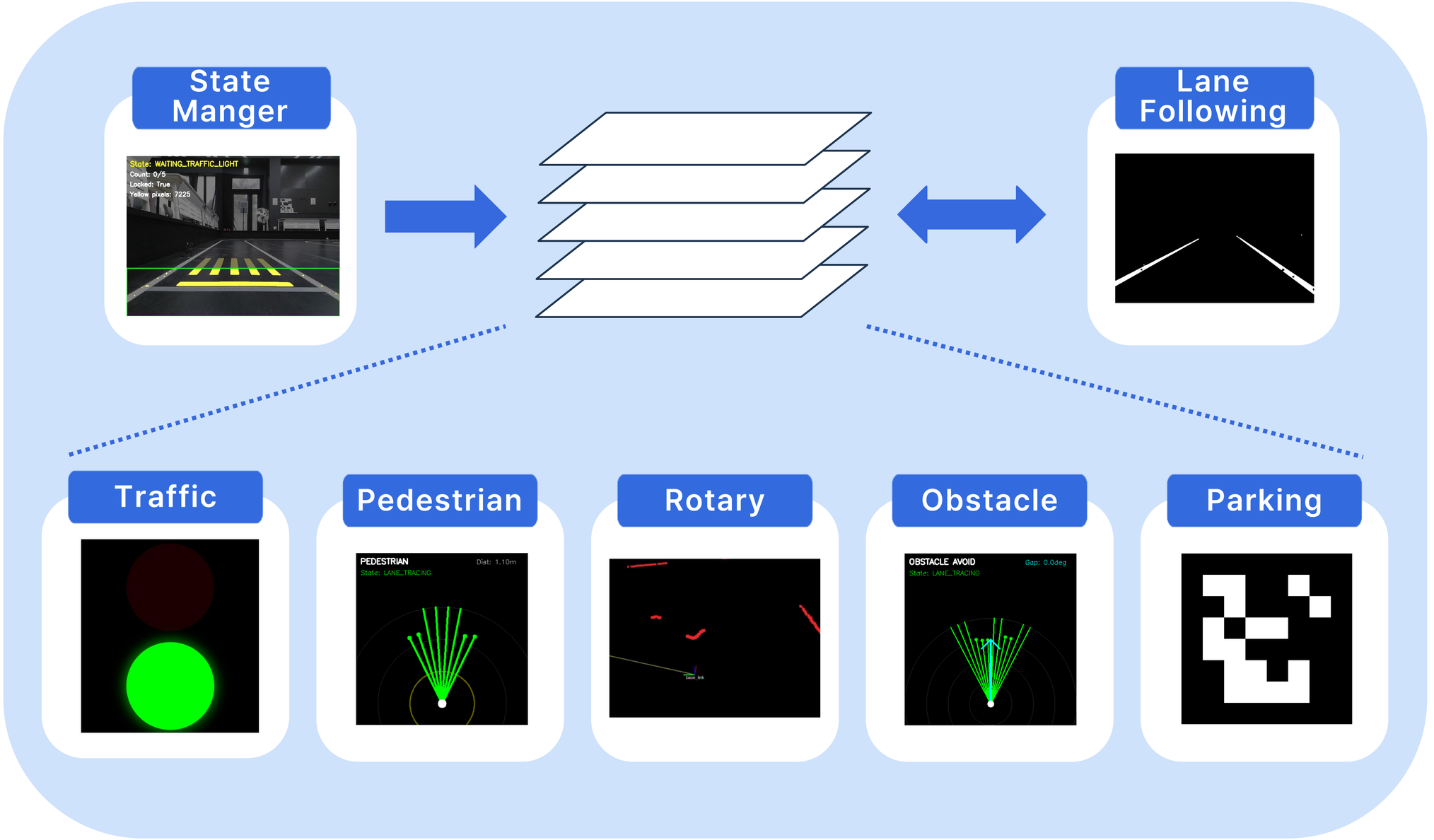
State 전환 구조



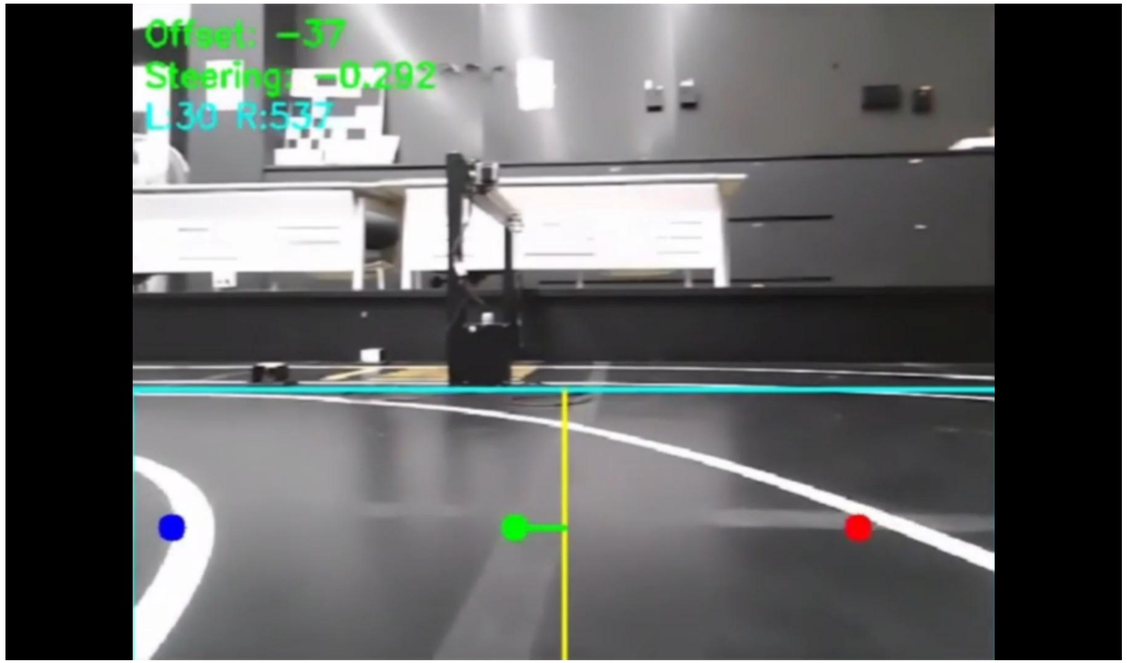




전체 주행 시스템 구조



Lane Tracing 개선



< 기존의 알고리즘>

Before

- ROI를 설정했음에도 불구하고 급격히 꺾이는 곡선의 경로에서 차선이 아닌 곳까지 인식을 하게 되어 전방의 장애물들 또한 차선을 탐지하는데 영향을 미침.
- 횡방향 오차에 대해서만 고려하기에 곡선 구간에서 횡방향 오차를 맞추기 위해 차선을 벗어나면서 맞추는 경우가 생김.
- 차선이 하나만 보이는 경우에 대한 대비가 안되어있음.

After

- Raw 이미지에서 ROI를 씌워서 차선을 탐지하는 것이 아닌 Bird Eye View로 변환을 한 뒤에 차선을 탐지하여 기존 알고리즘 대비 오검출이 훨씬 적어짐
- 횡방향 오차뿐 아니라 헤딩 오차 또한 고려하도록 하여서 차선을 벗어나지 않도록 함
- 차선이 하나만 보이는 경우 차선의 곡률에 따라 오프셋을 부여함

Lane Tracing 개선

Algorithm : Lane Detection

Input : camera Image I
Output : Follow path

```
1   : Image_white <-  white_mask(I)
2   : Image_gradient <- scharr_gradient(Image_white)
3   : Image_bev <- BirdEyeView(Image_gradient)
4   : C <- sliding window search(Image_bev)
5   : L <- poly fit (C)

6   : if L[0] is not None and L[1] is not None
7   :     LC = ( L[0] + L[1] )/ 2
8   : elif L[0] is not None or L[1] is not None
9   :     LC = L + offset
```

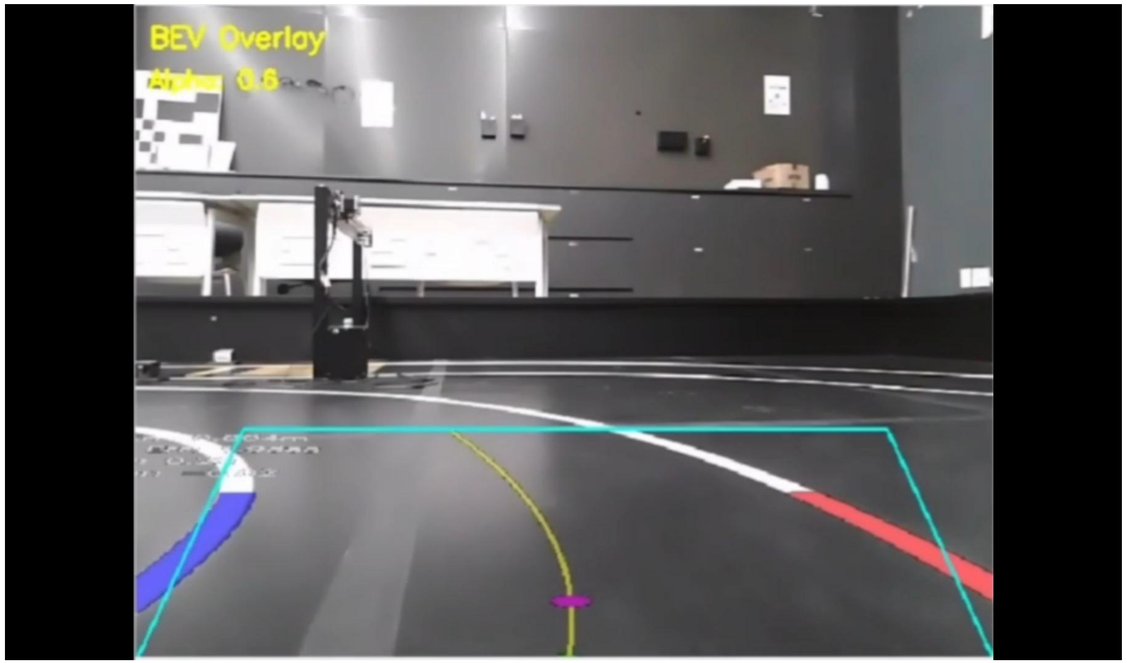
Before

- ROI를 설정했음에도 불구하고 급격히 꺾이는 곡선의 경로에서 차선이 아닌 곳까지 인식을 하게 되어 전방의 장애물들 또한 차선을 탐지하는데 영향을 미침.
- 횡방향 오차에 대해서만 고려하기에 곡선 구간에서 횡방향 오차를 맞추기 위해 차선을 벗어나면서 맞추는 경우가 생김.
- 차선이 하나만 보이는 경우에 대한 대비가 안되어있음.

After

- Raw 이미지에서 ROI를 씌워서 차선을 탐지하는 것이 아닌 Bird Eye View로 변환을 한 뒤에 차선을 탐지하여 기존 알고리즘 대비 오검출이 훨씬 적어짐
- 횡방향 오차뿐 아니라 헤딩 오차 또한 고려하도록 하여서 차선을 벗어나지 않도록 함
- 차선이 하나만 보이는 경우 차선의 곡률에 따라 오프셋을 부여함

Lane Tracing 개선



< 수정된 알고리즘>

Before

- ROI를 설정했음에도 불구하고 급격히 꺾이는 곡선의 경로에서 차선이 아닌 곳까지 인식을 하게 되어 전방의 장애물들 또한 차선을 탐지하는데 영향을 미침.
- 횡방향 오차에 대해서만 고려하기에 곡선 구간에서 횡방향 오차를 맞추기 위해 차선을 벗어나면서 맞추는 경우가 생김.
- 차선이 하나만 보이는 경우에 대한 대비가 안되어있음.

After

- Raw 이미지에서 ROI를 씌워서 차선을 탐지하는 것이 아닌 Bird Eye View로 변환을 한 뒤에 차선을 탐지하여 기존 알고리즘 대비 오검출이 훨씬 적어짐
- 횡방향 오차뿐 아니라 헤딩 오차 또한 고려하도록 하여서 차선을 벗어나지 않도록 함
- 차선이 하나만 보이는 경우 차선의 곡률에 따라 오프셋을 부여함

Traffic 개선

```
# Green HSV thresholds (from yaml)
green_h_low = rospy.get_param('~green_h_low', 35)
green_h_high = rospy.get_param('~green_h_high', 85)
green_s_low = rospy.get_param('~green_s_low', 80)
green_s_high = rospy.get_param('~green_s_high', 255)
green_v_low = rospy.get_param('~green_v_low', 80)
green_v_high = rospy.get_param('~green_v_high', 255)

self.GREEN_LOW = np.array([green_h_low, green_s_low, green_v_low])
self.GREEN_HIGH = np.array([green_h_high, green_s_high, green_v_high])

# Detection parameters (from yaml)
self.min_area = rospy.get_param('~min_area', 300)
self.circularity_thresh = rospy.get_param('~circularity_thresh', 0.4)

# ROI settings (from yaml)
self.roi_top = rospy.get_param('~roi_top', 100)
self.roi_bottom = rospy.get_param('~roi_bottom', 360)
self.roi_left = rospy.get_param('~roi_left', 0)
self.roi_right = rospy.get_param('~roi_right', 320)

# Brightness adjustment (from yaml)
self.brightness_factor = rospy.get_param('~brightness_factor', 0.5)
```

```
 # ⭐ 핵심: 원형 판별
 if circularity >= self.circularity_thresh:
        (x, y), radius = cv2.minEnclosingCircle(cnt)

        circles.append({
            'center': (int(x), int(y)),
            'radius': int(radius),
            'area': area,
            'circularity': circularity,
            'color': color_name
        })
```

```
def state_callback(self, msg):
    """State callback from mission controller"""
    prev_state = self.current_state
    self.current_state = msg.data
    if prev_state != self.current_state:
        rospy.loginfo(f"Traffic light node received state: {self.current_state}")
        # Reset passed flag when entering WAITING_TRAFFIC_LIGHT state
        if self.current_state == "WAITING_TRAFFIC_LIGHT":
            self.green_passed = False
            self.stop_flag = True
```

Before

State 토픽 구독하지 않음

사각형 기준 신호등 식별

cmd_vel 직접 제어

Parameter 조정되지 않음

After

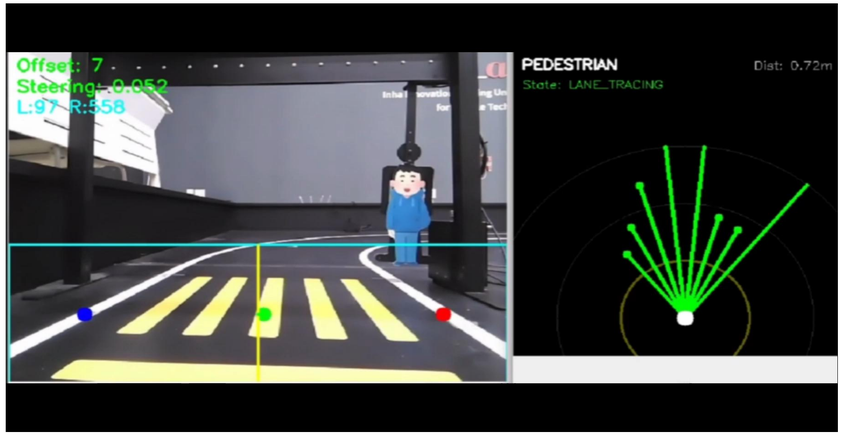
State 토픽 구독

원형 기준 신호등 식별

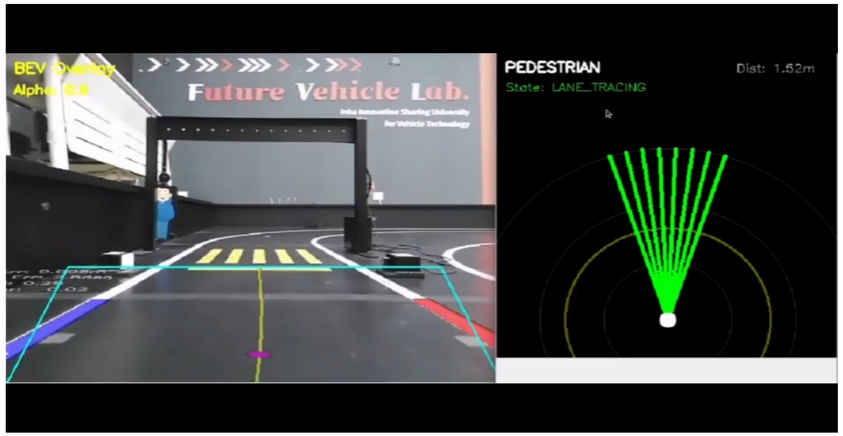
Stop flag만 발행

Parameter 조정

Detect Pedestrian 개선



< 기존의 알고리즘>



< 수정된 알고리즘>

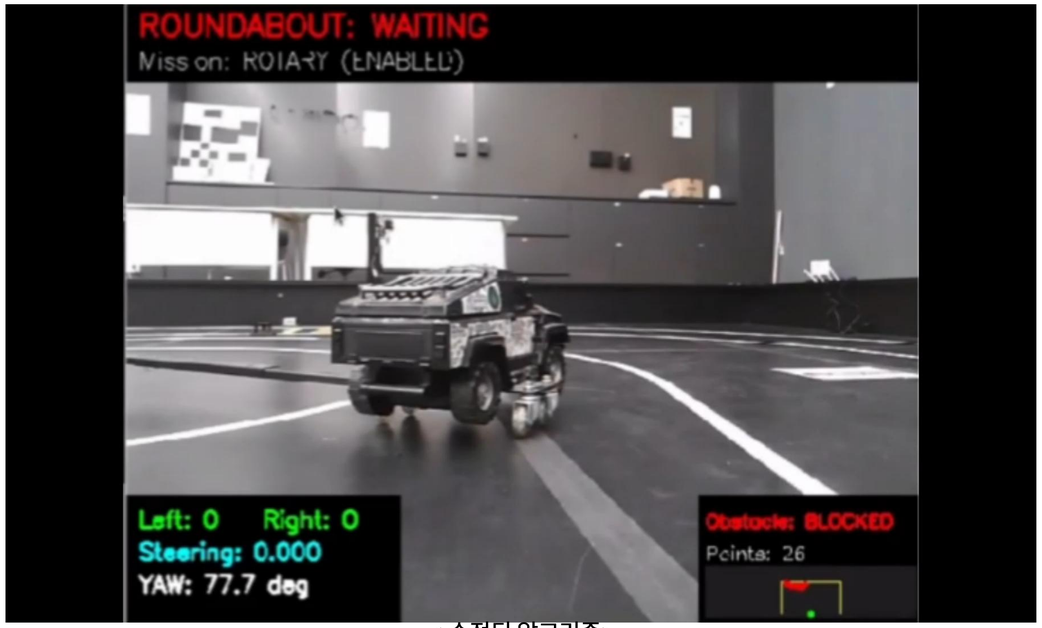
Before

- 너무 넓은 범위의 감지로 인해, 주행경로가 아닌 곳까지 감지하는 경우가 생겨 자주 멈추는 현상이 있음

After

- 라이다 스캔값을 쓰는 범위 및 거리를 소폭 감소시켜 보행자탐지는 유지하되 평상시에 주행할 때 멈추는 현상을 없앨 수 있었음

Rotary 개선



< 수정된 알고리즘>

Before

- 원형 곡선 커브 구간에서 안정적인 주행을 하지 못함
- 원형 커브 구간에 있는 차량이 아닌 앞의 보행자 구간의 장애물을 인식하여 주행 중에 멈추는 현상이 존재

After

- 파라미터 수정 후 전방의 다른 장애물을 인식하더라도 완전히 멈추지 않음

Obstacle_Avoidance_node 개선

```
# Parameters (from yaml/cfg)
self.scan_angle = rospy.get_param('~scan_angle', 60.0)  # 더 넓은 스캔 각도
self.safe_distance = rospy.get_param('~safe_distance', 1.4)  # 1.4m에서 회피 시작
self.stop_distance = rospy.get_param('~stop_distance', 0.4)  # 0.4m에서 정지
self.num_sectors = rospy.get_param('~num_sectors', 16)  # 더 세밀한 섹터

# Lane speed from lane_detect_node
self.lane_speed = 0.25  # 기본값, lane_detect_node에서 수신
self.avoid_speed = rospy.get_param('~avoid_speed', 0.2)  # 회피 전용 속도
self.max_steering = rospy.get_param('~max_steering', 0.5)
self.steering_gain = rospy.get_param('~steering_gain', 0.02)
```

```
for gap_start_idx, gap_end_idx in gaps:
    gap_width = gap_end_idx - gap_start_idx + 1

    # gap 내 섹터들의 평균 거리
    gap_distances = self.sector_distances[gap_start_idx:gap_end_idx + 1]
    avg_distance = np.mean(gap_distances)
    min_distance = np.min(gap_distances)

    # gap 중심이 중앙에 가까울수록 높은 점수
    gap_center = (gap_start_idx + gap_end_idx) / 2.0
    center_penalty = abs(gap_center - center_sector) * 0.5  # 중앙에서 멀수록 감점

    # 점수 계산: 너비 + 거리 보너스 - 중앙 패널티
    # min_distance가 safe_distance보다 충분히 커야 좋은 gap
    score = gap_width * 2 + min_distance * 3 + avg_distance - center_penalty

    # 최소 거리가 너무 짧으면 해당 gap 제외
    if min_distance < self.safe_distance * 0.8:
        score -= 10  # 큰 패널티

    if score > best_score:
        best_score = score
        best_gap = (gap_start_idx, gap_end_idx)
```

```
def control_loop(self, event):
    if self.current_state != "OBSTACLE_AVOIDANCE":
        return
    cmd = Twist()
    state = "IDLE"
```

Before

State 토픽 구독하지 않음

가장 넓은 gap 선택

cmd_vel 직접 복귀

Parameter 조정되지 않음

After

State 토픽 구독

Gap 너비, 최소거리, 평균거리, 중앙 근접도 조합 -> gap 선택

State 토픽으로 복귀 지시

Parameter 조정

Parking_node 개선

```
self.speed_slow = rospy.get_param('~speed_slow', 0.15)          # 저속
self.speed_reverse = rospy.get_param('~speed_reverse', -0.13)   # 후진 속도
self.steering_angle = rospy.get_param('~steering_angle', 0.8)   # 조향 각도

self.forward_time = rospy.get_param('~forward_time', 1.5)            # 주차 구역 지나 전진 시간 (초)
self.reverse_right_time = rospy.get_param('~reverse_right_time', 3.2)  # 후진 우회전 시간 (초)
self.reverse_left_time = rospy.get_param('~reverse_left_time', 3.2)    # 후진 좌회전 시간 (초)
self.final_forward_time = rospy.get_param('~final_forward_time', 1.0)  # 마지막 전진 직진 시간 (초)

# 트리거 거리 (미터) - 이 거리 이내일 때 주차 시작
self.trigger_distance = rospy.get_param('~trigger_distance', 0.5)  # 0.5m
self.target_marker_id = rospy.get_param('~target_marker_id', 0)    # 마커 ID 0

def mission_state_callback(self, msg):
    """미션 상태 콜백"""
    prev_state = self.mission_state
    self.mission_state = msg.data
    if prev_state != self.mission_state:
        rospy.loginfo(f"Parking node received mission state: {self.mission_state}")
```

Before

State 토픽 구독하지 않음

항상 /cmd_vel publish

Parameter 조정되지 않음

After

State 토픽 구독

IDLE 상태에서 /cmd_vel을 publish 하지 않음

Parameter 조정

구간별 성공률 분석

실험 setting

- Trial 횟수: 각 코스 당 10회
- 동일 조건 초기화

성공 기준 선정

Node	성공 기준
Line_Tracing	차량의 4바퀴 중 하나라도 나가지 않거나 차선 검출을 실패하지 않는 경우
Traffic	빨간불일 경우 stop이 풀리는 오검출이 0회
Pedestrian	보행자가 지나갈 때 멈추고 다 지나간 후에 정상적인 주행 복귀
Rotary	앞의 차량과 부딪히지 않으면서 안정적인 주행
Obstacle_Avoidance	장애물과 충돌 0회 / 회피 후 차선 복귀 및 유지
Parking	주차 시퀀스 수행 후 모든 바퀴가 주차 구간 내 위치

구간별 성공률 분석

측정 결과

Node	성공률(%)
Line_Tracing	60%
Traffic	60%
Pedestrian	100%
Rotary	70%
Obastacle_Avoidance	40%
Parking	70%

실패 원인 분석

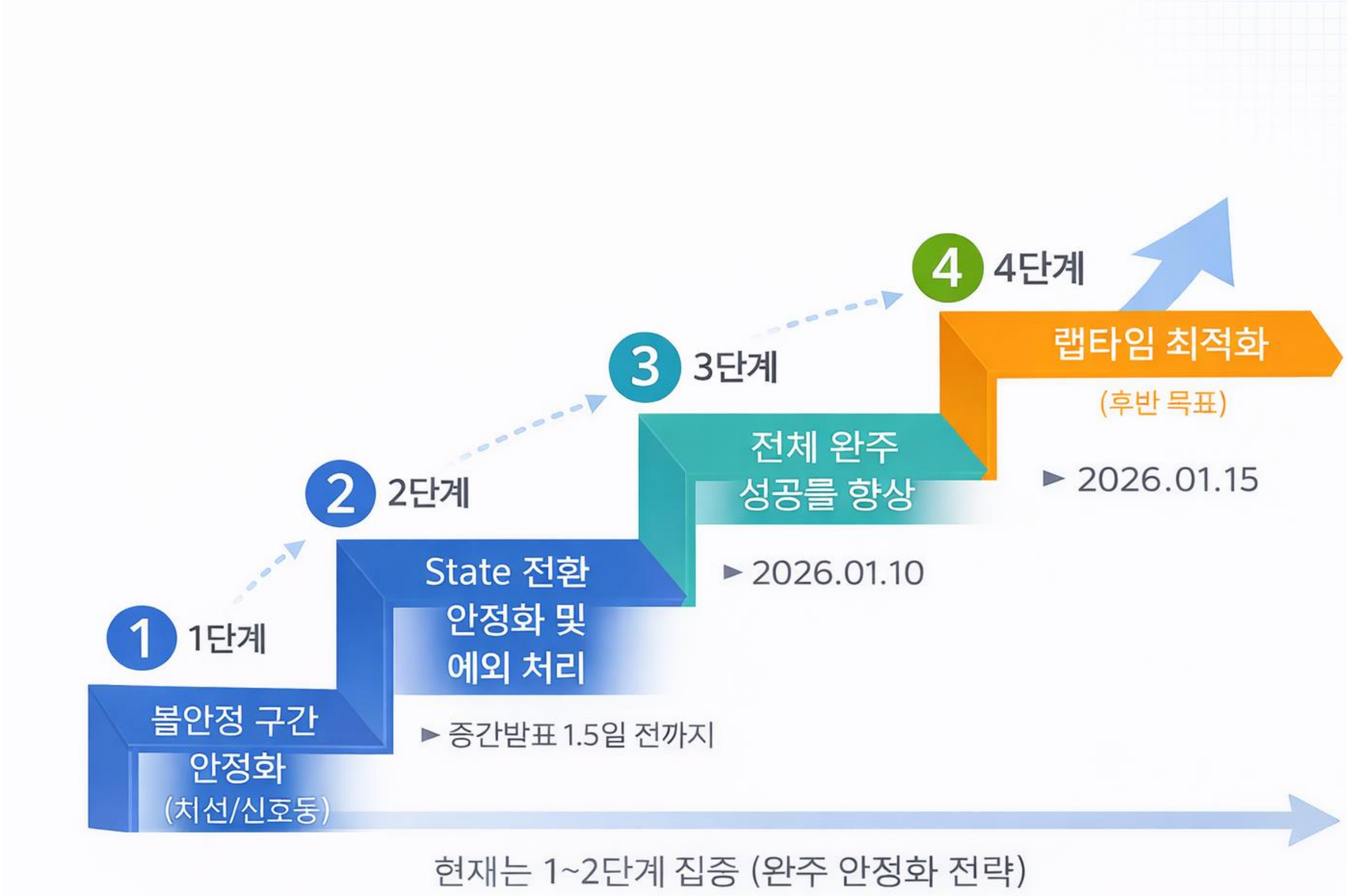
Node	주요 원인
Line_Tracing	여러 차선이 있는 경우에 대한 차선 검출 실패
Traffic	조도 변화/빛 반사로 인한 HSV 변화
Pedestrian	X
Rotary	정확하지 못한 탐지 구역 범위 설정
Obastacle_Avoidance	Safe distance, scan angle과 같은 parameter 튜닝 실패
Parking	하드코딩으로 인한 실험 재현성 부족

전체 완주 결과

Obstacle	성공률(%)
왼쪽	0%
가운데	80%
오른쪽	0%

완주 시간
3분

향후 개선 로드맵



향후 개선 로드맵

Q&A