

객체지향모델링 결과 보고서

제목: ROS 기반의 터틀봇 자율주행

제출일	2021년 12월 04일
합반 2조	김두영(20153294), 박진우(20173176), 안대현(20173217), 허세진(20194152)

© 2018 동의대학교 컴퓨터소프트웨어공학과

이 결과 보고서 양식은 ISO/IEC/IEEE 29148:2011 요구공학 국제표준과 스크럼 애자일 프로세스를 적용하여 진행된 객체지향설계 교과목의 설계 프로젝트에 맞게 테일러링한 것입니다.

<제목 차례>

1. 프로젝트 개요	4
1.1 비전	4
1.2 문제 기술	4
1.3 요구사항 목록	5
1.4 관련 기술	6
1.5 이해 당사자 요구사항	6
2. 시스템 개발	7
2.1 상태 패턴 StateMachine	7
2.2 차단바 인식	9
2.2.1 분석	9
2.2.2 설계	10
2.2.3 구현	11
2.2.4 테스트	13
2.3 정지선 인식 및 정지	13
2.3.1 분석	13
2.3.2 설계	15
2.3.3 구현	16
2.3.4 테스트	19
2.4 차선 인식 및 추적	19
2.4.1 분석	19
2.4.2 설계	21
2.4.3 구현	22
2.4.4 테스트	27
2.5 정지 표지판 인식 및 정지	28
2.5.1 분석	28
2.5.2 설계	29
2.5.3 구현	30
2.5.4 테스트	32
2.6 장애물 인식 및 정지	33
2.6.1 분석	33
2.6.2 설계	34
2.6.3 구현	35
2.6.4 테스트	36

3. 프로젝트 결과	37
3.1 프로젝트 완성도	37
3.2 1차선/2차선 주행 완성도	37
3.3 일정 계획 대비 달성도	40
3.4 역할 수행 및 협업 도구 사용	40
3.5 소스 코드 버전 제어 도구 사용	42
3.6 설계 구성요소	43
3.7 현실적 제한조건	43

1. 프로젝트 개요

1.1 비전

자율주행기술은 다가오는 미래에 가장 기대되는 기술 중 하나이다. 자율주행 기술은 '인간이 운전하는 것보다 안전한 운전'으로 차량 사고를 줄여주어 운행의 안정성을 높여준다. 또한, 전동화를 통해 차의 부품 수가 줄어들어 고장률도 줄어들고, 호텔 산업, 승차 공유 산업, 항공 산업, 부동산 산업 등 우리 생활 전반에 영향을 미칠 것이다.

본 프로젝트는 다양한 코스에서도 안정적인 자율 주행이 가능한 ROS 기반의 자율 주행 터틀봇 시뮬레이터를 제공한다.

1.2 문제 기술



[그림 1] 자율주행 가제보 시뮬레이션 월드

실제 차량을 이용하여 주행 시험장을 실험하기에는 비용, 장소, 시간 모든게 부족하므로 ROS와 가제보 시뮬레이터를 통해 자율 주행을 가상으로 테스트하여 실제 주행 테스트에서 발생할 수 있는 위험 상황을 막고, 경제적 비용을 절감할 수 있다. 또한, 오류 발생 혹은 성능 증진을 위해 기능을 수정하거나 센서를 추가하는 것 또한 비교적 자유롭기 때문에 원하는 상황에 맞는 여러 가지 기능들이 오픈소스로 공개되어 있어서 테스트를 하는데 유리하다.

1.3 요구사항 목록

연번	우선순위	설 명
SFR-101	2	터틀봇은 자동으로 출발할 수 있다.
SFR-102	10	터틀봇은 자동으로 방향 전환이 가능하다.
SFR-103	3	터틀봇의 최대 속도는 1m/s을 초과해서는 안된다.
SFR-104	5	중앙선을 넘지 않는다.
SFR-105	9	정해진 차로를 유지한다.
SFR-201	7	정지선 앞에서 3초간 정지한다
SFR-202	18	장애물이 앞에 있으면 정지한다
SFR-203	16	정지 표지판 앞에 정지한다
SFR-204	20	종료 정지선 앞에 정지한다
SFR-301	8	터틀봇은 자신의 출발 차로에 맞는 코스를 선택한다
SFR-302	12	굴절코스를 주행한다
SFR-303	13	곡선코스를 주행한다
SFR-304	14	방향전환 코스를 주행한다
SFR-305	19	평행주차 코스를 주행한다.
SFR-401	1	차단바를 인식한다.
SFR-402	4	정지선을 인식한다.
SFR-403	6	중앙선을 인식한다.
SFR-404	11	벽을 인식한다.
SFR-405	17	장애물을 인식한다.
SFR-406	15	정지 표지판을 인식한다.

[표 1] 요구 사항 목록

1.4 관련 기술

연번	특징	관련 기술
SFR-401	차단바를 인식한다.	-터틀봇의 가상 카메라로 OPENCV 로 받은 영상을 데이터에 따라 처리한다. - numpy 로 색상의 최대값 최소값을 입력받는다
SFR-402	정지선을 인식한다.	-터틀봇의 가상 카메라로 OPENCV 로 받은 영상을 데이터에 따라 처리한다.
SFR-403	중앙선을 인식한다.	-터틀봇의 가상 카메라로 OPENCV 로 받은 영상을 데이터에 따라 처리한다. - Canny알고리즘 을 이용하여 차선을 인식
SFR-405	장애물을 인식한다.	-터틀봇의 LaserScan 으로 장애물을 인식한다.
SFR-406	정지 표지판을 인식한다.	-터틀봇의 가상 카메라로 OPENCV 로 받은 영상을 데이터에 따라 처리한다.

[표 2] 관련기술

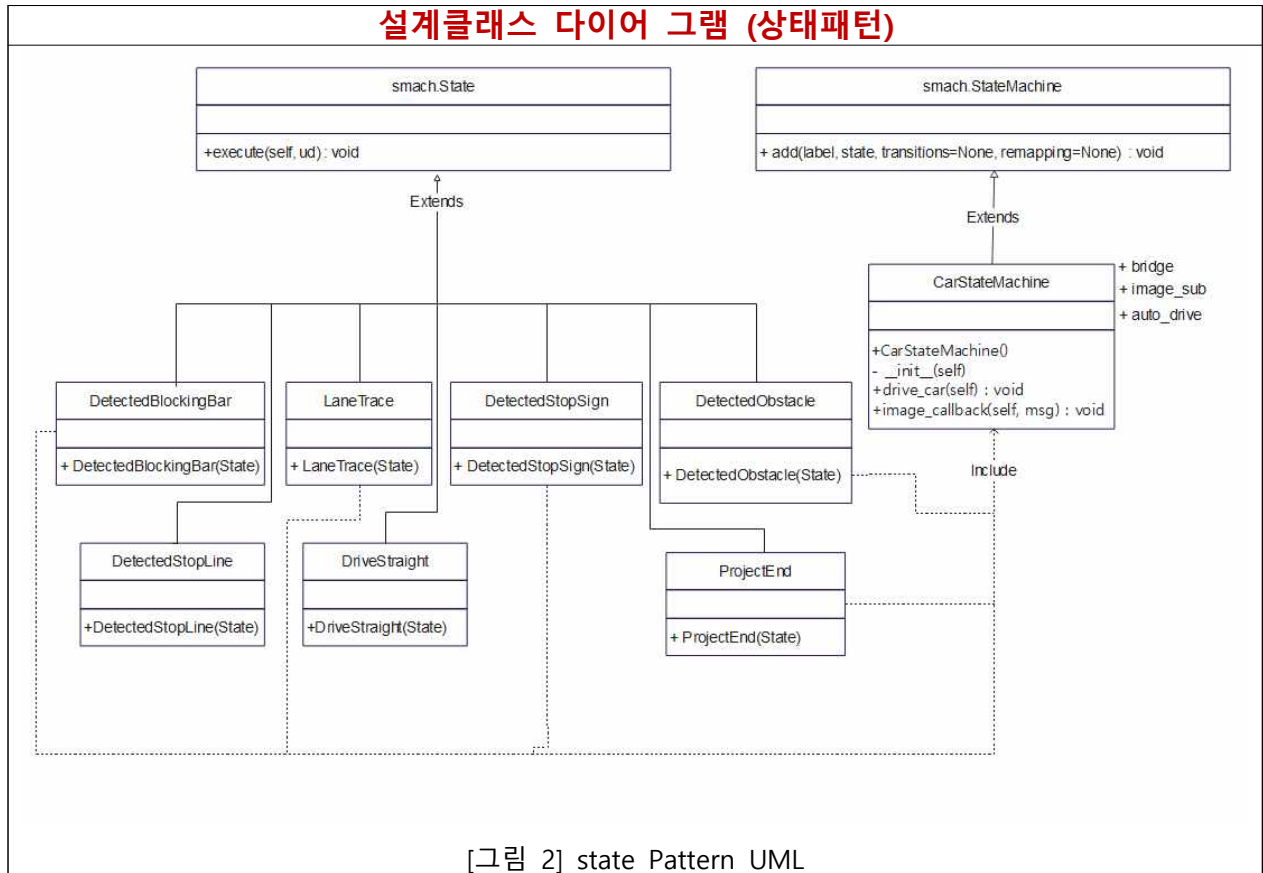
1.5 이해 당사자 요구사항

연번	이해 당사자	요구사항
St-01	서비스 기획자	- 제공한 환경에서 기획자가 요구한 상황에 맞게 터틀봇이 동작해야한다.
St-02	서비스 사용자	- 터틀봇이 자율적으로 주행해야한다. - 터틀봇이 명령이 들어오면 차선을 변경해야한다.
St-03	개발자	- 터틀봇의 동작 및 인식을 세분화해야한다. - 기획자의 요구사항을 모두 반영해야한다.

[표 3] 이해 당사자 요구사항

2. 시스템 개발

2.1 상태 패턴 StateMachine



소스

```

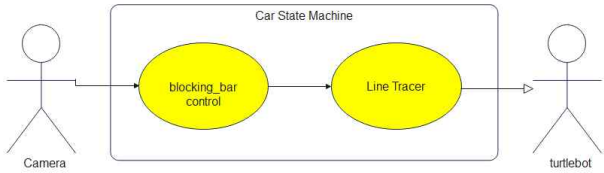
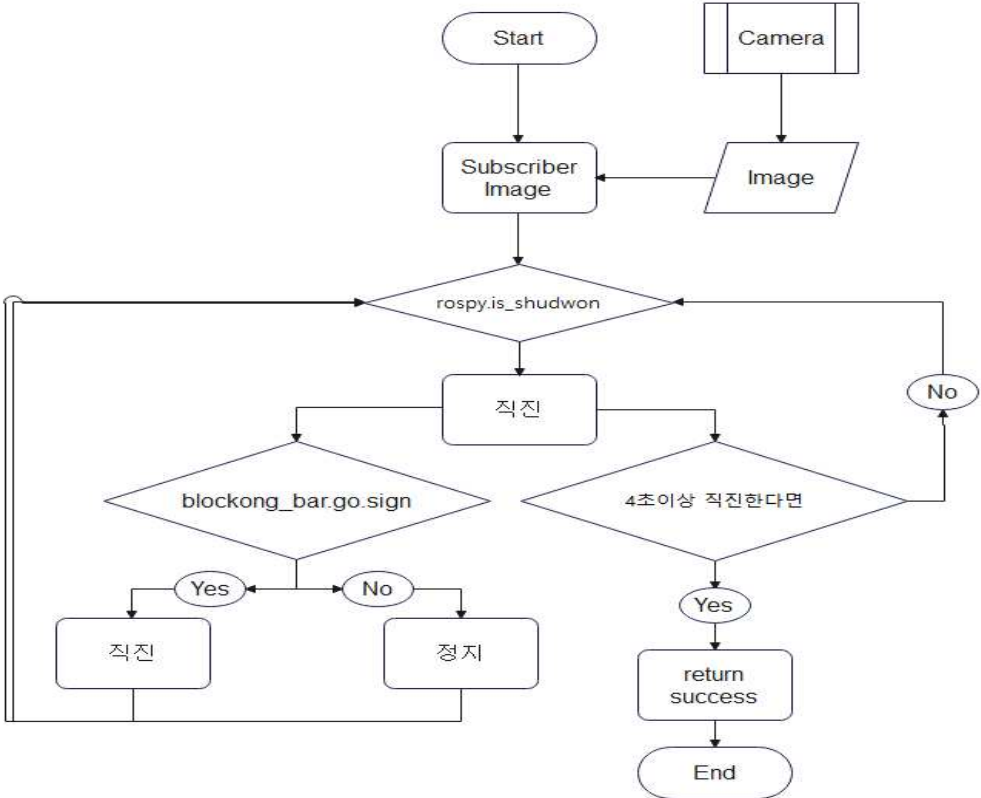
class CarStateMachine(object):
    def __init__(self):
        self.auto_drive = StateMachine(outcomes=['success'])
        self.image_sub = rospy.Subscriber('camera/rgb/image_raw',
                                           Image, self.image_callback)

        self.bridge = cv_bridge.CvBridge()
    def image_callback(self, msg):
        image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        cv2.imshow("Driving screen", image)
        cv2.waitKey(3)
    def drive_car(self):
        with self.auto_drive:
            StateMachine.add('DETECT_BLOCKING_BAR',
                             car_state.DetectedBlockingBar(),
                             transitions={'success': 'LANE_TRACE'})
            StateMachine.add('LANE_TRACE',
                             car_state.LaneTrace(),
                             transitions={'detected_stop_line': 'DETECT_STOP_LINE',
                                          'detected_obstacle': 'DETECT_OBSTACLE',
                                          'detected_stop_sign': 'DETECT_STOP_SIGN'})
            StateMachine.add('DETECT_STOP_LINE',
                             car_state.DetectedStopLine(),
                             transitions={'success': 'LANE_TRACE', '
    
```

	<pre> drive_straight': "DRIVE_STRAIGHT"}} StateMachine.add('DRIVE_STRAIGHT' , car_state.DriveStraight(), transitions={'success': 'LANE_TRACE'}) StateMachine.add('DETECT_STOP_SIGN', car_state.DetectedStopSign(), transitions={'success': 'PROJECT_END', 'lane_trace': 'LANE_TRACE'}) StateMachine.add('DETECT_OBSTACLE', car_state.DetectedObstacle(), transitions={'success': 'LANE_TRACE'}) StateMachine.add('PROJECT_END', car_state.ProjectEnd(), transitions={'success': 'success'}) self.auto_drive.execute() if __name__ == "__main__": rospy.init_node('auto_drive') car_state_machine = CarStateMachine() car_state_machine.drive_car() while not rospy.is_shutdown(): rospy.spin() </pre>
참조	<p style="text-align: center;">[그림 3] 상태 패턴 Smach Viewer</p>
설명	<ol style="list-style-type: none"> 1. 파일이 실행되면 drive_car함수를 통해 DetectedBlockingBar클래스의 execute 함수가 실행된다. 2. execute의 조건이 만족되면 DetectedBlockingBar가 success 상태를 리턴 3. StateMachine이 LaneTrace 클래스를 찾아 LaneTrace의 exeute 함수가 실행 4. LaneTrace 클래스에 상태에 따라 다른 값을 리턴 5. 이와 같은 방식으로 return 값 상태에 따라 다음 동작이 이어지도록 하는 상태 패턴을 구현하였다.

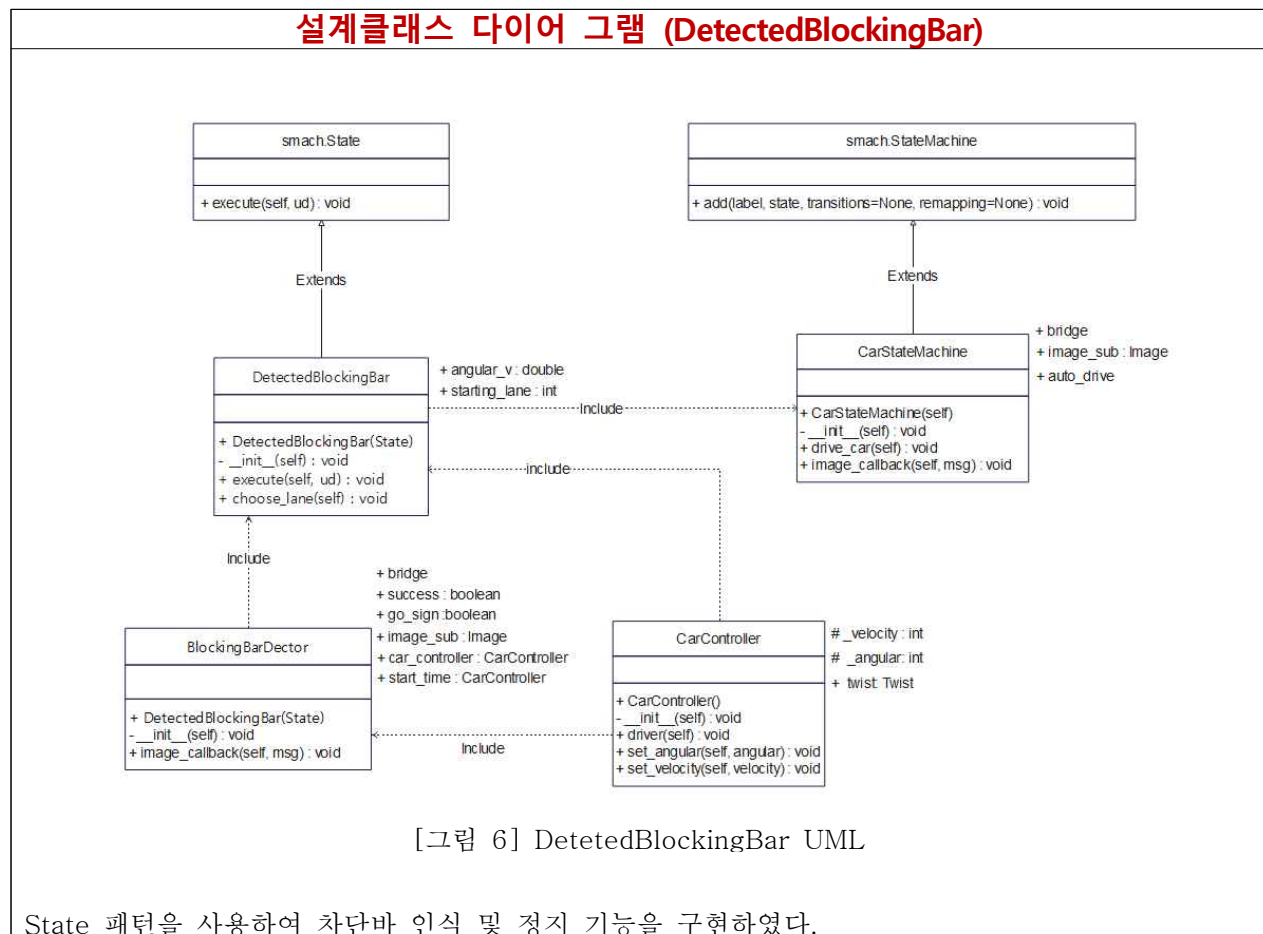
2.2 차단바 인식

2.2.1 분석

유스 케이스	차단바 인식
액터	turtlebot
목적	gazebo simulator에서 turtlebot이 자율주행 중에 차단바를 만났을 때, 차단바가 내려가 있다면 turtlebot이 잠시 정지했다가 다시 차단바가 올라가면 다시 출발합니다.
개요	turtlebot이 차단바와의 충돌을 방지 하기 위하여 중앙 카메라에 차단바가 인식되는 상태 인지를 점검하고 출발한다.
유형	기능
참조	 <p>[그림 4] 차단바 인식 use case diagram</p>
	

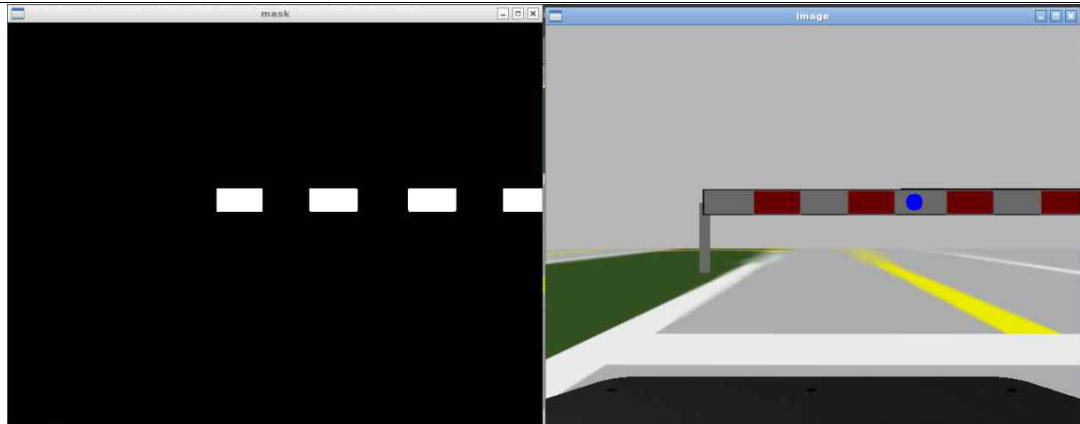
	액터	시스템
주 흐름 (main flow, basic flow)	<p>① gazebo simulator에서 turtlebot에게 기본으로 장착되어 있는 camera에 대한 image_msgs.msg를 전달한다.</p> <p>⑤ turtlebot은 받아온 image_msgs를 image_callback함수로 이미지를 가공하여 처리</p> <p>⑥ Line Tracer로 차선을 따라 주행</p>	<p>② Car State Machine이 blocking_bar_control을 실행하여 turtlebot을 전진시킨다</p> <p>③ opencv를 통해 전달 받은 이미지의 빨간 색상을 추출 한다.</p> <p>④ 카메라 인식 범위 안에 빨간색이 나타나면 turtlebot을 멈춘다</p>
대체 이벤트	④ 4초동안 turtlebot이 주행 하면 ②~⑤의 동작을 수행하지 않고 바로 ⑥으로 넘어간다	

2.2.2 설계



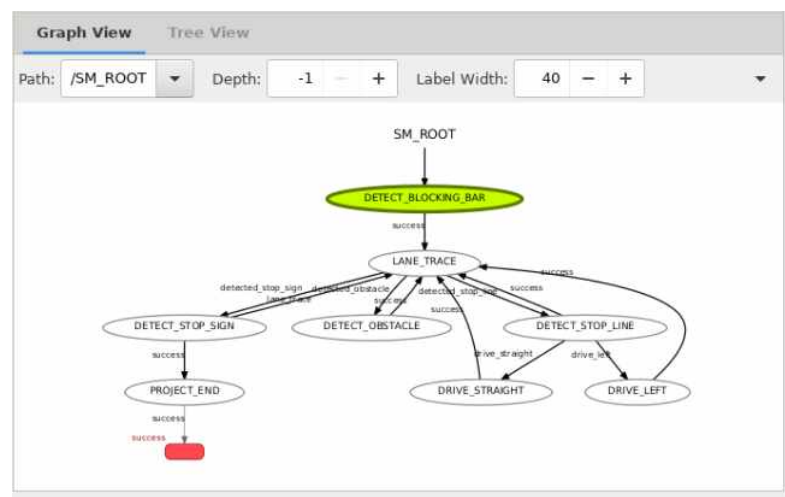
2.2.3 구현

유스 케이스	차단바 인식
기능	turtlebot이 차단바와의 충돌을 방지 하기 위하여 중앙 카메라에 차단바가 인식되는 상태 인지를 점검하고 출발한다.
소스	StateMachine으로 DetectedBlockingBar() 실행 DetectedBlockingBar (차단바 인식 액션)
	<pre> class DetectedBlockingBar(State): def __init__(self): State.__init__(self, outcomes=['success']) // StateMachin의다음 줄 상태로 간다 self.starting_lane = 0 self.angular_v = math.pi / 2 def execute(self, ud): start = self.choose_lane() if start == 1 or start == 2: blocking_bar = BlockingBarDetector() // BlockingBarDetector 객체 생성 while not rospy.is_shutdown(): if blocking_bar.go_sign: // go_sign이 true이면 turtlebot의 속도를 변경 blocking_bar.car_controller.set_velocity(0.8) else: // go_sign이 false이면 turtlebot의 속도를 0으로 변경 = 정지 blocking_bar.car_controller.set_velocity(0) blocking_bar.car_controller.drive() // 변경한 값을 발행 if time.time() - blocking_bar.start_time > 0: // 4초이상 움직이면 바로 다음 state로 return 'success' </pre>
	BlockingBarDetector class (차단바 인식)
소스	<pre> class BlockingBarDetector: def __init__(self): self.bridge = cv_bridge.CvBridge() self.success = False self.go_sign = True self.image_sub = rospy.Subscriber('camera/rgb/image_raw', Image, self.image_callback) self.car_controller = CarController() self.start_time = time.time() + 4 // 시간 초기화 현재 시간 + 4초 def image_callback(self, msg): image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8') hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV) // hsv 색상 초기화 lower_red = numpy.array([0, 30, 30]) // 가장 연한 빨강 upper_red = numpy.array([10, 255, 130]) // 가장 진한 빨강 img = cv2.inRange(hsv, lower_red, upper_red) // 찾는 이미지 색상의 최대, 최소값 초기화 h, w, d = image.shape // image의 높이, 넓이, 채널을 초기화 search_top = 1 search_bot = 3 * h / 4 img[0:search_top, 0:w] = 0 img[search_bot:h, 0:w] = 0 img[0:h, 0:250] = 0 M = cv2.moments(img) // 빨간색 이미지의 모멘텀 초기화 self.go_sign = True if M['m00'] > 0: // 빨간 이미지가 발견되면 cx = int(M['m10'] / M['m00']) cy = int(M['m01'] / M['m00']) cv2.circle(image, (cx, cy), 10, (255, 0, 0), -1) // 중심점에 파란점 찍기 self.go_sign = False self.start_time = time.time() + 4 // 현재 시간보다 4초 뒤 </pre>
	StateMachine으로 LaneTrace() 실행

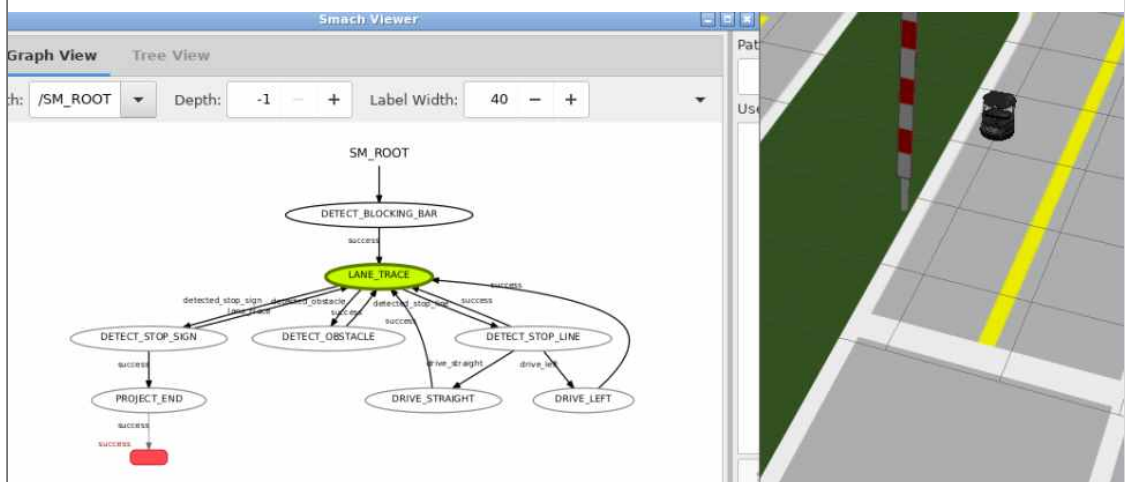


[그림 7] 마스킹하여 차단바 검출

실행
화면



[그림 8] 차단바 인식 상태



[그림 9] 차단바 통과 시 lane_trace로 상태 변화

참조

StateMachine, CarController

2.2.4 테스트

Test Case ID	TC-003				
Test Case 이름	차단바 인식 및 정지				
테스터	허세진				
관련 유스케이스	Object Recognition, Stop				
관련 기능 요구사항	SFR-401				
관련 소스 코드	blocking_bar_detecotor.py, car_state.py, car_state_machine.py				
테스트 시나리오	단계	단계 액션 (액터)	예상 결과 (시스템)	실행 기록 (테스터)	결과
	1		drive_car() 함수를 호출한다	가장 첫 번째 State인 DETECT_BLOCKIN_BAR가 실행된다	passed
	2	전진한다	linear.x를 0.8로 set	linear.x를 0.8로 초기화시키고 twist 메시지 구독	passed
	3		차단바를 인식한다	빨간색을 인식하는 M모멘트에 빨간색이 검출되면	passed
	4	정지한다.	카메라에 차단바 경계선과 중심점에 판점 출력 go_sign = false strat_time = 현재시간 + 4초	go_sign이 false가 되어서 linear.x가 0으로 초기화되고 twist 메시지 발행	passed
	5	4초 후	현재 시간 - start_time이 1을 넘어가면 현재 상태 종료하고 다음 상태로	DetectedBlockingBar(State) 종료 다음 상태	passed
작성자	합반 2조 / 허세진				
작성일	2021.12.02				
실행 모드	automatic				
테스터	합반 2조 / 허세진				
실행 일자	2021.12.02				
실행 결과	passed				

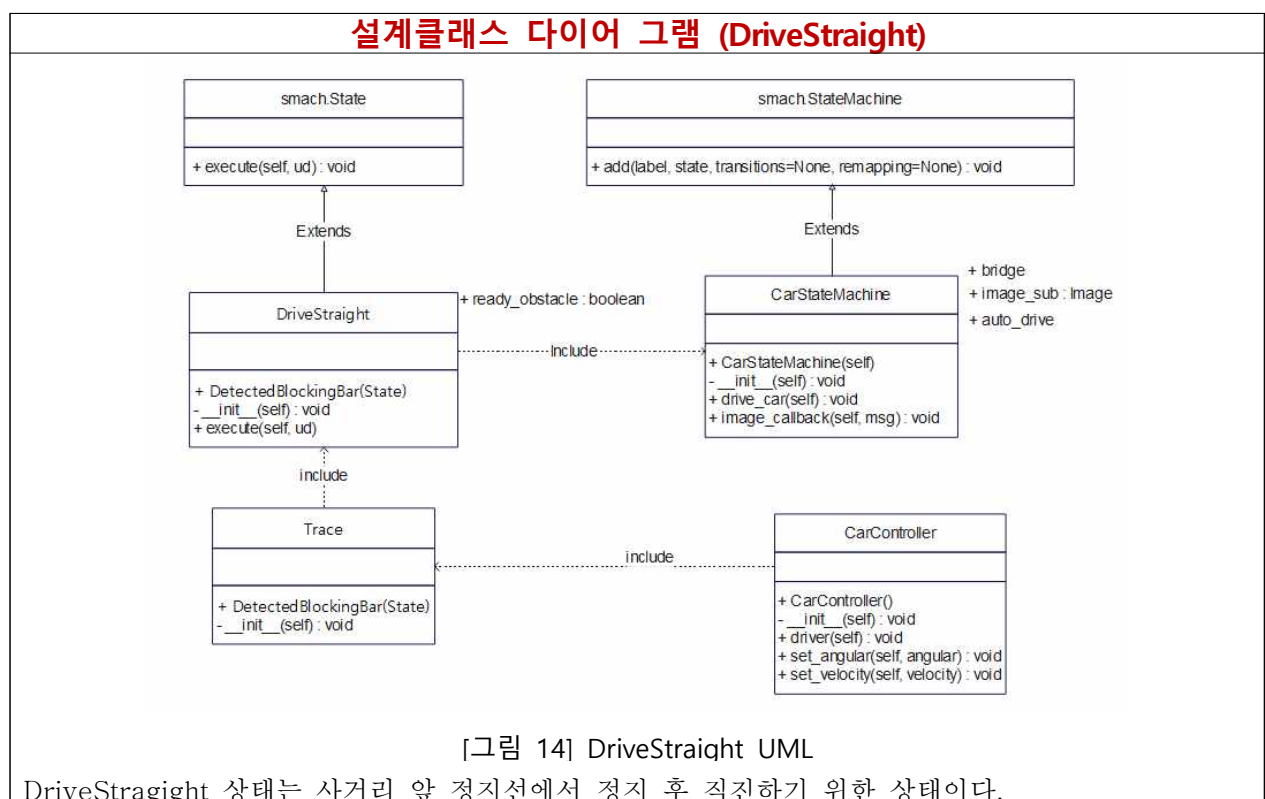
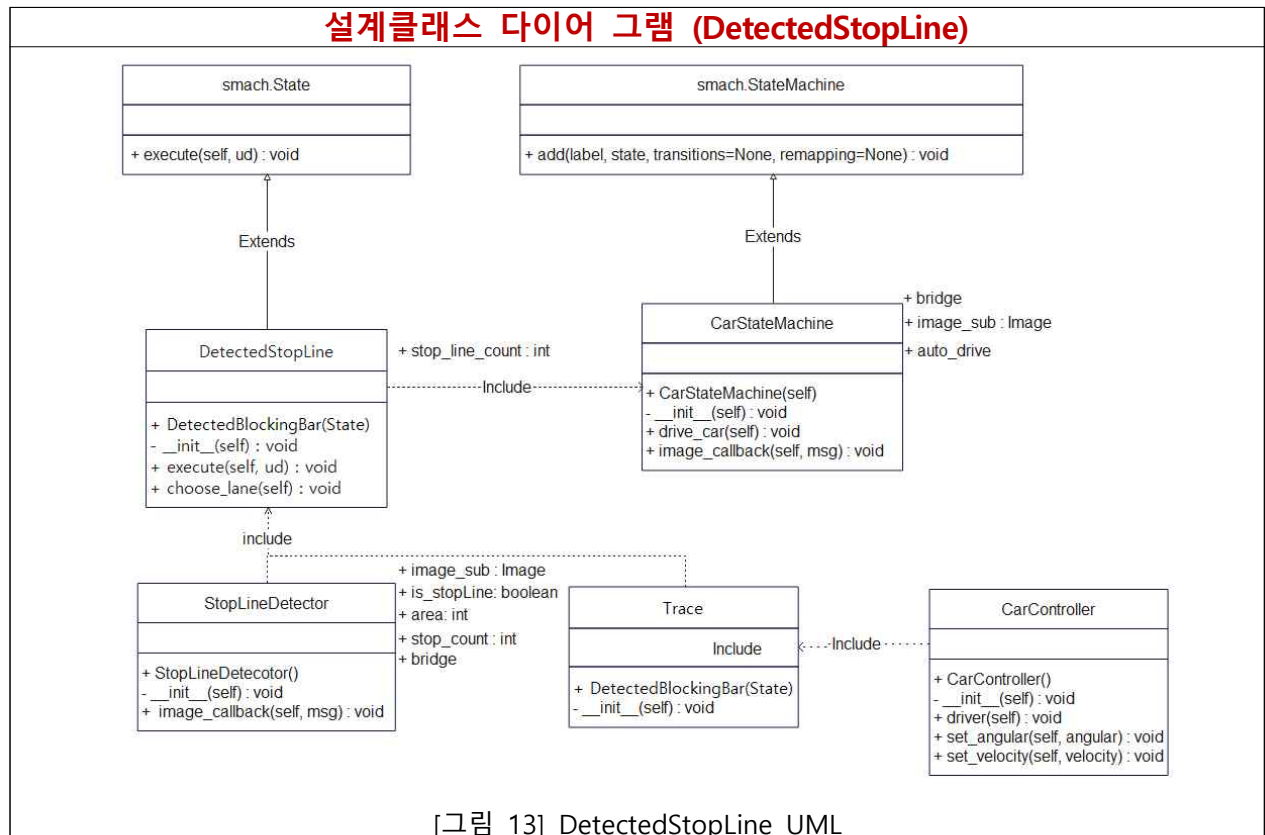
2.3 정지선 인식 및 정지

2.3.1 분석

유스 케이스	정지선 인식 및 정지
액터	Camera
목적	주행 중 정지선이 있으면 터틀봇을 3초간 정지시킨다.
개요	카메라에서 받아온 Image를 통해 정지선을 인식한다. 정지선이 인식된 경우 상태가 Lane Trace에서 detected_stop_line으로 변경되고, 3초 동안 정지시킨다.
유형	기능

참조	<div data-bbox="434 230 1267 495"> </div> <p>[그림 11] StopLineDetector Usecase</p> <div data-bbox="619 555 1102 1413"> </div> <p>[그림 12] StopLineDetector 순서도</p>				
주 흐름 (main flow, basic flow)	<table border="1"> <thead> <tr> <th data-bbox="292 1462 853 1512">액터</th><th data-bbox="853 1462 1418 1512">시스템</th></tr> </thead> <tbody> <tr> <td data-bbox="292 1512 853 1825"> ① Camera를 통해 Image를 받아온다. </td><td data-bbox="853 1512 1418 1825"> ② opencv를 통해 전달 받은 이미지에서 정지선을 인식한다. ③ 정지선의 area가 8000보다 큰 경우, is_stopline 변수를 True로 설정하고, 작은 경우는 False로 설정한다. </td></tr> </tbody> </table>	액터	시스템	① Camera를 통해 Image를 받아온다.	② opencv를 통해 전달 받은 이미지에서 정지선을 인식한다. ③ 정지선의 area가 8000보다 큰 경우, is_stopline 변수를 True로 설정하고, 작은 경우는 False로 설정한다.
액터	시스템				
① Camera를 통해 Image를 받아온다.	② opencv를 통해 전달 받은 이미지에서 정지선을 인식한다. ③ 정지선의 area가 8000보다 큰 경우, is_stopline 변수를 True로 설정하고, 작은 경우는 False로 설정한다.				
대체 이벤트	③ is_stopline이 True이면 상태가 detected_stop_line으로 변환되어 3초간 sleep 한다.				

2.3.2 설계



2.3.3 구현

유스 케이스	정지선 인식 및 정지
기능	turtlebot이 정지선을 인식하면 3초 동안 정지하도록 한다.
소스	<p>StopLineDetector 클래스의 초기화 함수에서 self.image_sub = rospy.Subscriber('camera/rgb/image_raw', Image, self.image_callback)를 통해 이미지를 구독하고, 콜백 함수에서 받아온 Image에 대한 마스크를 진행한다.</p> <pre> def image_callback(self, msg): image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8') hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV) lower_white = numpy.array([0, 0, 260]) upper_white = numpy.array([0, 0, 255]) mask = cv2.inRange(hsv, lower_white, upper_white) h, w = mask.shape mask[0:h * 3 / 5, 0:w] = 0 mask[h - (h / 8):h, 0:w] = 0 mask[0:h, 0:w / 4] = 0 mask[0:h, w - (w / 4):w] = 0 </pre> <p>[그림 15] stop_line_detector.py의 콜백 함수</p> <p>image의 색상 모델을 BGR에서 HSV로 변환한다. HSV 모델로 색상 모델을 변환함으로써, H(색조)만으로 순수한 색상 정보를 얻을 수 있기 때문에 주변 환경에 따른 수치 오차를 방지한다.</p> <p>lower white와 upper white 변수에 하얀 색상의 범위를 설정하여, hsv로 변환된 이미지에서 하얀색 색상 정보만이 검출되도록 한다. 또한, 주행 중 발생할 수 있는 다른 하얀색 물체로 인한 오작동을 방지하기 위해 mask 범위를 주행 화면의 정면 바닥 부분으로 좁혀준다.</p> <pre> _, thr = cv2.threshold(mask, 127, 255, 0) </pre> <p>cv2.threshold 함수를 사용하여 문턱값을 127로 설정하고, mask의 화소가 127보다 크면 해당 화소를 255로, 작으면 0으로 설정하여 이진화시킨다.</p>


```

_, contours, _ = cv2.findContours(thr, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
if len(contours) <= 0:
    return

cnt = contours[0]
self.area = cv2.contourArea(cnt)

if self.area > 8000.0:
    self.is_stopLine = True
else:
    self.is_stopLine = False

```

cv2.findContours 함수를 사용하여 이진화 이미지에서 윤곽선을 검색한다. 검색 방법으로는 모든 윤곽선을 검출하고, 계층 구조를 모두 형성하게 하는 cv2.RETR_TREE를 사용하고, 근사화 방법으로 cv2.CHAIN_APPROX_SIMPLE을 사용하여 윤곽점을 단순화 수평, 수직 및 대각선 요소를 압축하고 끝점만 남겨두도록 한다.

만일, contours의 길이가 0 이하이면 아무것도 검출되지 않은 것이므로, return시킨다.

cv2.contourArea 함수를 사용하여 contours의 0번 인덱스(최외곽 윤곽선)의 넓이를 구한다.

area가 8000보다 크면 is_stopLine을 True로 설정하여 정지선이 인식되었다는 것을 알리도록 한다. 반대로 8000 이하인 경우는 is_stopLine을 False로 설정하여 인식된 하얀 객체가 정지선이 아니라는 것을 알리도록 한다.

```

class LaneTrace(State):
    def __init__(self):
        State.__init__(self, outcomes=['detected_stop_line', 'detected_stop_sign', 'detected_obstacle'],
                        ready_obstacle = False)

    def execute(self, ud):
        lane = Trace()
        stop_line = StopLineDetector()
        stop_sign = StopSignDetector()
        detect_obstacle = ObstacleDetector()

        while not rospy.is_shutdown():
            if stop_line.is_stopLine:
                return 'detected_stop_line'

```

[그림 18] Car_State_machine.py의 LaneTrace 클래스

LaneTrace 클래스에서 생성한 StopLineDetector의 객체 stop_line을 사용하여 is_stopLine이 True인 경우 'detected_stop_line'을 반환하여 상태를 변환시킨다.

```
class DetectedStopLine(State):
    def __init__(self):
        State.__init__(self, outcomes=['success', 'drive_straight', 'drive_left'])
        self.stop_line_count = 0
        self.is_stop = True

    def execute(self, ud):
        lane = Trace()
        self.is_stop = True

        while not rospy.is_shutdown():
            if self.is_stop:
                self.stop_line_count += 1
                self.is_stop = False
                rospy.sleep(3)

            if self.stop_line_count == 4 or self.stop_line_count == 6:
                return 'drive_straight'

            if self.stop_line_count == 7:
                return 'drive_left'

        return 'success'
```

[그림 20] DetectedStopLine 클래스

DetectedStopLine 클래스는 정지선이 인식된 경우 호출되는 상태로, 3초간 sleep 시킴으로써 터틀봇이 정지선 앞에서 3초 동안 정지하도록 한다. 3초 후, success를 반환하여 상태를 LaneTrace로 전환하도록 한다.

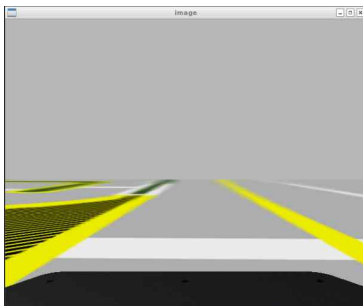


그림 21. Image 화면

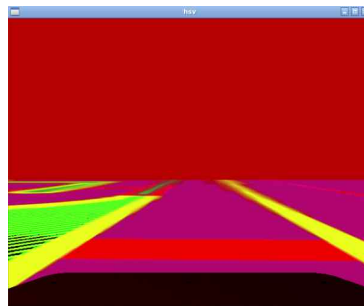


그림 22. HSV 화면

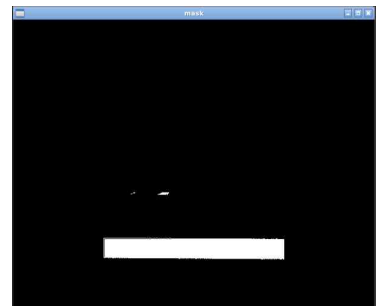


그림 23. mask 화면

실행
화면

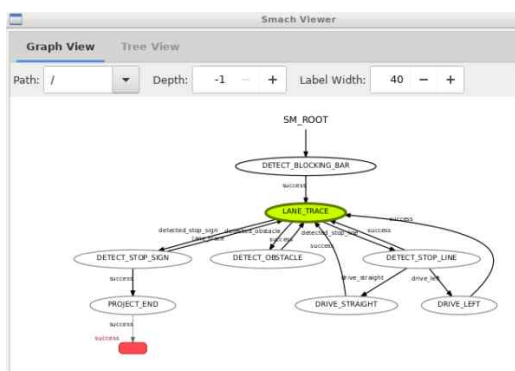


그림 24. 일반 주행의 경우의 상태

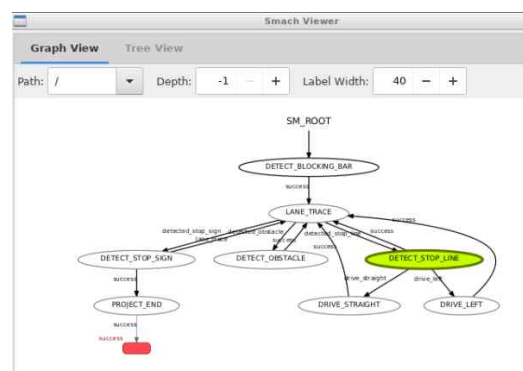


그림 25. 정지선 인식된 경우의 상태

참조

StateMachine, CarController

2.3.4 테스트

Test Case ID	TC-001				
Test Case 이름	정지선 인식 및 정지				
테스터	안대현				
관련 유스케이스	Line Recognition, Stop				
관련 기능 요구사항	SFR-201, SFR-402				
관련 소스 코드	stop_line_detector.py, car_state.py, car_state_machine.py				
테스트 시나리오	단계	단계 액션 (액터)	예상 결과 (시스템)	실행 기록 (테스터)	결과
	1	앞으로 전진한다.			passed
	2	정지선을 인식한다.			passed
	3		정지선 범위가 8000 초과이면 StopLineDetector의 is_stopLine을 True로 설정한다.	area가 8000보다 큰 경우 is_stopLine이 True로 설정되고, 8000 이하인 경우는 False로 설정된다.	passed
	4		is_stopLine이 True이면 lane_trace 상태에서 detected_stop_line 상태로 전이된다.	is_stopLine이 True이면 detected_stop_line을 반환하여 상태가 변경됨.	passed
	5		3초간 sleep한다.	3초간 sleep되어 터틀봇이 3초 동안 정지한다.	passed
	6		detected_stop_line 상태에서 lane_trace 상태로 전이된다.	success 반환을 통해 상태가 변경됨.	passed
	7	앞으로 전진한다.			passed
작성자	합반 2조 / 안대현				
작성일	2021.12.02				
실행 모드	automatic				
테스터	합반 2조 / 안대현				
실행 일자	2021.11.21				
실행 결과	passed				

2.4 차선 인식 및 추적

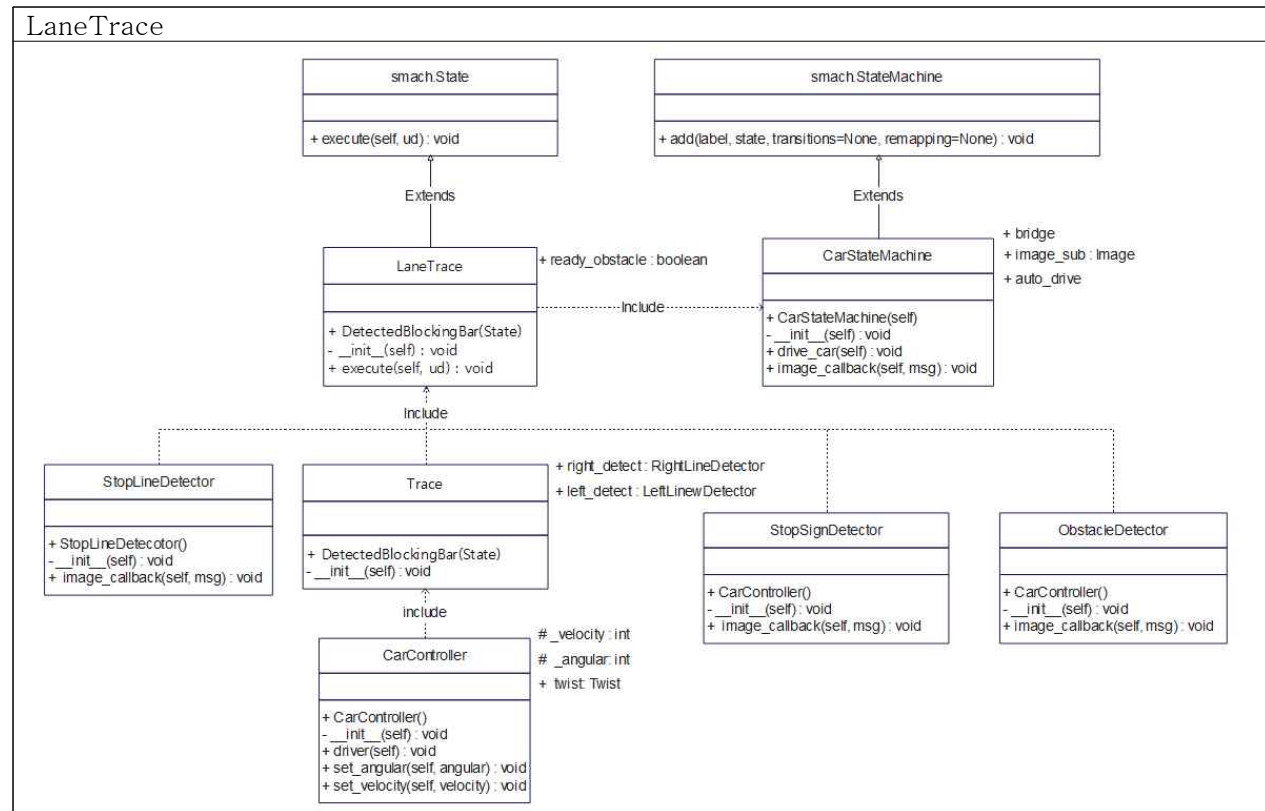
2.4.1 분석

유스케이스	차선 인식 및 추적
액터	Camera, Turtle Bot
목적	차선을 인식하여 터틀봇이 주행 중 차로를 이탈하지 않도록 한다.
개요	왼쪽 카메라와 오른쪽 카메라에서 Canny 알고리즘을 사용하여 차선을 인식하고, 인식되는 차선에 따라 주행 방향을 설정한다.
유형	기능

<p>참조</p>	<pre> graph TD Start([Start]) --> LeftCamera[Left Camera] Start --> RightCamera[Right Camera] LeftCamera -- Image --> Subscriber[Subscriber(Image)] RightCamera -- Image --> Subscriber Subscriber --> Decision1{양쪽 선을 다 인식하는가?} Decision1 -- YES --> GoStraight1[직진] Decision1 -- NO --> Decision2{오른쪽 선을 인식하지 못하는가?} Decision2 -- YES --> RightTurn[우회전] Decision2 -- NO --> Decision3{왼쪽 선을 인식하지 못하는가?} Decision3 -- YES --> LeftTurn[좌회전] Decision3 -- NO --> GoStraight2[직진] GoStraight1 --> End([End]) RightTurn --> End LeftTurn --> End GoStraight2 --> End </pre>				
<p>주 흐름 (main flow, basic flow)</p>	<table border="1"> <thead> <tr> <th data-bbox="292 1478 853 1556">액터</th><th data-bbox="853 1478 1418 1556">시스템</th></tr> </thead> <tbody> <tr> <td data-bbox="292 1556 853 1989"> <p>① Left Camera와 Right Camera를 통해 Image를 각각 받아온다.</p> <p>④ 터틀봇이 직진한다.</p> </td><td data-bbox="853 1556 1418 1989"> <p>② opencv를 통해 전달 받은 이미지에서 Canny 알고리즘을 통해 차선을 검출한다.</p> <p>③ 양쪽 선을 다 인식하는 경우 각속도를 0으로 설정한다.</p> </td></tr> </tbody> </table>	액터	시스템	<p>① Left Camera와 Right Camera를 통해 Image를 각각 받아온다.</p> <p>④ 터틀봇이 직진한다.</p>	<p>② opencv를 통해 전달 받은 이미지에서 Canny 알고리즘을 통해 차선을 검출한다.</p> <p>③ 양쪽 선을 다 인식하는 경우 각속도를 0으로 설정한다.</p>
액터	시스템				
<p>① Left Camera와 Right Camera를 통해 Image를 각각 받아온다.</p> <p>④ 터틀봇이 직진한다.</p>	<p>② opencv를 통해 전달 받은 이미지에서 Canny 알고리즘을 통해 차선을 검출한다.</p> <p>③ 양쪽 선을 다 인식하는 경우 각속도를 0으로 설정한다.</p>				

	<p>⑥ 터틀봇이 우회전한다.</p> <p>⑧ 터틀봇이 좌회전한다.</p> <p>⑩ 터틀봇이 직진한다.</p>	<p>⑤ 오른쪽 선을 인식하지 못하는 경우 각속도를 -0.8로 설정한다.</p> <p>⑦ 왼쪽 선을 인식하지 못하는 경우 각속도를 0.8로 설정한다.</p> <p>⑨ 양쪽 선을 인식하지 못하는 경우 각속도를 0으로 설정한다.</p>
<p>대체 이벤트</p>	-	


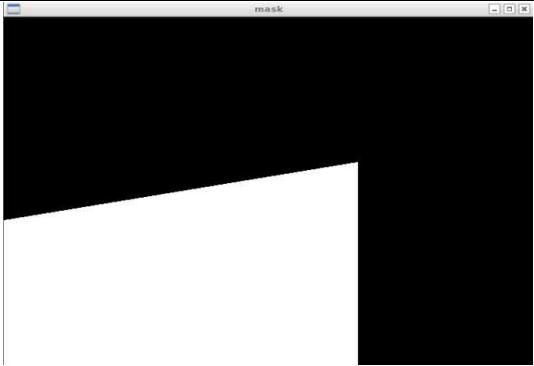
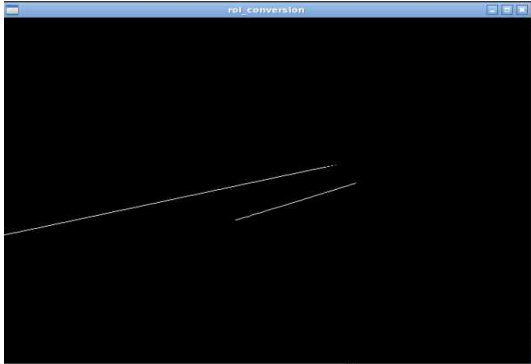
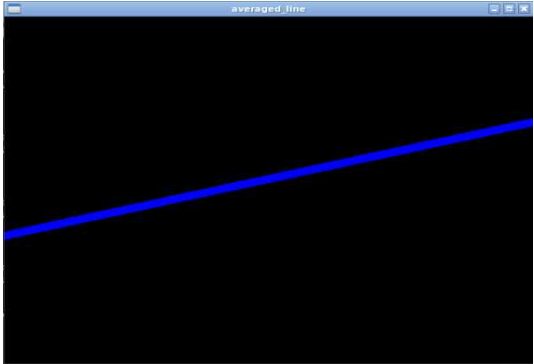
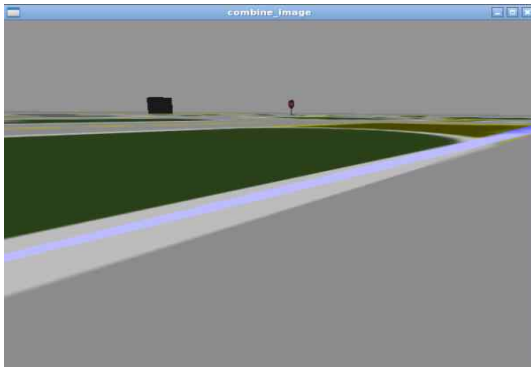
2.4.2 설계



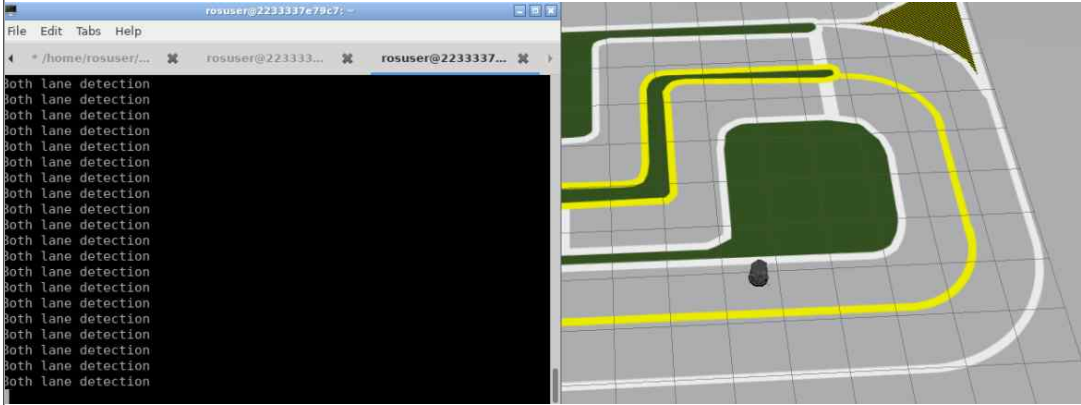
2.4.3 구현

유스 케이스	차선 인식
기능	터틀 봇의 양옆 차선을 인식한다. (Canny Edge 알고리즘 사용)
소스	<div data-bbox="558 448 1125 504"> <p>left_line_detector.py의 LeftLineDetector 클래스 초기화 함수에서 구독하는 Subscriber의 콜백 함수</p> </div> <div data-bbox="296 537 1396 1713"> <p>1. 이미지 받아오기</p> <pre>def image_callback(self, msg): cv2_img = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8') lanelines_image = cv2_img.copy()</pre> <p>왼쪽 카메라로부터 이미지를 받아오고, 해당 이미지의 복사본을 lanelines_image에 저장한다.</p> <p>2. cv2.Canny() 함수로 Canny Edge 알고리즘 실행</p> <pre>edge = cv2.Canny(lanelines_image, 100, 200)</pre> <p>첫 번째 인자인 lanelines_image는 canny 알고리즘을 수행할 이미지, 100은 최소 문턱값, 200은 최대 문턱값을 의미한다. 문턱값에 따라 객체의 테두리가 이진화되어 검출된다.</p> <p>3. 관심영역 설정</p> <pre>image_height = edge.shape[0] image_width = edge.shape[1] vertices = np.array([[(0, image_height), (0, image_height/2+40), (int(image_width - image_width / 3), int(image_height / 2) - 40), (int(image_width - image_width / 3), int(image_height))]]) image_mask = np.zeros_like(edge) if len(edge.shape) > 2: color = (255,255,255) else: color = 255 cv2.fillPoly(image_mask, vertices, color) roi_conversion = cv2.bitwise_and(edge, image_mask)</pre> <p>이미지의 높이와 너비를 받아온 뒤, 차선을 인식할 범위를 지정하여 vertices에 저장한다.</p> <p>np.zeros_line() 함수를 사용하여 edge 사이즈 만큼의 0으로 가득 찬 Array를 만들어서 image_mask에 저장한다.</p> <p>cv2.fillPoly 함수를 사용하여 image_mask에서 vertices 영역을 하얀색으로 채운 다각형을 그려서 영역을 확인할 수 있도록 한다.</p> <p>cv2.bitwise_and() 함수를 통해 edge와 image_mask 영역에서 서로 공통으로 겹치는 부분을 추출하여 roi_conversion에 저장한다.</p> </div>

	<h4>4. 라인 이어 그리기</h4> <pre> self.lines = cv2.HoughLinesP(roi_conversion, 1, np.pi / 180, 100, minLineLength=20, maxLineGap=5) left_fit = [] right_fit = [] if self.lines is not None: for line in self.lines: x1, y1, x2, y2 = line.reshape(4) parameter = np.polyfit((x1, x2), (y1, y2), 1) slope = parameter[0] intercept = parameter[1] if slope < 0: left_fit.append((slope, intercept)) else: right_fit.append((slope, intercept)) left_fit_average = np.average(left_fit, axis=0) left_fit_average = np.round(left_fit_average, 8) try: slope, intercept = left_fit_average except TypeError: slope, intercept = 0, 0 if slope == 0: slope = -0.4 y1 = lanelines_image.shape[0] y2 = int(y1 * (3 / 5)) x1 = int((y1 - intercept) / slope) x2 = int((y2 - intercept) / slope) lines_image = np.zeros_like(lanelines_image) if self.lines is not None: cv2.line(lines_image, (x1, y1), (x2, y2), (255, 0, 0), 10) combine_image = cv2.addWeighted(lanelines_image, 0.8, lines_image, 1, 1) </pre> <p>cv2.HoughLinesP() 함수를 통해 roi_conversion에서 검출된 픽셀들 중 서로 직선 관계를 갖는 픽셀들만 골라내도록 허프 선 변환을 수행한다.</p> <p>이때, 결론적으로 여러 직선들이 검출되는데 이를 하나의 선으로 만들어주기 위해 기울기와 y절편을 평균으로 해서 하나의 기울기와 y절편을 갖도록 만든다.</p> <p>combine_image = cv2.addWeighted(lanelines_image, 0.8, lines_image, 1, 1)</p> <p>cv2.addWeighted() 함수를 사용하여 원본 이미지에 평균 차선을 덧댄 combine_image를 구한다.</p>
소스	<p>Right_line_detector.py의 RightLineDetector 클래스 초기화 함수에서 구독하는 Subscriber의 콜백 함수</p> <pre> vertices = np.array([(image_width/3, image_height), (image_width/3, image_height/2), (image_width, image_height/2-20), (image_width, image_height)]) </pre> <p>관심영역을 오른쪽 카메라에 맞추어 설정하고, LeftLineDetector 클래스와 동일하게 Canny Edge 알고리즘을 적용하여 직선을 검출해낸다.</p>
실행 화면	<p>실행 화면은 왼쪽 카메라에서 차선을 인식하는 방법에 해당한다.</p> <p>원본 이미지에서 마스크 영역을 설정하고, 해당 영역 내의 차선을 검출한다.</p> <p>검출된 차선들의 평균값을 통해 평균 차선을 구하고, 원본 이미지에 평균 차선을 가중치를 조절하여 더한다.</p>

		
	그림 36. 원본 이미지	그림 37. 마스크 영역
		
	그림 38. 차선 검출	그림 39. 평균 차선
		
	그림 40. 원본 + 평균 차선	
참조	StateMachine, CarController	

유스 케이스	차선 추적
기능	터틀봇 주행 중 인식되는 차선에 따라 주행 방향을 결정한다.
소스	car_state.py의 LaneTrace 클래스 execute 함수
	<pre> while not rospy.is_shutdown(): if stop_line.is_stopLine: return 'detected_stop_line' if stop_sign.is_stopSign: self.ready_obstacle = True </pre>

	<pre> return 'detected_stop_sign' if detect_obstacle.is_obstacle and self.ready_obstacle: return 'detected_obstacle' if lane.right_detect.lines is None and lane.left_detect.lines is None: lane.car_controller.set_velocity(0.5) lane.car_controller.set_angular(0) elif lane.right_detect.lines is None: lane.car_controller.set_angular(-0.8) lane.car_controller.set_velocity(0.5) elif lane.left_detect.lines is None: lane.car_controller.set_angular(0.8) lane.car_controller.set_velocity(0.5) else: lane.car_controller.set_velocity(0.8) lane.car_controller.set_angular(0) lane.car_controller.drive() </pre>
	<p>주행 중 정지선, 정지표지판, 장애물이 검출되는 경우 해당 상태로 전이되도록 해당 상태 값을 반환한다.</p> <p>다른 이벤트가 발생하지 않는 경우에는 차선에 따라 주행이 가능하도록 한다. 이때, 양쪽 선 모두 검출되지 않으면 직진, 우측 선이 검출되지 않으면 우회전, 좌측 선이 검출되지 않으면 좌회전, 양쪽 선 모두 검출되면 직진하도록 한다.</p>
<p>실행 화면</p>	 <p>그림 41. 양쪽 차선 인식</p>

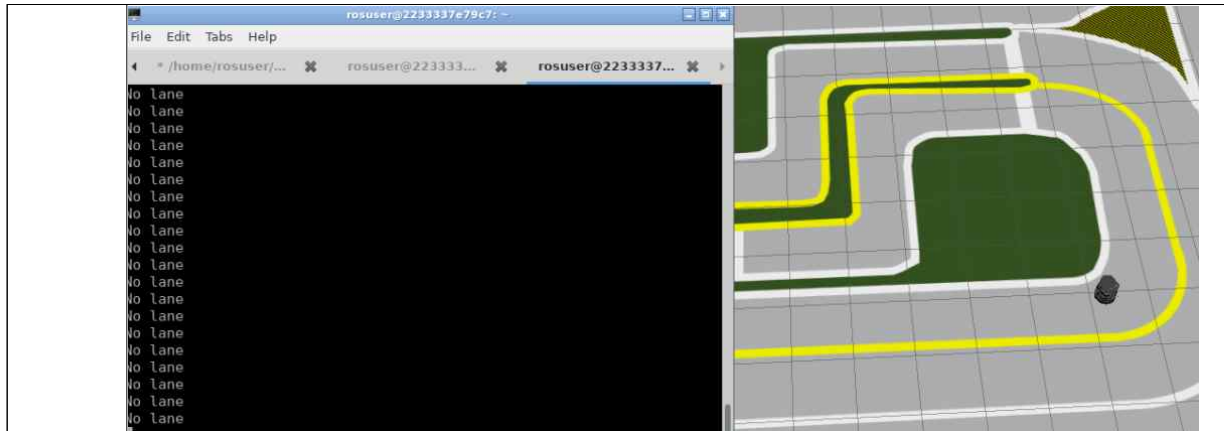


그림 42. 왼쪽 차선 미검출

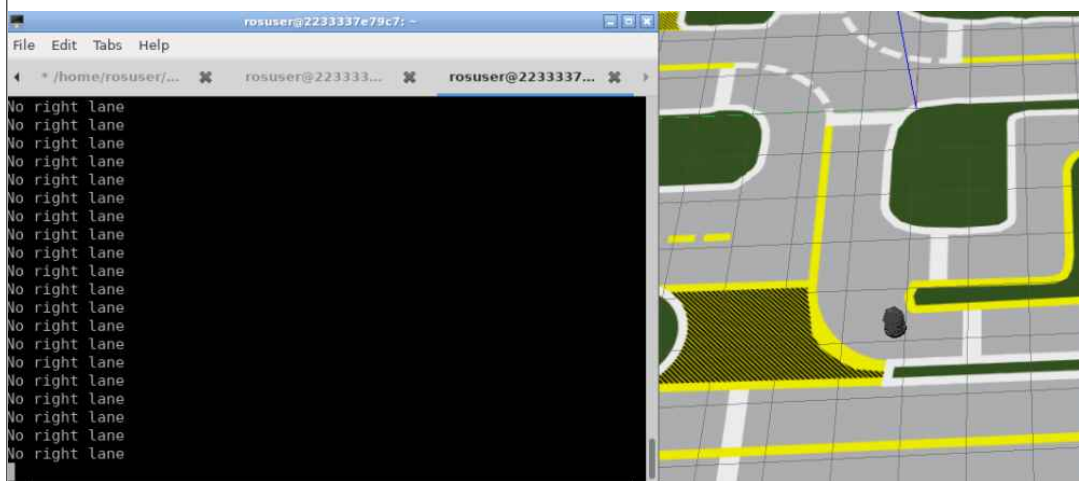
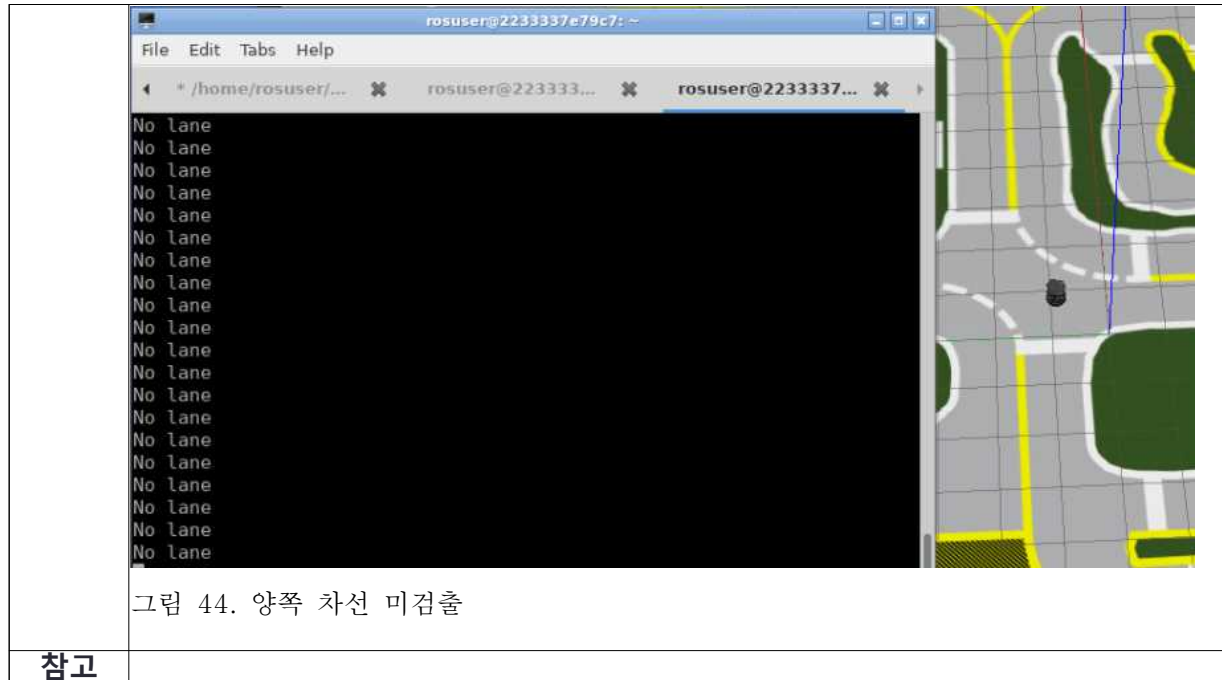


그림 43. 오른쪽 차선 미검출

양쪽 차선이 미검출되는 경우는 주행 중 속도나 터틀봇 각도에 따라서 일시적으로 차선이 인식되지 않는 경우로, 해당 상황에서는 터틀봇을 다음 차선을 인식할 때까지 직진하게 하였다.



2.4.4 테스트

Test Case ID	TC-005				
Test Case 이름	차선 인식 및 추적				
테스터	안대현				
관련 유스케이스	line recognition				
관련 기능 요구사항	SFR-104, SFR105, SFR-403				
관련 소스 코드	trace.py, left_line_detector.py, right_line_detector.py, car_state.py, car_state_machine.py				
테스트 시나리오	단계	단계 액션 (액터)	예상 결과 (시스템)	실행 기록 (테스터)	결과
	1	왼쪽, 오른쪽 카메라에서 이미지를 받아온다.			passed
	2		왼쪽 차선 미검출 시 좌회전한다.	왼쪽 차선 미검출 시, 각속도는 0.8, 선속도는 0.5로 주행한다.	passed
	3		오른쪽 차선 미검출 시 우회전한다.	오른쪽 차선 미검출 시, 각속도는 -0.8, 선속도는 0.5로 주행한다.	passed
	4		양쪽 차선 미검출 시 직진한다.	양쪽 차선 미검출 시, 각속도는 0, 선속도는 0.5로 주행한다.	passed
	5		양쪽 차선 검출 시 직진한다.	양쪽 차선 검출 시, 각속도는 0, 선속도는 0.8로 주행한다.	passed
작성자	합반 2조 / 안대현				
작성일	2021.12.01				
실행 모드	automatic				
테스터	합반 2조 / 안대현				
실행 일자	2021.11.19				
실행 결과	passed				

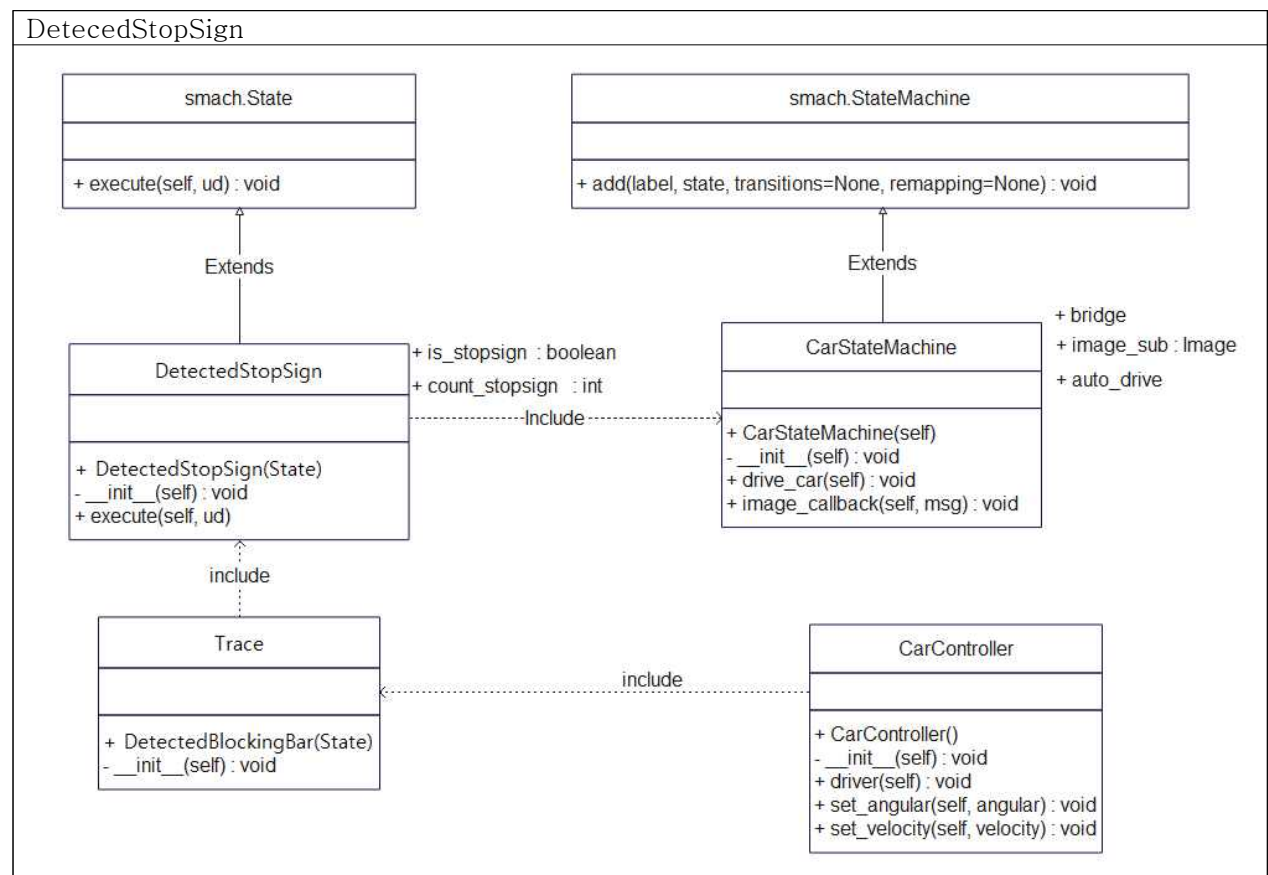
2.5 정지 표지판 인식 및 정지

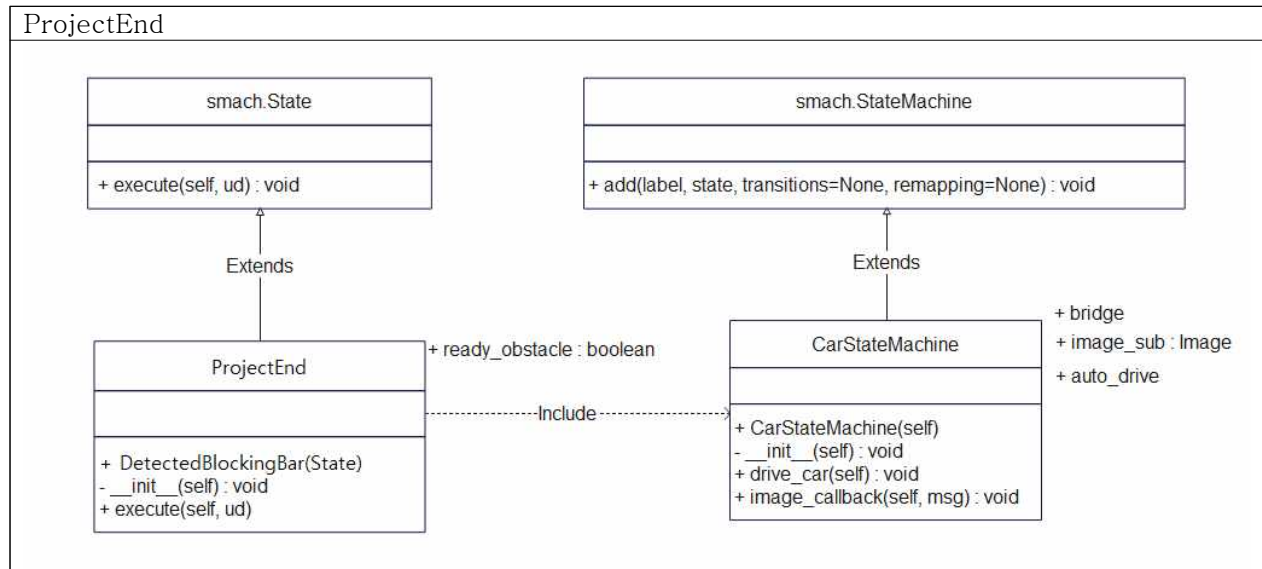
2.5.1 분석

유스 케이스	정지 표지판 인식 및 정지
액터	Camera, Turtle Bot
목적	주행 중 정지 표지판이 있으면 터틀봇을 3초간 정지시킨다.
개요	카메라에서 받아온 Image를 통해 정지 표지판을 인식한다. 정지 표지판이 인식된 경우 상태가 Lane Trace에서 detected_stop_sign으로 변경되고, 터틀 봇을 3초 동안 정지시킨다.
유형	기능
참조	<pre> sequenceDiagram actor camera participant CarStateMachine as Car State Machine participant TurtleBot as Turtle Bot camera->>CarStateMachine: Image activate CarStateMachine CarStateMachine->>CarStateMachine: StopSignDetector CarStateMachine->>CarStateMachine: DetectedStopSign CarStateMachine->>TurtleBot: Stop deactivate CarStateMachine Start([Start]) --> Camera[Camera] Camera --> Image[/Image/] Image --> Subscriber[Subscriber(Image)] Subscriber --> Detect{표지판의 테두리가 검출되었는가?} Detect -- No --> End([End]) Detect -- Yes --> StateChange[detected_stop_sign 으로 상태 변화] StateChange --> Count{검출된 표지판 카운트가 몇 개인가?} Count -- 1개 --> Stop1[터틀봇을 3초 동안 정지시킨다] Stop1 -- "lane_trace 반환" --> End Count -- 2개 --> Stop2[터틀봇을 3초 동안 정지시킨다] Stop2 -- "success 반환" --> End </pre>

	액터	시스템
주 흐름 (main flow, basic flow)	① Camera를 통해 Image를 받아온다.	② opencv를 통해 전달 받은 이미지에서 정지 표지판을 인식한다. ③ 테두리가 검출된 경우, is_stopSign을 True로 설정한다. ④ 터틀봇의 주행을 제어한다.
대체 이벤트	③ is_stopSign이 True이면 상태가 detected_stop_sign으로 변환되어 ④ count_stopsign이 1이면 터틀봇을 3초 동안 정지시키고, 2이면 6초 뒤에 터틀봇을 3초 동안 정지시킨다.	

2.5.2 설계

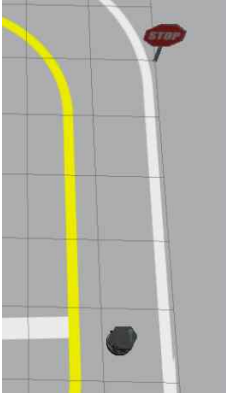
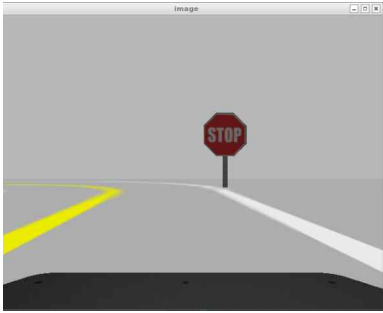

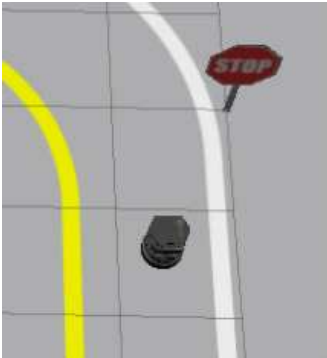





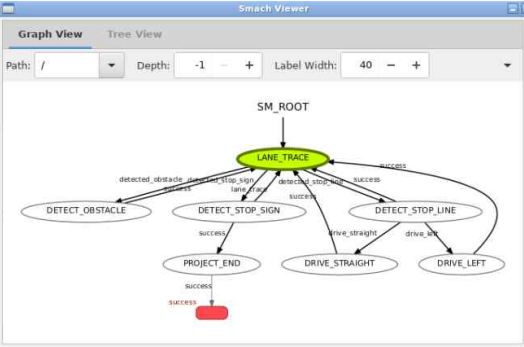
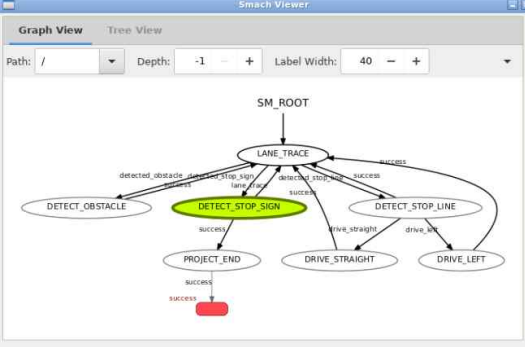
2.5.3 구현

주요 코드

유스 케이스	정지 표지판 인식
기능	터틀봇 주행 중 정지 표지판을 인식한다.
소스	<p>stop_sign_detector.py의 StopSignDetector 클래스 초기화 함수에서 구독하는 Subscriber의 콜백 함수</p> <pre> def image_callback(self, msg): image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8') hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV) # BGR 색상 모델을 HSV로 변경 # 빨간색의 범위 지정 lower_red = np.array([0, 0, 90]) upper_red = np.array([5, 5, 110]) red_img = cv2.inRange(hsv, lower_red, upper_red) h, w = red_img.shape block_bar_mask = red_img # 인식할 영역을 마스킹 block_bar_mask[0:0, 0:w] = 0 block_bar_mask[10:h, 0:w] = 0 # 윤곽선 검색 block_bar_mask, self.contours, hierarchy = cv2.findContours(block_bar_mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE) # 윤곽선이 검출되면 is_stopeSign을 True로 설정 if len(self.contours) > 0: self.is_stopSign = True </pre> <p>BGR 색상 모델을 HSV로 변경한 이미지에서 마스킹된 영역에 대해 빨간색 객체의 윤곽선을 검출한다. 정지 표지판 검출이 상당히 먼 거리에서부터 인식이 되기 때문에, 정지판의 1.5 ~ 2m 앞에서 정지할 수 있도록 마스크 영역을 화면의 상단으로 설정한다.</p>

<p>실행 화면</p>	<div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;">  <p>그림 50. 화면1</p> </div> <div style="text-align: center;">  <p>그림 51. image 화면</p> </div> <div style="text-align: center;">  <p>그림 52. 윤곽선 검출 화면</p> </div> </div> <div style="display: flex; justify-content: space-around; align-items: flex-start; margin-top: 20px;"> <div style="text-align: center;">  <p>그림 53. 화면 2</p> </div> <div style="text-align: center;">  <p>그림 54. mask 화면</p> </div> </div>
---------------------	--

<p>유스 케이스</p>	<p>정지 표지판 앞에서 정지</p>
<p>기능</p>	<p>터틀봇 주행 중 정지 표지판을 인식하면 3초 동안 정지한다.</p>
<p>소스</p>	<pre> car_state.py의 DetectedStopSign 클래스 execute 함수 while True: lane.car_controller.set_velocity(1) if lane.right_detect.lines is None: lane.car_controller.set_angular(-0.3) if lane.left_detect.lines is None: lane.car_controller.set_angular(0.3) lane.car_controller.drive() if time.time() - start_time > 0: if self.is_stopsign: if self.count_stopsign == 2: return 'success' return 'lane_trace' break start_time = time.time() + 3 while True: lane.car_controller.set_angular(0) </pre>

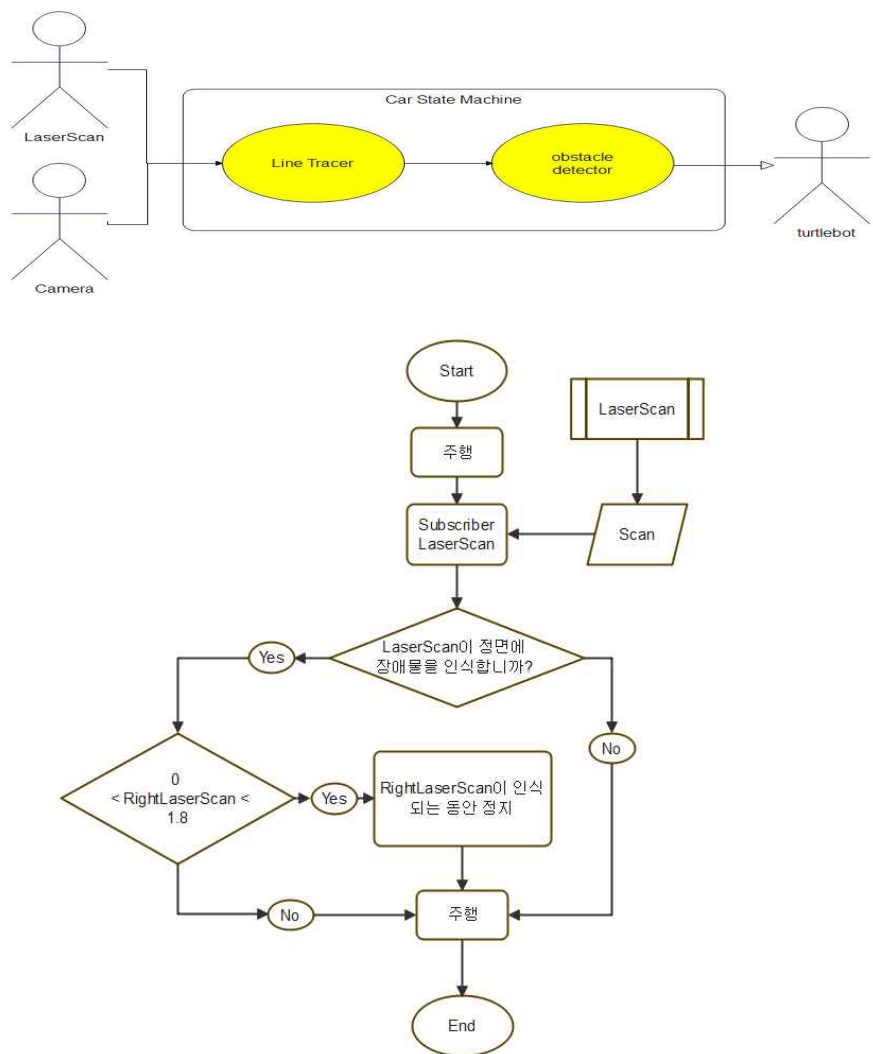
	<pre> lane.car_controller.set_velocity(0) lane.car_controller.drive() if time.time() - start_time > 0: self.is_stopsign = True break </pre> <p>count_stopsign이 2인 경우는 3초 정지 후 'success'를 반환하고, 2가 아닌 경우는 3초 정지 후 'lane_trace'를 반환한다.</p> <p>start_time을 현재 시간에서 3초를 더한 값으로 설정해준다. 무한 반복문에서 car_controller의 함수를 사용하여 각속도와 선속도를 0으로 설정한 뒤, 터틀 붓을 drive 시킨다. 만일, 현재 시간에서 start 값을 뺀 값이 0보다 크면 3초가 지난 경우이므로 break한다.</p>
사진	 
참조	<p>그림 55. 주행 상태</p> <p>그림 56. 정지 표지판 인식 상태</p> <p>StateMachine, CarController</p>

2.5.4 테스트

Test Case ID	TC-002				
Test Case 이름	정지표지판 인식 및 정지				
테스터	안대현				
관련 유스케이스	Sign Recognition, Stop				
관련 기능 요구사항	SFR-203, SFR-204, SFR-406				
관련 소스 코드	stop_sign_detector.py, car_state.py, car_state_machine.py				
테스트 시나리오	단계	단계 액션 (액터)	예상 결과 (시스템)	실행 기록 (테스터)	결과
	1	앞으로 전진한다.			passed
	2	정지 표지판을 인식한다.			passed
	3		정지 표지판 인식 상태로 전환한다.	State가 LaneTrace에서 DetectedStopSign으로 변환된다.	passed
	4		카운트에 따라 3초동안 선 속도를 0으로 설정한다.	count가 1이면 바로 3초 정지하고, count가 2이면 6초 뒤에 3초 정지한다.	passed
	5	정지한다.			passed
	6		카운트에 따라 상태 값을 반환한다.	count가 1이면 lane_trace를 반환하고, 2이면 success를 반환한다.	passed
작성자	함반 2조 / 안대현				
작성일	2021.12.02				
실행 모드	automatic				
테스터	함반 2조 / 안대현				
실행 일자	2021.11.30				
실행 결과	passed				

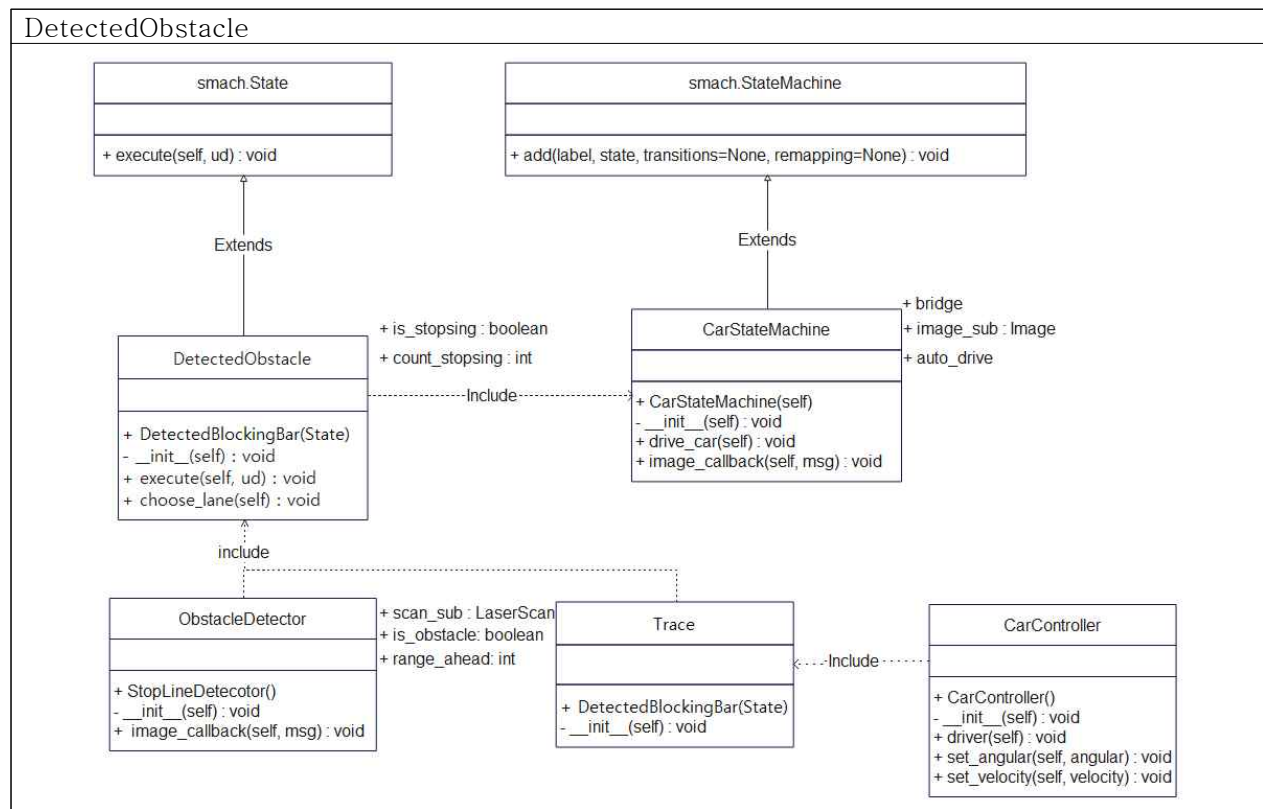
2.6 장애물 인식 및 정지

2.6.1 분석

유스 케이스	장애물 회피
액터	turtlebot
목적	gazebo simulator에서 turtlebot이 도착 지점까지 가기위해 장애물을 인식하고 장애물을 인식 했을 때, 멈췄다가 장애물이 사라지고 움직이도록한다.
개요	turtlebot이 진행하는 방향에 장애물을 센서로 감지하면 정지하여 장애물이 지나가고 난 후를 움직여야 한다.
유형	기능
참조	 <p>The diagram illustrates the obstacle avoidance process for a turtlebot. The top part is a Use Case Diagram showing the 'Car State Machine' containing two use cases: 'Line Tracer' and 'obstacle detector'. 'LaserScan' and 'Camera' actors are connected to the 'Car State Machine'. The 'obstacle detector' use case is connected to the 'turtlebot' actor. The bottom part is a Flowchart detailing the logic: It starts with 'Start', followed by '주행' (Driving). A 'Scan' event triggers a 'Subscriber LaserScan' process. A decision diamond asks 'LaserScan이 정면에 장애물을 인식합니까?' (Does LaserScan detect an obstacle in front?). If 'Yes', it leads to another decision diamond: '$0 < \text{RightLaserScan} < 1.8$'. If 'Yes' to this, it leads to a process box 'RightLaserScan이 인식되는 동안 정지' (Stop during RightLaserScan recognition), then to '주행' (Driving), and finally to 'End'. If 'No' to either decision, it goes directly to '주행' (Driving) and then to 'End'. There is also a 'LaserScan' data store connected to the 'Scan' event.</p>

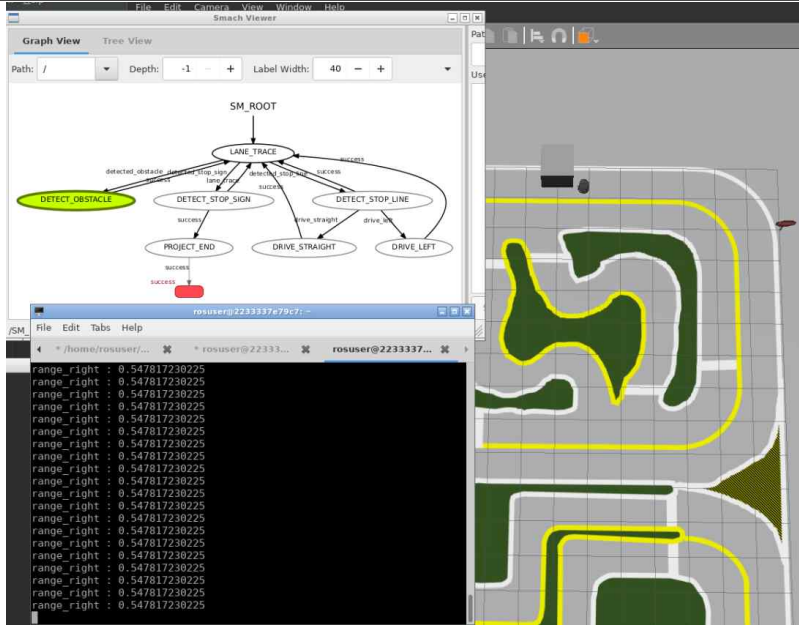
	액터	시스템
주 흐름 (main flow, basic flow)	<p>① gazebe simulalater에서 turtlebot에게 기본으로 장착되어 있는 camera에 대한 image_msgs.msg를 전달한다.</p> <p>② turtlebot에게 기본으로 장착되어 있는 LaserScan에 대한 sensor_msgs.msg를 전달한다.</p> <p>⑤ turtlebot은 시스템에서 받아온 senser_msgs를 처리</p>	<p>③ LaneTrace 중에 Laser가 정면의 obstacle을 인식하면 state패턴으로 dedcted_obstacle 실행</p> <p>④ range_right 값이 1.8 or range_ahead 가 3.7이하면 turtlebot 정지</p>
대체 이벤트	③ 장애물을 인식하지 않으면 터틀봇은 계속 움직인다.	

2.6.2 설계



2.6.3 구현

유스 케이스	장애물 인식
기능	turtlebot이 진행하는 방향에 장애물을 센서로 감지하면 정지하여 장애물이 지나가고 난 후를 움직여야 한다.
소스	<pre> StateMachine으로 DetectedObstacle() 실행 DetectedObstacle (장애물 인식 액션) class DetectedObstacle(State): def __init__(self): State.__init__(self, outcomes=['success']) // StateMachin의다음 줄 상태로 간다 def execute(self, ud): lane = Trace() // Trace 객체 초기화 detect_obstacle = ObstacleDetector() // ObstacleDetector 객체 초기화 while not rospy.is_shutdown(): start_time = time.time() + 5 // 시작 시간 + 5sec while True: // 오른쪽 스캔 값이 0보다 크거나 1.8보다 작으면 멈춘다 if 0 < detect_obstacle.range_right < 1.8: lane.car_controller.set_velocity(0) lane.car_controller.set_angular(0) lane.car_controller.drive() else: // 선을 탐지해서 turtlebot을 움직인다. if lane.right_detect.lines is None and lane.left_detect.lines is None: lane.car_controller.set_velocity(0.3) lane.car_controller.set_angular(0) elif lane.right_detect.lines is None: lane.car_controller.set_angular(-0.2) lane.car_controller.set_velocity(1.0) elif lane.left_detect.lines is None: lane.car_controller.set_angular(0.2) lane.car_controller.set_velocity(1.0) else: lane.car_controller.set_velocity(0.8) lane.car_controller.set_angular(0) lane.car_controller.drive() if time.time() - start_time > 0: return 'success' ObstacleDetector class (장애물 인식) class ObstacleDetector: def __init__(self): self.range_ahead = 0 self.range_right = 0 self.is_obstacle = False // LaserScan 값을 구독 self.scan_sub = rospy.Subscriber('scan', LaserScan, self.scan_callback) self.car_controller = CarController() // CarController 객체를 구독 def scan_callback(self, msg): // range_ahead에 scan 정면 값 초기화 self.range_ahead = msg.ranges[len(msg.ranges) / 2] self.range_right = msg.ranges[220] // range_right 오른쪽 센서값 초기화 if math.isnan(self.range_ahead): // range_ahead에 nan값이 들어있으면 0으로 변경 self.range_ahead = 0 // range_ahead을 0으로 </pre>

	<pre> if math.isnan(self.range_right): // range_right에 nan값이 들어있으면 self.range_right =0 // range_right를 0으로 if 0 <self.range_ahead <3.7: // range_ahead 0보다 크고 3.7보다 작으면 self.is_obstacle =True // is_obstacle값을 true로 변경 </pre>
	<p style="text-align: center;">StateMachine으로 LaneTrace() 실행</p>
실행 화면	
참조	<p style="text-align: center;">그림 61. detect_obstacle 상태 StateMachine, Trace, CarController</p>

2.6.4 테스트

Test Case ID	TC-004				
Test Case 이름	장애물 인식 및 정지				
테스터	허세진				
관련 유스케이스	Object Recognition, Stop				
관련 기능 요구사항	SFR-202, SFR-405				
관련 소스 코드	obstacle_detector.py, car_state.py, car_state_machine.py				
테스트 시나리오	단계	단계 액션 (액터)	예상 결과 (시스템)	실행 기록 (테스터)	결과
	1	주행 중	선을 따라서 주행한다		passed
	2	장애물을 인식한다	is_obstacle = True	obstacle이 laserscan에 검출되어 range_ahead에 값이 0 이상 3.7 이하가 되어 is_obstacle = true	passed
	3	정지한다.	range_right가 0이상 1.8이하 linear.x = 0 angular.z = 0	twist.linear.x = 0 twist.angular.z = 0 twist 메시지 발행	passed
	4	움직인다	range_rigth가 1.8 이상이면 선을 인식하여 움직인다		passed
	5	5초 후	현재 시간 - start_time이 1을 넘어가면 현재 상태 종료하고 다음 상태로	DetectedObstacle(State) 종료 다음 상태	passed
작성자	합반 2조 / 허세진				
작성일	2021.12.02				
실행 모드	automatic				
테스터	합반 2조 / 허세진				
실행 일자	2021.12.02				
실행 결과	passed				

3. 프로젝트 결과

3.1 프로젝트 완성도

주차	추정치 총합	기능 요구사항 수	계획 총합	작성일
1	0	0	0	21.10.05
2	63	20	0	21.10.12
3	63	20	0	21.10.15
4	63	20	1	21.10.23
5	63	20	3	21.11.09
6	63	20	7	21.11.15
7	59	19	26	21.11.23
8	59	19	39	21.11.30

그림 63. 개발 진행 상황

SFR-303	곡선코스를 주행한다	8	중	분석
SFR-304	방향전환 코스를 주행한다	8	중	시작
SFR-305	평행주차 코스를 주행한다.	5	중	시작

그림 64. 미완료 기능 요구 사항

3.2 1차선/2차선 주행 완성도

3.2.1 차선 변경 방법

차선 변경 방법

car_state_machine.py를 실행하여 프로그램을 시작하면 사용자로부터 출발 차선 1 또는 2를 입력 받도록 한다. 사용자가 1을 입력하면 기본 위치에서 시작하고, 2를 입력하면 2차선으로 차선을 변경한 후 주행을 시작한다. 해당 기능은 time을 사용하여 구현하였기 때문에 FPS에 따라서 차선 변경에 오차가 발생할 수 있다.




그림 66. 사용자가 2를 입력한 경우

3.2.2 구간별 달성도

수행 작업	A 달성도(%)	B 달성도(%)
차단바 인식 및 정지	100	100
정지선 인식 및 정지	90	90
차선 인식 및 주행	80	80
정지표지판 인식 정지	100	100
장애물 인식 및 정지	100	100
굴절 코스 주행	70	80
곡선 코스 주행	50	50
방향 전환	30	30
평행 주차 주행	0	0

정지선 인식 및 정지의 달성도 미흡 원인

커브길 다음에 나오는 정지선의 경우, 터틀봇의 회전 각도에 따라서 정지선을 놓치게 되는 경우가 발생하여 가제보를 실행하는 FPS 또는 실행 횟수에 따라 인식률이 달라진다.

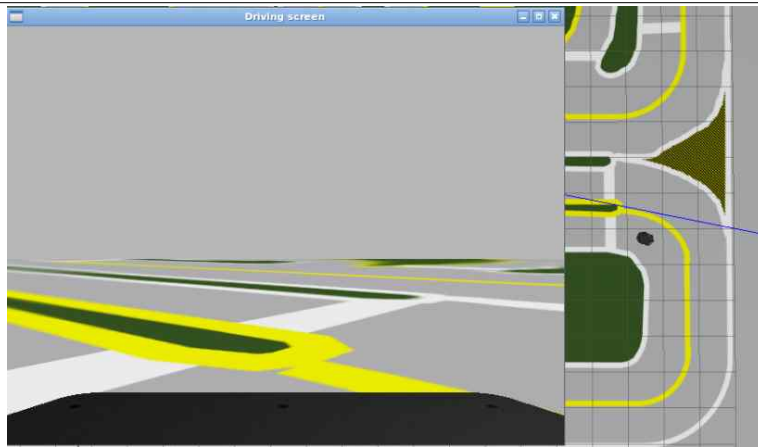


그림 67. 정지선이 미인식되는 상황

차선 인식 및 주행의 달성도 미흡 원인

초기 차선 인식을 Canny Edge 알고리즘을 사용하여 구현하려던 이유는 양 옆 차선을 검출해내고, 각 차선의 연장선을 이어 그려 두 차선이 교차하는 교차점을 터틀봇의 이동 방향으로 정하려고 하였다. 그러나, 주행 맵에서 곡선에 대한 차선 인식이 쉽지 않았고, 결국 커브길에서 양 옆 차선의 교차점을 구해내지 못하게 되었다.

해당 차선 인식 알고리즘을 구현하기 위해 상당한 시간이 투자되었기 때문에 기존과 아예 다른 방법으로 전환하기가 어려웠고, 프로젝트의 마감 기간을 고려하여 2.4.3에 기술한 방법으로 차선 인식 기능을 구현하였다.

초기 차선 인식 계획 사진

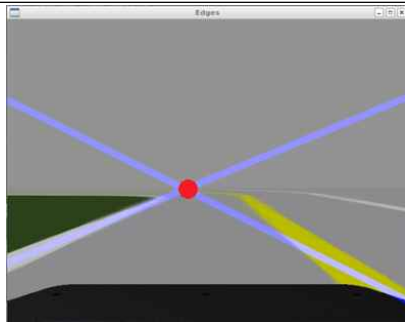


그림 68. 양 차선의 교차점

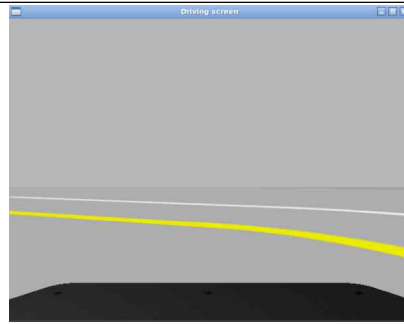


그림 69. 교차점 생성 불가

변경된 방법으로 해결하지 못한 문제

초기 계획에서 변경하여 본 프로그램에 적용한 차선 인식 방법 역시 다음과 같은 문제가 발생하였다. 그림 70의 경우는 터틀봇이 검정색과 노란색 선들로 이루어진 경고 바닥에 근접했을 때, 검출된 직선들의 평균을 구하는 과정에서 문제가 발생한다. 해당 상황은 터틀봇의 각도에 따라 차선이 정상 검출되는 경우도 있고, 그림 70과 같이 오검출되는 경우도 있다.

또한, FPS에 따라 양 옆 차선 인식을 놓치는 경우를 보완하기 위해 본 프로그램은 터틀봇을 전진시켜 다음 차선을 찾도록 구현하였는데 그림 71과 같은 경우는 회전을 하지 못하는 상황에 놓이게 된다.

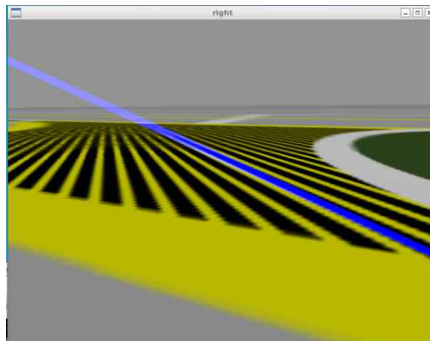


그림 70. 경고 바닥 차선 인식 오류

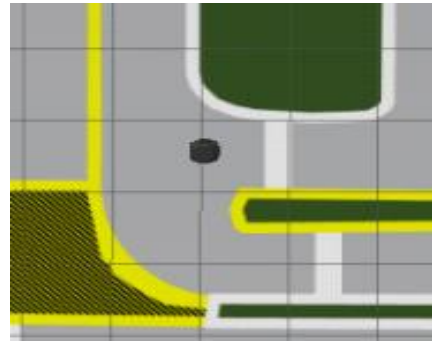


그림 71. 좌우 차선 인식 불가능한 경우

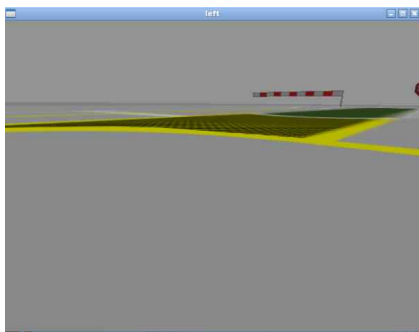


그림 72. 왼쪽 차선 인식 실패



그림 73. 오른쪽 차선 인식 실패

각 코스별 미흡한 달성도 원인

각 코스의 주행 성공률은 차선 인식의 정확도에 비례한다. 그러나, 본 팀이 구현한 차선 인식 알고리즘은 주행 상황에 따라 오검출이 발생하는 등 인식률이 불안정하기 때문에 달성도가 기대 결과보다 낮게 나왔다.

3.3 일정 계획 대비 달성도

Activity No.	소작업	소요 기간(일)	선행 작업	달성도(%)
A	터틀봇은 자동으로 출발할 수 있다.	1	O	100
B	터틀봇은 자동으로 방향 전환이 가능하다.	1	E, J	100
C	터틀봇의 최대 속도는 1m/s을 초과해서는 안된다.	1	A	100
D	중앙선을 넘지 않는다.	5	C	75
E	정해진 차로를 유지한다.	5	P, F	75
F	정지선 앞에서 3초간 정지한다	3	D, Q	90
G	장애물이 앞에 있으면 정지한다	2	H, T	100
H	정지 표지판 앞에 정지한다	2	L	100
I	종료 정지선 앞에 정지한다	2	M	100
J	터틀봇은 자신의 출발 차로에 맞는 코스를 선택한다	3	P, F	95
K	굴절코스를 주행한다	5	R	75
L	방향전환 코스를 주행한다	8	N	30
M	평행주차 코스를 주행한다	5	S, G	0
N	곡선코스를 주행한다	8	K	50
O	차단바를 인식한다	1	Start	100
P	정지선을 인식한다	1	D, Q	100
Q	중앙선을 인식한다.	3	C	75
R	벽을 인식한다.	3	B	삭제
S	장애물을 인식한다.	3	H, T	100
T	정지 표지판을 인식한다.	1	L	100

3.4 역할 수행 및 협업 도구 사용

3.4.1 역할 수행

이름	역할
김두영	PPT 제작, 결과보고서 작성
박진우	PPT 제작, 결과보고서 작성, 주행 관련 기능 구현
안대현	회의록 작성, 결과보고서 작성, 전반적인 기능 구현
허세진	회의록 작성, 결과보고서 작성, 전반적인 기능 구현

3.4.2 협업 도구

프로젝트 일정 및 자료를 관리하기 위한 협업 도구로 잔디를 사용하였다. 주차별 회의록 문서를 회의록 탭에 정리하고, 조사한 자료나 산출물을 업무 자료 탭에 정리하여 프로젝트 진행 상황에 맞추어 QA.01 문서와 같은 자료를 수정할 수 있었다.

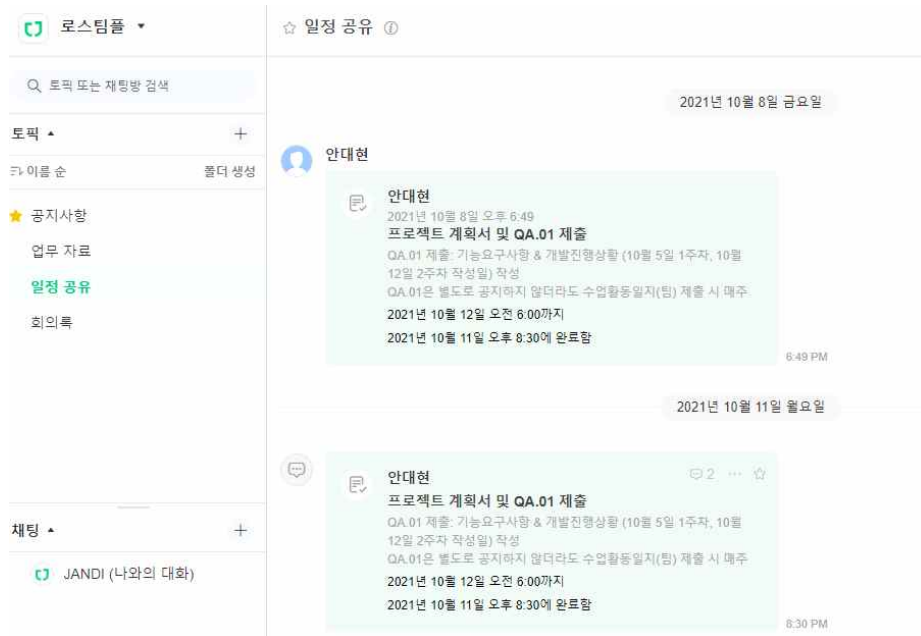


그림 74. 일정 공유

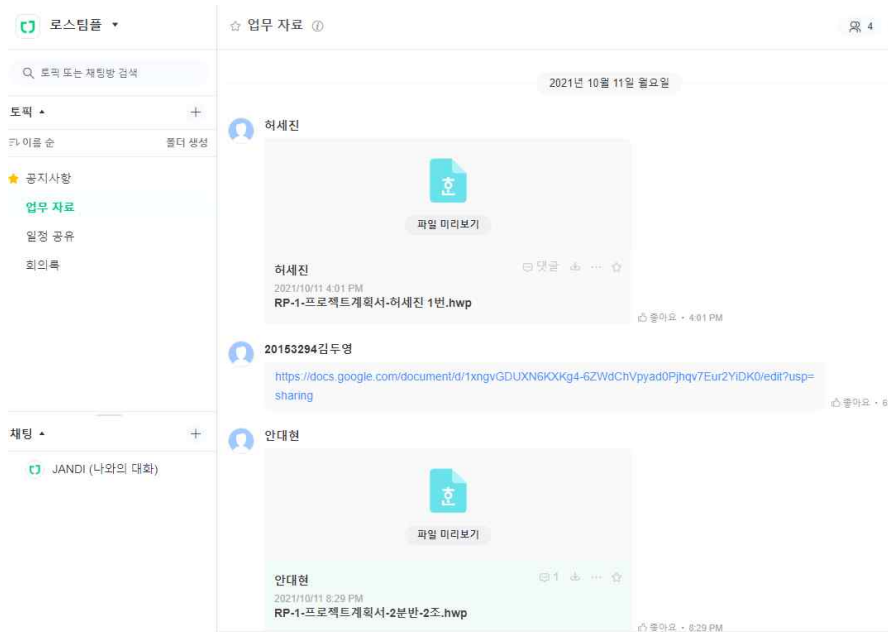


그림 75. 자료 정리

3.5 소스 코드 버전 제어 도구 사용

소스 코드 버전 제어 도구 : GitHub

팀원들이 공통적으로 접해본 소스 코드 버전 제어 도구가 깃허브이기 때문에 해당 프로젝트를 깃허브를 사용하여 진행하기로 하였다.

The screenshot shows the GitHub interface for the repository 'ads0070 / deu_car'. The repository is public and has 2 branches and 0 tags. The commit history is displayed, showing a series of commits by Heosejin98 and ads0070. The README.txt file is open, showing instructions on how to launch the project. The file list includes .idea, launch, scripts, world, ?, CMakeLists.txt, README.txt, gazebo_env.sh, and package.xml. The right sidebar shows the repository's About section, Releases, Packages, Contributors, and Languages.

File	Commit Message	Author	Time
.idea	delete testfile	Heosejin98	22 days ago
launch	Init	Heosejin98	27 days ago
scripts	12.03	Heosejin98	2 hours ago
world	Init	Heosejin98	27 days ago
?	Added stop line detection function	Heosejin98	22 days ago
CMakeLists.txt	Init	Heosejin98	27 days ago
README.txt	Add function of refractive course.	Heosejin98	14 days ago
gazebo_env.sh	Init	Heosejin98	27 days ago
package.xml	Init	Heosejin98	27 days ago

```

[How to launch]

$ roscd deu_car
$ source ../gazebo_env.sh
$ chmod +x ./scripts/blocking_bar_control.sh
$ chmod +x ./scripts/obstacle_spawn.py
$ roslaunch deu_car car_test.launch
$ rosrn deu_car car_state_machine.py
    
```

The screenshot shows the Git log interface, displaying a list of commits. The log includes the commit message, the author, the date, and the commit hash. The log is sorted by date, with the most recent commit at the top. The log shows a series of commits by Heosejin98 and ads0070, including fixes, cleanups, and new features.

Commit Message	Author	Date
Fix stop sign recognition function	ads0070	12/1/21, 3:28 PM
Fix blocking bar recognition function	ads0070	12/1/21, 3:01 PM
code cleanup	ads0070	11/30/21, 3:01 PM
Added stop sign function and departure lane change function	ads0070	11/30/21, 9:40 AM
code cleanup	ads0070	11/30/21, 9:39 AM
Added obstacle detection function and state	ads0070	11/29/21, 10:56 AM
Organizing files	ads0070	11/28/21, 4:42 AM
Organizing files	ads0070	11/19/21, 4:39 AM
Added function of refractive course.	ads0070	11/18/21, 4:19 AM
Add function of refractive course.	ads0070	11/18/21, 4:13 AM
test	014787410	11/04/21, 8:23 PM
init	ads0070	11/12/21, 7:06 PM
init	ads0070	11/12/21, 7:05 PM
Added stop line detection function	ads0070	11/11/21, 12:41 AM
delete testfile	ads0070	11/10/21, 10:16 PM
Update README.txt	Heosejin98*	11/7/21, 11:42 PM
Update auto_drive.py	Heosejin98*	11/7/21, 11:40 PM
add auto_drive.py Add-on: Blocker detection	sejin3319@naver.com	11/7/21, 11:38 PM
add auto_drive.py Add-on: Blocker detection	sejin3319@naver.com	11/7/21, 11:35 PM
Create test	Heosejin98*	11/5/21, 7:52 PM
init	ads0070	11/5/21, 7:14 PM

3.6 설계 구성요소

항목	내 용
분석	문제 정의와 작업 분해를 통하여 디자인 패턴을 적용하기 위한 요구사항을 도출할 수 있다. (분석 모델링은 없음)
	프로젝트의 주요 기능들을 구현하기 위해 각 기능별 흐름도를 작성하여 해당 유스케이스의 전체적인 흐름을 분석하고, 상황별 이벤트가 어떻게 일어나는지 정리하였다.
제작	분석과 설계 산출물을 사용하여 소프트웨어를 구현할 수 있다.
	흐름도와 시스템 유스케이스 다이어그램을 통해 SW 구현에 필요한 기술을 조사하였다. 인식 기능에 전반적으로 사용되는 OpenCV에서 지원하는 함수들의 사용법과 차선 검출을 위한 Canny Edge 알고리즘 등을 조사하고, 이를 적용하여 관련 기능을 구현하였다.
시험	제시된 기능/비기능 요구사항을 나열하고, 구현되었음을 테스트 케이스로 보여줄 수 있다.
	주행 상황별 요구사항을 테스트하기 위해 터틀봇 가제보 환경에서 주행 테스트를 반복 진행하였다.

3.7 현실적 제한조건

항목	내 용
생산성	<ul style="list-style-type: none"> ● Pycharm Professional IDE의 Diagram 기능을 사용하여 클래스 다이어그램을 확인하였다. ● 개발된 프로그램을 사용하면 가상 환경에서 자율 주행 테스트를 할 수 있기 때문에 위험 부담과 경제적 부담이 적고, 다양한 상황에 대한 테스트를 유동적으로 추가할 수 있기 때문에 생산적이다.
산업표준	<ul style="list-style-type: none"> ● 설계 단계에서 UML 클래스 다이어그램을 사용하여 요구사항 추적 가능성을 확보하였다.