제17회 임베디드SW경진대회
개발완료보고서 [자율주행 모형자동차]

1. 팀 정보

팀명	Eri-Car	팀장	강기훈
팀원	김지일	팀원	송희원
팀원	이지화	팀원	정은수

2. 개발완료보고서

1. 개요

1.1. 작품 개요

자율주행 자동차는 사람이 개입이 전혀 없이 주행이 가능한 자동차이다. 하지만 자동차는 본래 인간이 운전하기에 최적화된 형태이므로, 본 팀에서는 이번 개발을 통해 운전하면서 발생하는 많은 변수에 대해 사람처럼 적절한 대응을 하면서 주행하는 자율주행 자동차를 선보이고자 한다.

1.2. 개발 목표

자율주행 자동차는 사람이 없는 대신 다양한 센서 및 카메라를 이용하여 발생하는 변수를 파악하고 판단하게 된다. 이때 대표적인 변수들이 대회의 미션이므로, 미션을 해결해가며 발생하는 변수들에 대한 대책을 강구해볼 수 있다.

기존의 복잡한 프로세스에 각각의 미션 스레드 및 함수를 추가하면, 가독성과 충돌 등의 문제가 발생할 수 있다. 따라서 함수 헤더를 따로 구성하고, 변수 개수를 최소화하며, 불필요한 스레드를 제거하는 등 다양한 방법으로 가독성을 높이고 오류를 줄이도록 한다.

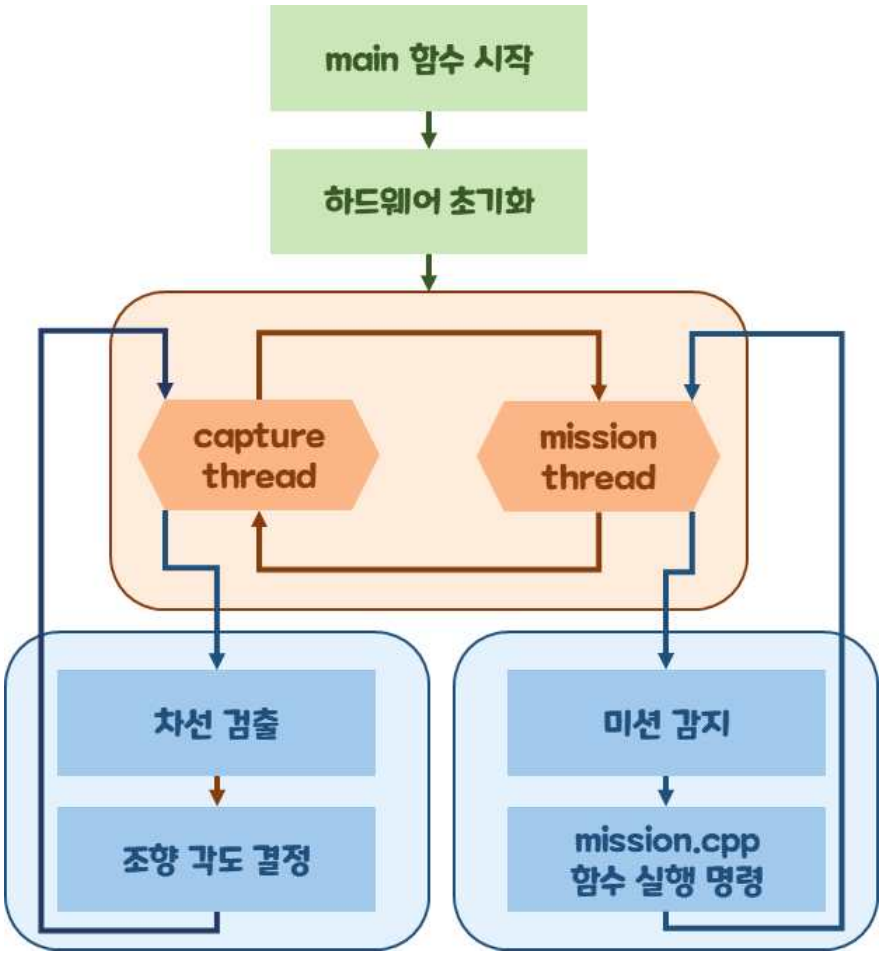
2. 개발 환경 설명

2.1. Software 구성

자율주행 자동차의 Software는 C파일(혹은 C++파일)과 헤더파일 20개로 구성이 되어있다. 대개 예제에서 사용되었던 파일들을 이용하여 Makefile을 구성하였지만, dump를 통한 캡쳐 함수 및 스레드 등과 같이 주행에서 불필요한 작업이라고 생각되는 것은 가능한 한 제거하였다. 미션 함수는 mission.cpp으로 분리하여, 작업은 주로 main.c, exam_cv, mission.cpp에서 이루어졌다.

Mission 클래스에 각각 미션에 대한 변수를 두었으며, main.c의 미션을 감지하는 스레드를 통해서 변수 값을 바꾸게 된다. 이 값으로 인해 mission.cpp의 함수가 실행되어 미션을 수행하게 된다.

2.2. Software 설계도(흐름도 및 클래스 다이어그램 등 (개발언어에 따라 선택))



2.3. Software 기능 (알고리즘 설명 포함)

[main.c] : 자동차의 초기 세팅 CarcontrolIInit()을 한 뒤, Steering 서보와 CameraX 서보를 1500으로 초기화시킨다. 이때 CamerY 서보는 1650으로 두어 약간 아래를 바라보도록 한다. 마지막으로 기본적으로 속도 제어를 하기 때문에 PositionControl은 UNCONTROL, SpeedControl은 CONTROL로 세팅하고 Gain값을 설정한 뒤, 스레드가 시작되기 전까지 1초를 기다린다. 만약 1초를 기다리지 않는다면 CameraY가 내려가는 동안 영상처리가 되어버려 차선위치 저장 값이 이상해지기 때문이다.

스레드는 capture_thread와 mission_thread가 있다. capture_thread는 영상처리 스레드로, 차선 인식을 통해 조향각과 속도를 메인으로 제어한다. mission_thread는 미션판단 스레드로, 센서를 통해서 상황을 판단하고 Mission 클래스의 미션 변수를 제어한다.

미션을 감지하면 Mission 클래스 변수를 1로 바꾸고, 이에 따라서 if문을 통해 mission.cpp의 미션 수행 함수를 실행하도록 했다. 마지막으로 signal_handler를 통해, 무한루프인 프로그램을 'Ctrl + c'으로 종료할 때 정상종료 되도록 했다.

[exam_cv.cpp] : main.c 스레드 내부의 hough_transform 함수로 인해, exam_cv.cpp의 OpenCV_hough_transform 함수가 계속적으로 실행된다. 이는 예제의 허프변환 함수였지만, 허프변환을 하지 않고 ROI설정 및 차선인식을 하도록 구성하였다.

[mission.cpp] : Mission 클래스 변수에 의해 Mission_thread로부터 호출되는 미션 주행 함수들을 모두 포함하고 있다. 함수는 실행된 뒤, 다시 Mission 클래스 변수를 초기화하여 정상상태로 만들어준다.

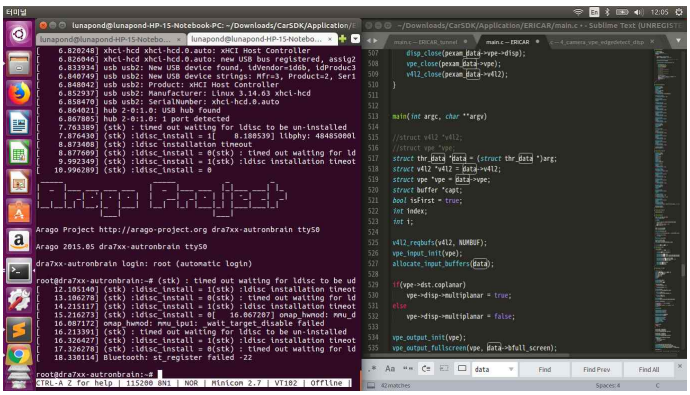
[car_lib.c] : 자동차 제어 함수를 포함하고 있다. 주차 미션과 회전 교차로 미션 등에서 EncoderCounter_Read와 EncoderCounter_Write를 통해 엔코더를 이용하기도 한다. CarLight와 Alarm은 자동차의 상태를 피드백하는 역할을 한다.

2.4. 프로그램 사용법 (Interface)

프로그램 상에서 printf 함수를 통해 steer, WisY, enco 등과 같은 디버깅이 필요한 변수를 출력하였다. 하지만 이는 시리얼 케이블을 연결해야만 값을 받아올 수 있다. 따라서 상태를 피드백 하는 경우에는 CarLight나 Alarm으로 확인할 수 있도록 했다.

자동차의 디스플레이에는 ROI를 나누어, 하단부는 이진화 후에 차선을 인식하여 좌측 차선은 녹색 직선을. 우측 차선은 파란 직선을 오버레이하였고, 상단부는 신호등과 비상정지 표지판을 인식하여 사각형을 오버레이하였다. 그리고 전체 영상처리 시간을 draw_operatingtime 함수를 통해 표시하였다.

2.5. 개발환경 (언어, Tool, 사용시스템 등)



▲ ubuntu 16.04 Terminal(minicom) and Sublime Text 3

ubuntu 16.04 LTS에서 작업하였으며, 텍스트 편집기 Sublime Text 3를 이용하였다. 예제 파일을 최대한 이용하기 위해 C 및 C++로 코딩하였으며, Makefile의 참조경로 CP_PATH를 SDK에 포함되어 있었던 CrossCompiler로 두어 opencv 등 별도의 헤더를 추가로 설치하지 않고도 컴파일 할 수 있었다.

3. 개발 프로그램 설명 (최대한 자세하게 기술)

3.1. 파일 구성

[main.c]

① capture_thread : 영상에 hough_transform을 하여 vpe 클래스의 disp에 저장하고, 자동차의 디스플레이에 결과를 출력한다.

② mission_thread : 센서를 통해 미션을 감지하고, Mission 클래스 변수를 통해 미션 함수를 실행한다.

③ hough_transform : 차선 감지와 함께 steer 변수를 통해 조향한다. 이때 hough_driver 변수는 터널 미션을 제외한 모든 상황에서 조향을 하도록 한다. 터널 미션 중에서는 PSD센서를 통해 조향하기 때문이다.

[exam_cv.cpp]

① OpenCV_hough_transform : ROI 영역을 이진화한 뒤, Canny를 적용한다. 그리고 차선을 인식하여 steer를 계산한다.

[mission.cpp]

① mission_sudden_stop : 표지판을 감지하여, mission.sudden_stop 변수가 1이 되면 실행된다. 카메라에 표지판이 사라질 때까지 정차한다.

② mission_long_parking : 수평주차장 크기를 인식하여, mission.long_parking 변수가 1이 되면 실행된다. 수평주차를 한다.

③ mission_short_parking : 수직주차장 크기를 인식하여, mission.short_parking 변수가 1이 되면 실행된다. 수직주차를 한다.

④ mission_rotation : 진입선 감지변수 Wline이 5 이상이 되어, mission.rotation 변수가 1이 되면 실행된다. 주행장애물을 감지하여 회전교차로를 통과한다.

⑤ mission_tunnel : 모든 PSD센서가 일정거리 이하로 감지되어, mission.tunnel 변수가 1이 되면 실행된다. 2번 PSD센서를 이용하여 터널을 통과한다.

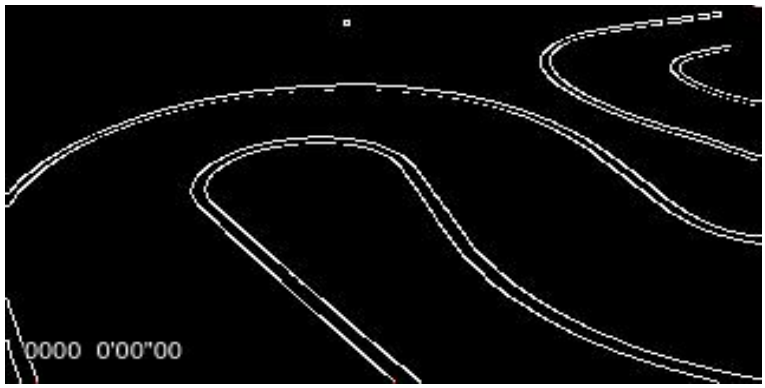
⑥ mission_pass : 1번 PSD센서가 자동차를 감지하여, mission.pass 변수가 1이 되면 실행된다. 자동차를 추월한다.

⑦ mission_signal_light : 진입선 감지변수 Wline이 5 이상이 되고 mission_rotation 변수가 0이 아닐 때, mission.signal_light 변수가 1이 되면 실행된다. 신호등의 방향을 판단하여 올바른 종료지점까지 주행 후 정지한다.

⑧ mission_detect : mission.cpp에 들어있는 모든 mission함수를 실행시키기 위해 해당 미션을 인식하기 위한 함수이다. 각각의 코스에 있는 특정 조건을 충족하면 mission.(해당미션)이라는 변수를 ++하여 해당 미션이 실행되도록 한다.

3.2. 함수별 기능

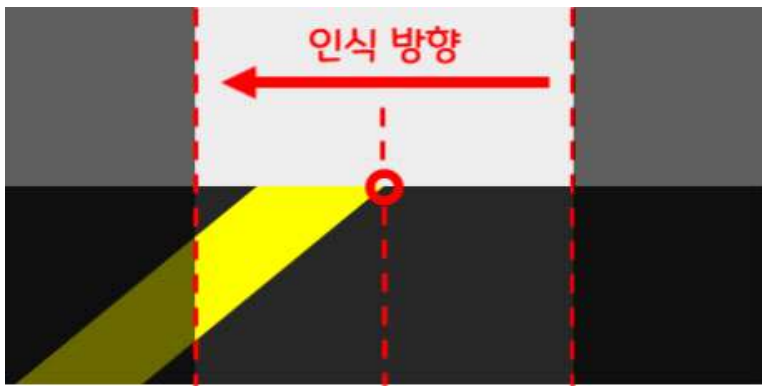
[기본 주행] : 차선 인식은 기존의 exam_cv.cpp 내부의 OpenCV_hought_transform 함수를 변형하도록 했다. 먼저, 차선을 정확하게 인식하기 위해서 '밝기' 요소를 조절해야 했으므로, BGR을 HSV로 변환하였다. 그리고 노란색에 대한 HSV 값을 이용하여 이진화를 시키고, 테두리를 검출하기 위해 Canny 변환을 하였다.



▲ 연습 경기장 이진화 및 Canny

이때 주행을 위해서 화면 전체를 위와 같은 작업을 거치는 것은 시간과 리소스의 낭비이다. 따라서 ROI를 설정하여 하단 부분만 처리를 하도록 했다. 이때 main.c의 roi_h라는 변수를 통해 그 범위를 유동적으로 변화시킬 수 있도록 했다.

차선 인식을 위해 사용한 방법은 '이전 차선 범위 내 재탐색'이다.



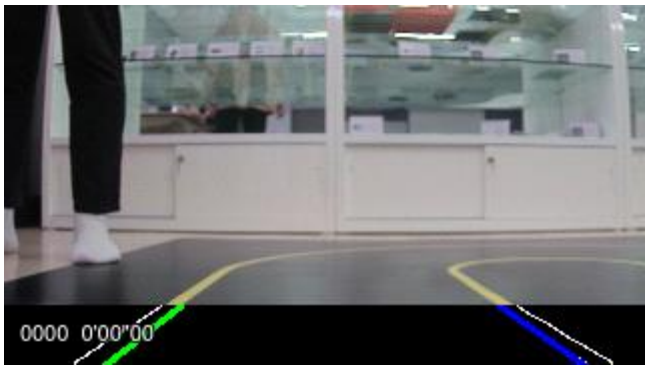
▲ 이전 차선 범위 내 재탐색

먼저, 위의 그림에서 동그라미 친 부분을 이전 차선의 좌표로 둔다. 그리고 다음 프레임에서 이 좌표를 기준으로 '중앙에서부터 가장자리 방향으로' 일정 범위만큼만 탐색하여 인식되면 그 점을 새로운 차선의 좌표로 둔다.

위와 같은 알고리즘이 가능한 이유는, 한 프레임 내에서 해당 점이 이동할 수 있는 범위에는 한계가 있기 때문이다. 또한 방향을 위와 같이 설정하면 범위 내에 바깥 차선이 있어도 무시할 수 있다. 더 나아가 카메라에 여러 차선이 검출

되어도 기존의 주행 차선만 유도할 수 있다.

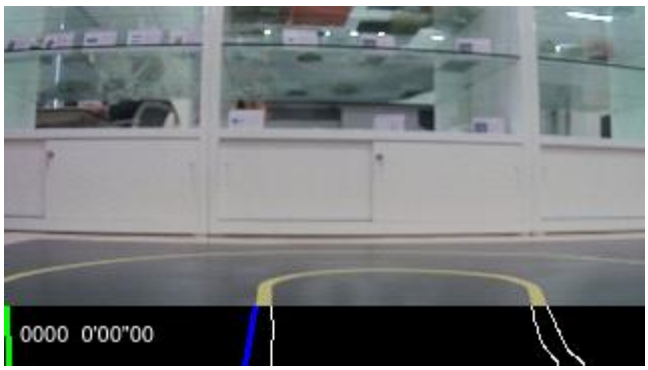
다음은 실제 차선에 적용한 사진으로, 좌측 차선은 녹색, 우측 차선은 파란색으로 오버레이하였다.



▲ 직진 차선 인식

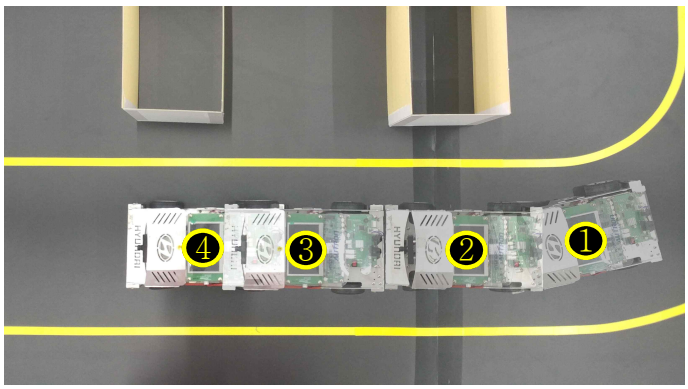
▲ 우회전 차선 인식

따라서 좌측 차선 검출 결과인 녹색 직선이 위와 같이 가장자리로부터 멀어지면 우회전임을 판단할 수 있었다.



▲ 다른 차선이 검출되어도 인식하지 않음

[주차] : 주차공간의 감지는 자동차의 우측 상단에 위치한 PSD센서로 이루어진다. PSD센서에 일정 거리 이내의 벽이 감지된 뒤 엔코더 값을 확인한다. 이후 벽이 감지되지 않았을 때, 그때부터 엔코더 값을 초기화하고 다시 누적하여 측정한다. 일정 거리 이내의 벽이 감지되면 누적되어있는 엔코더 값을 확인한 뒤, 그 값에 따라 수직주차(mission_short_parking()) 혹은 평행주차(mission_long_parking())를 시작한다. 이 값이 조건에 포함되지 않으면 엔코더 값을 초기화한다. 다음은 후진 주차공간의 감지와 주차 순서이다.



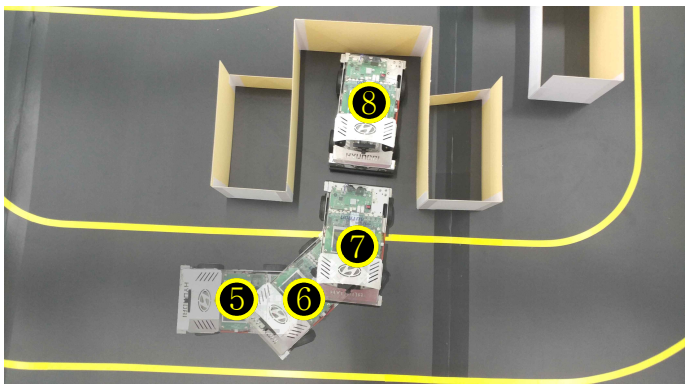
▲ 수직주차 ① ~ ④

① 벽 감지 - 누적 엔코더 값 확인 후 엔코더 초기화

② 벽 미감지 - 엔코더 누적 시작

③ 벽 감지 – 누적 엔코더 값 조건에 의해 수직주차 시작
(mission.short_parking = 1)

④ 우측 앞쪽 PSD 가 벽 감지할 때까지 직진



▲ 수직주차 ⑤ ~ ⑧

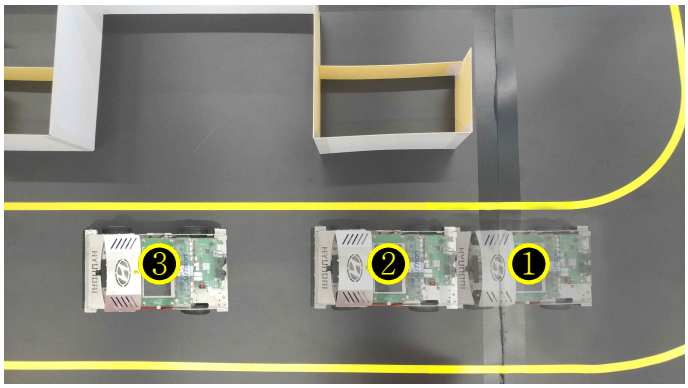
⑤ 우측 뒤쪽 PSD가 벽 감지할 때까지 직진

⑥ Steer 방향을 최대한 오른쪽으로 튼 뒤 후진

⑦ 좌측 뒤쪽 PSD가 벽 감지하면 Steer를 가운데로 맞춘 후 후진

⑧ 후면 PSD가 벽 감지하면 주차 완료

다음은 평행 주차 공간의 감지와 주차 순서이다.

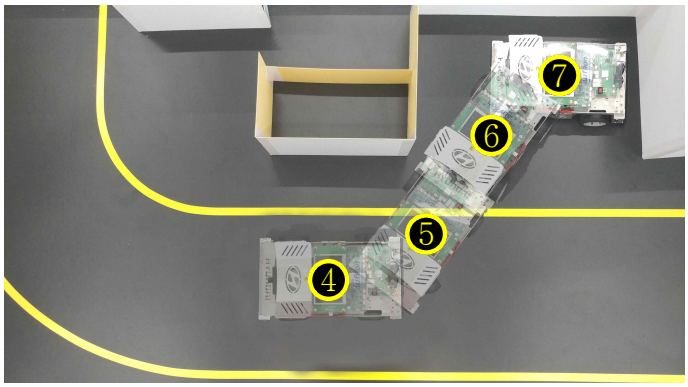


▲ 수평주차 ① ~ ③

① 벽 감지 - 누적 엔코더 값 확인 후 엔코더 초기화

② 벽 미감지 - 엔코더 누적 시작

③ 벽 감지 - 누적 엔코더 값 조건에 의해 수평주차 시작
(mission.long_parking = 1)



▲ 수평주차 ④ ~ ⑦

⑤ 우측 앞쪽 PSD가 벽 감지할 때까지 직진 후 일정 엔코더만큼 직진

⑥ Steer 방향을 최대한 오른쪽으로 튼 뒤 우측 뒤쪽 PSD 감지할 때까지 후진

⑦ Steer 방향을 가운데로 맞춘 후 우측 앞쪽 PSD 감지할 때까지 후진

⑧ Steer 방향을 왼쪽으로 튼 뒤 후면 PSD 감지할 때까지 후진하여 주차 완료

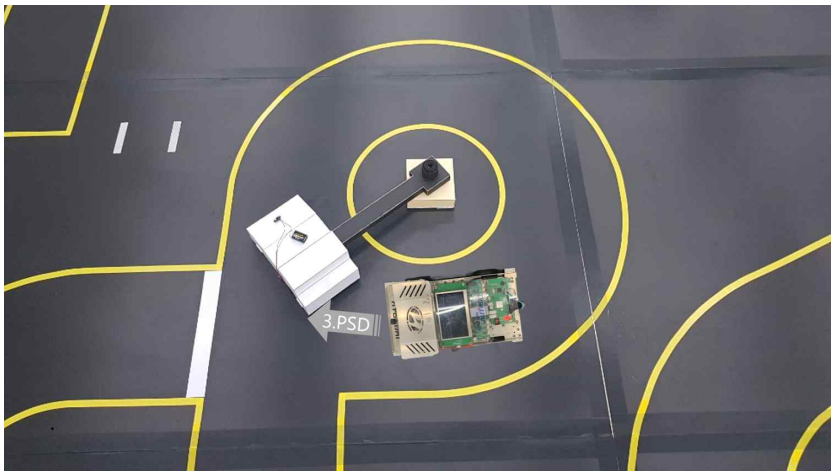
주차가 완료되면 주차 과정에서 이용한 조건을 역순으로 판단하며 주차 공간을 빠져나온다. 주차 공간을 빠져나왔다고 판단되면 주행 스레드로 들어간다.

[회전교차로] : 회전교차로라는 미션을 수행할 때는 가장 중요하게 쓰이는 두 가지의 센서가 있다. PSD 센서와 Line 센서가 바로 그 두 가지 인데, 이는 회전교차로라는 미션을 detect할 때 회전교차로 진입부에 그려진 흰 선을 인지하고, 전방과 후방에 돌고 있는 차량을 인식하기 위함이다. 그렇기에 수행해야할 미션이 회전교차로라는 것을 인식하기 위한 가장 큰 전제조건을 aWall == 0 && mission.signal_light < 1로 설정하였는데, 이는 현재 오른쪽에 벽이 존재하지 않고,(주차장 코드와의 오류를 방지하기 위함.) 신호등이 실행되기 전에만 detect를 실행할 수 있게 하기 위함이다.

가장 큰 전제 조건을 통과하면, 비트 연산을 통해 총 7개의 line 센서부가 검은색과 흰색을 인식하고 인식한 결과를 바탕으로 회전교차로가 실행된 적이 없으면 회전교차로를 실행된 적이 이미 있다면 신호등 미션을 수행하게 된다.

이렇게 미션을 실행하게 될 때 첫 번째로 인식하는 것은 전방 PSD 센서이다. 아래의 그림처럼 전방에 차량이 감지되기 시작하고, 대상차량이 회전교차로를 아래와 같이 돌면서 주행차량과 멀어지면 대상 차량이 도는 방향을 따라 앞으로 직진하게 된다. 이때 주행알고리즘을 사용하여 앞으로 나아가게 되는데, 대상차량이 앞에 붙어있으면 카메라가 주행 차선을 인식하기 어려울 수 있고, 충돌방지 위험을 방지하기 위해 직진 후 회전을 시작하면(주행 알고리즘을 통해 앞바퀴의 Servocontrol의 steer가 회전된다고 감지되면 회전으로 판단한다.) 주행차량은 잠시 정지한다.

그렇게 정지하게 된 후 대상차량이 한 바퀴를 돌고 주행차량의 후방 아래 사진과 같이 PSD 센서를 감지하게 되면 다시 주행을 시작한다. 이때 안정적인 속도를 유지하면서 후방 대상 차량과의 충돌을 막기 위해 후방차량과의 거리에 비례하며 가까워질수록 속력이 증가하는 알고리즘을 채택하였다.



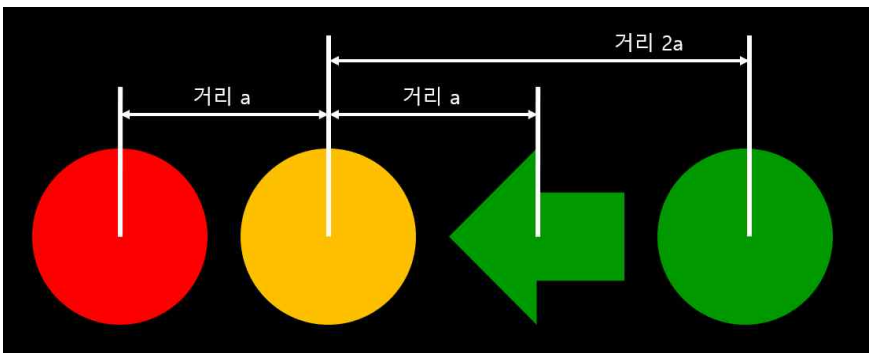
회전을 하는 중에 흰 선을 인식하지 못해 우회전을 해버리는 경우가 발생하기도 하여 회전교차로 미션을 수행하는 동안에 흰 선도 노란색과 같은 차선으로 인식하는 'WisY'함수를 제작하여 회전 후 교차로를 빠져나올 수 있게 프로그램을 설계하였다.

[터널] : 양 측면의 PSD센서가 벽을 감지하면 터널에 진입한 것으로 인식(mission.tunnel = 1)하고 터널 주행을 시작한다. (mission_tunnel()) 터널 내부에서는 우측 앞쪽 PSD센서를 이용하여 오른쪽 벽을 따라간다. 터널 주행 후 양 측면의 앞쪽 PSD센서가 벽을 감지하지 않으면 터널이 끝난 것으로 보고 일반 주행 스레드로 들어간다.

터널 주행 시에는 PSD센서로 측정한 차량과 벽 사이의 거리에 일정 값을 곱하여 Steer 값을 조정한다.

[신호등] : IR 센서가 두 번째 정지선을 감지(mission.light_signal = 1)하면 차량 정지 후 신호등을 감지한다. (mission_light_signal())

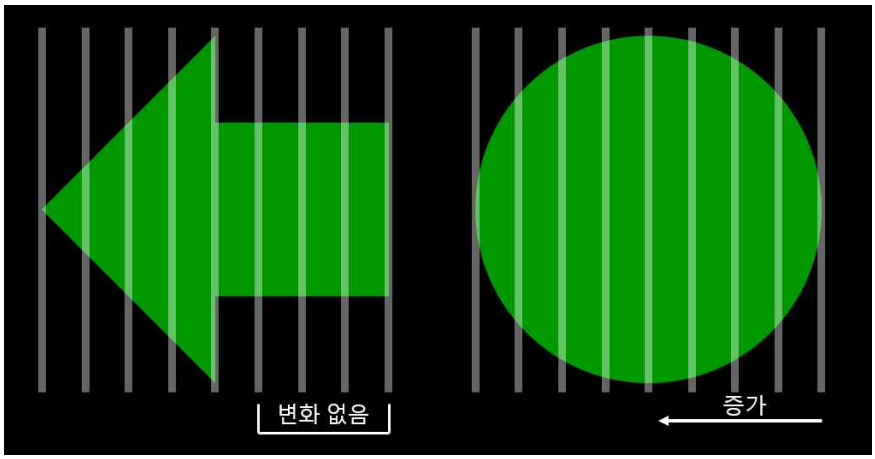
개발 초기에 신호등 미션을 수행할 때에는 청색 화살표와 청색 원형 신호의 구별이 어려워 적생, 황색, 청색 화살표, 청색등을 모두 인식하여 구별하는 방법을 이용하였다. 먼저, 적색등이 감지되면 적색등이 화면의 좌측에 위치하도록 카메라의 각도를 돌린다. 그 후 적색등의 중심 좌표를 찾아 저장한 뒤 황색등의 중심 좌표 또한 찾아낸다. 이 두 중심 좌표 사이의 거리를 황색등과 다음 청색등의 거리와 비교하여 청색등의 위치를 구별해 내게 된다. 아래 그림에서 황색등과 청색등의 거리가 a이면 화살표, 2a 이면 원형 신호로 판단하여 움직이게 된다.



위와 같은 방법을 사용할 경우 차량이 정지선에 도착하였을 때 적색등이 아닌 다른 등이 점등되어있을 경우 다음 적색등이 점등되기를 기다려 적색등의 중심 좌표

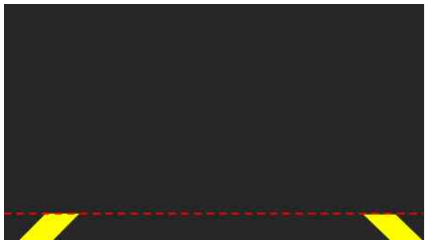
를 설정해 주어야 한다는 문제가 있었다. 즉 차량이 정지선에 도착하였을 때 청색 등이 점등되어있어도 바로 출발하지 못한다는 것이다. 이를 해결하기 위해 두 청색등의 모양을 구별해 내는 방법을 고안하였다.

두 청색 등에 각각 9개의 세로줄을 그어 청색 영역과 겹치는 부분의 길이를 확인해 보았을 때 화살표의 경우 우측 4개의 세로줄의 겹치는 길이는 거의 변화가 없음을 확인할 수 있다. 그러나 원형 등의 경우 가장 우측 세로줄과 가운데 세로줄은 1.5배 이상의 길이 차이를 보이게 된다. 이를 이용하여 화살표등과 원형 등의 모양을 구별해 낼 수 있다.

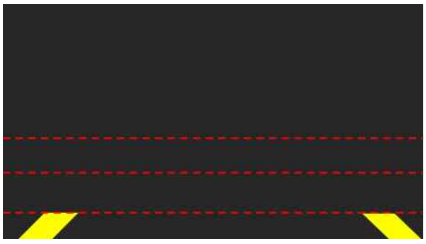


[고가도로] : 내리막이 아닌 고가도로 위에서는 평지에서와 같은 주행으로 구간을 통과한다. 이 고가도로 위에서는 일반 주행에 사용하는 ROI 외에 조금 더 높은 구간을 선택한 ROI를 만들어 이 범위 내에 노란 선이 인식되지 않으면 내리막 구간으로 판단하여 속도를 줄인 후 일정구간 직진한다. 이때 고가도로에서 추가로 만든 ROI는 주행 즉, steer에 영향을 주지 않으며 내리막을 모두 내려온 뒤 삭제한다.

▲ 고가도로 주행 시 ROI



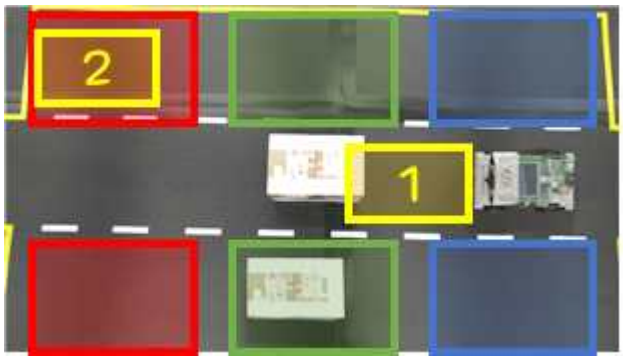
▲ 일반 주행 시 ROI



▲ 내리막이 가까워졌을 때

[차로추월] : 터널을 통과하고 난 뒤에는 (mission.pass = 1) 노란색과 흰색에 대하여 모두 이진화한 뒤 라인 검출을 한다. (mission_pass()) 만약 흰색 라인이 검출되기 시작하면 차로추월 구간에 진입한 것으로 판단하고 PSD센서로 전방의 장애물 유무를 판단한다. 만약 전면 장애물이 감지되면 PSD센서와 카메라로 좌,우측 차선에 추월차량이 있는지 확인한다. 추월차선을 확인했으면 추월차량이 없는 곳으로 차선변경을 시작한다. [기본 주행]방식으로 차선을 변경하고, 전면 노란차선이 카메라_이미지_Y값 160에 오거나 전면 장애물이 발견 될 경우 다시 중앙차선으로 변경한다. 이때도 역시 [기본 주행]방식으로 차선을 변경한다.

다음은 추월차선의 차량 감지 설명이다.



▲ 추월차량 위치파악하기

① 파란영역 - 중앙차선에 추월차량을 감지하면 PSD센서로 주행차량의 바로 좌측 우측에 2번째 추월차량이 존재하는지 확인한다.

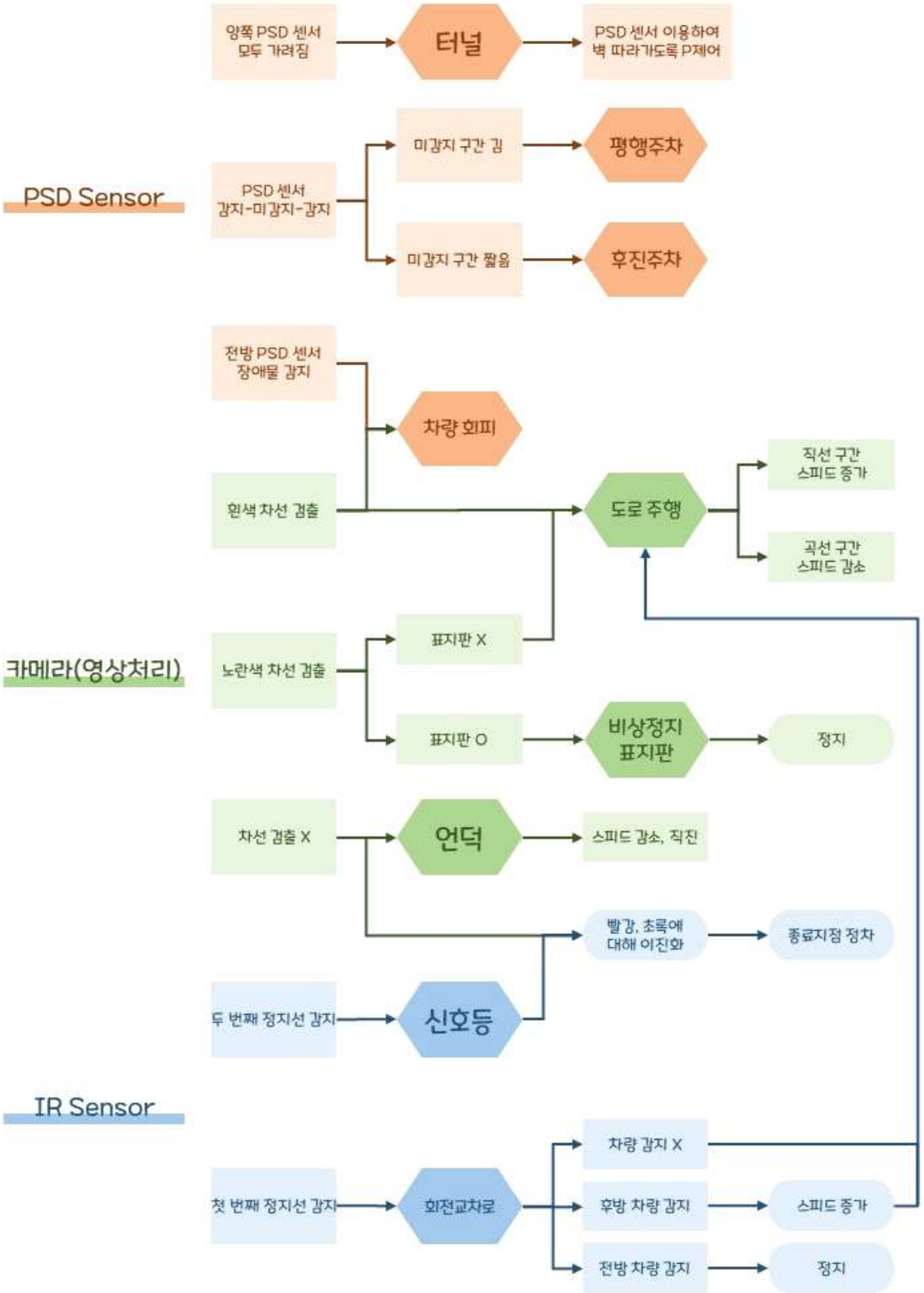
② 녹색영역 - ①(파란영역)에서 차량이 발견 안 되면 카메라를 좌(우)측으로 꺾어서 좌(우)측 차선에 또 다른 추월차량이 존재하는지 확인한다.

③ 빨간영역 - ②(녹색영역)에서 차량이 발견 안 되면 주행차량을 약간 후진시킨 뒤 녹색영역의 차량 존재 여부를 확인하듯이 빨간 영역에 또 다른 추월차량이 있는지 확인한다.

노란영역 - 중앙 추월차량이 1번 자리에 있고 2번째 추월차량이 2번 자리에 있으면 주행차량의 위치를 바꾸지 않는 이상 2번 차량의 존재여부를 알 수 없다. 하지만 두 추월차량의 간격이 넓어 그 간격사이로 주행차량이 문제없이 지나갈 수 있다.

3.3. 주요 함수의 흐름도

PSD센서, 카메라, IR센서에 대해 미션 판단 및 진행을 다음과 같이 흐름도로 구성해보았다. 미션 진행에 따라 실행될 함수는 '3.2. 함수별 기능'과 같다.



▲ 미션 판단 및 진행 흐름도

3.4. 기술적 차별성

1) Mission 클래스로 각각 미션에 대한 변수를 따로 두어, 처음부터 시작하는 것이 아닌 원하는 부분부터 주행이 가능하다.

2) 차선 내 범위 재탐색 방식을 통해, 카메라에 많은 차선이 검출되어도 기존의 주행 차선만을 유도한다.

3) 많은 계산 요소(재탐색 범위, ROI 범위 설정, steer 반응 구간 등)를 변수로 두어, 실험값 선별을 하는데 유리하도록 모델링하였다.

4) steer를 선형적으로 계산하여, 곡선 구간을 부드럽게 주행할 수 있다.

5) steer에 따른 속도 또한 선형적으로 계산하여, 기본적으로 빠른 속도를 갖추면서도 곡선 구간에서도 안전한 주행을 할 수 있다.

4. 개발 중 장애요인과 해결방안

4.1 주차

[장애요인] : 주차 공간을 인식할 때 차선과 주차장 사이의 거리가 정해져 있지 않아 벽을 인식하는 기준 거리를 잡기 어려웠다. 또한 매 시행마다 주차 공간을 측정하는 엔코더 값이 달라져 오차 범위가 컸다.

[해결방안] : 벽을 인식하는 기준 거리는 경기장에 설치된 다른 구조물을 인식하지 않으며 주차공간의 뒷벽을 인식하지 않는 정도로 최대한 멀리 설정하였다. 주차 공간을 측정하는 엔코더의 범위 또한 넓게 설정하여 허용 오차 범위를 늘려 주었다.

[장애요인] : 주차 공간 인식 후 주차 동작까지 엔코더로 제어하기에는 매 실행마다 결과가 달라지고 정확하지 않았다. 또한 차선에 대한 주차공간의 위치가 명확하지 않기 때문에 상황에 맞는 제어 방법이 필요하였다.

[해결방안] : 6개의 PSD센서를 최대한 많이 이용하기로 하였다. 주차공간의 모서리를 양 측면의 PSD센서로 인식하고 주차 완료 여부를 판단할 때 전면, 후면의 PSD센서를 이용하여 자동차의 위치를 제어하였다.

4.2 터널

[장애요인] : 터널 구간을 통과하는 초기 계획은 전면, 후면을 제외한 4개의 PSD센서를 모두 활용하여 터널 양쪽 벽과의 거리와 방향 틀어짐을 모두 제어하는 것이었다. 이때 벽과의 거리 유지와 방향 유지를 위한 두 판단 사이에 충돌이 있어 4개의 PSD센서를 모두 이용하는 것은 어렵다고 보았다.

[해결방안] : 터널을 부드럽게 통과하지 못하는 것을 감안하고 우측 앞쪽의 PSD 센서 1개만을 이용해 보기로 하였다. 1개의 PSD센서 측정값에 대해 비례 제어를 하자 정확하고 부드러운 터널 통과가 가능해졌다.

4.3 회전교차로

[장애요인] : 주행차량은 노란색 선만을 차선으로 추출하여 주행하며 미션을 수행한다. 그로 인해 (탈출하는 과정에서) 노란색 차선이 끊기는 회전교차로와 흰색 차선이 있는 추월구간에서 문제가 발생하였다. 하지만 이진화 과정에서 흰색을 노란색으로 인식하면, 회전교차로 진입조차 차선으로 보아 진입할 수 없었다.

[해결방안] : 흰색 차선도 노란 차선으로 인식하게 할 수 있도록 WisY 변수를 생성하였다. 회전교차로의 경우, 중간지점부터 빠져나갈 때까지 WisY를 1로 유지하여 차량이 교차로 출구를 향할 수 있도록 하였다. 또한 터널을 통과한 뒤, WisY를 다시 1로 설정 해주어서 바로 다음의 추월구간 흰색 차선을 인식할 수 있게 하였다.

▲ 흰색 검출 가능 (WisY = 1)

▲ 흰색 검출 불가 (WisY = 0)

4.4 차로추월

[장애요인] : 차선을 바꿀 때 엔코더만을 이용할 경우 바퀴의 미끄러짐으로 인해 일정한 모션을 보여주지 못한다. 특히 제일 큰 요인은 앞바퀴 조향을 할 때 조향축이 이동을 하여 뒷바퀴 슬립을 유도되는 점이다. 이로 인해 자동차는 정확히 옆 차선으로 이동을 못하고 흰색차선에 걸치거나 노란차선을 밟게 된다. PSD센서 역시 안정적으로 사용할 환경이 제공되지 않아 도움이 되지 못한다.

[해결방안] : 카메라를 이용하여 이를 해결했다. 기존에 쓰던 [기본 주행]함수를 사용하여 새로운 함수를 만들 필요가 없어 메모리를 아꼈다. 자동차가 넘어가고 싶은 차선 쪽으로 머리를 집어넣고 카메라를 주행하고 싶은 방향으로 돌리면 차선변경을 위한 준비는 끝난다. 여기서 [기본 주행]함수를 사용하여 차선을 따라가면서 카메라를 계속 차선을 향하게 하면 자동차는 안정적으로 차선변경을 하게 된다.

[장애요인] : 우리팀의 주행차량은 점선같이 차선이 떨어져 있으면 주행에 방해가 된다. 차선이 끊어져 있으면 끊어진 차선쪽으로 새로운 길이 있다고 판단을 내릴 수 있기 때문인데 점선간격이 넓은 부분을 만나게 되면 자동차는 그 점선 간격 사이로 비집고 들어가게 된다.

[해결방안] : 사람이 점선을 인식하는 방식으로 접근했다. 사람도 점 하나만 찍혀 있으면 이

를 선으로 인식하지 않고 점이 여러 개 이어져 있어야 비로소 점선이라고 인식한다. 따라서 우리는 더 멀리있는 점선까지 보기 위해 화면의 절반 이상을 점선을 따라가기 위한 roi영역으로 잡았다. 그렇게하고 제일 위에 있는 점선과 제일 뒤에 있는 점선을 이어 하나의 흰색 진선 차선을 그려주고 [기본 주행]함수로 차선을 따라 갈 수 있게 하였다.

4.5 신호등

[장애요인] : 기존 주행에서 곡선 차로가 있을 경우 휘어진 방향과 반대 방향에 있는 차선을 보고 바퀴 조향각을 결정할 수 있었다. 그러나 신호등 미션의 경우 휘어진 방향과 반대 방향에는 차선이 매우 멀리 존재하여 카메라가 이를 확인하지 못하고 차로를 이탈하는 경우가 발생하였다.

[해결방안] : 이를 해결하기 위해 신호등 방향을 확인한 뒤 그 방향으로 카메라 x 조향을 꺾어 회전하고자 하는 방향에 있는 차선을 확인하고 따라가는 방식을 이용하였다.

4.6 언덕

[장애요인] : 언덕 구간의 경사도가 생각보가 커서 자동차가 미끄러지는 현상이 발생하였다. 자동차의 속도를 줄이거나 높여보아도 해결되지 않아 언덕 구간의 외벽에 부딪히거나 경로를 벗어나는 일이 일어났다.

[해결방안] : 실제 자동차를 운전할 때 미끄러짐을 최소화하기 위한 '카운터 스티어링'이라는 운전법이 있다. 카운터 스티어링이란 자동차의 뒷바퀴가 미끄러지는 쪽으로 핸들(스티어링 휠)을 돌려 차량을 제어하는 운전 기술이다. 회전관성체의 반작용 원리를 이용한 선회 법으로, 이 기술을 이용하면 뒷바퀴의 미끄러짐을 막고 스핀을 방지할 수 있다. 뒷바퀴가 오른쪽으로 미끄러지면 스티어링 휠을 오른쪽으로, 왼쪽으로 미끄러지면 왼쪽으로 꺾으므로 '역핸들'이라고도 불린다.

이 '카운터 스티어링' 기법을 모형자동차에 적용시켜 2,6번 PSD 센서를 이용해 벽과의 거리 즉, 미끄러진 정도와 방향을 파악하고 그 방향대로 스티어링을 조절하여 모형자동차의 미끄러짐이 완화되도록 하였다.

5. 개발결과물의 차별성

대회 미션의 경우 미션의 대략적인 순서나 위치가 정해져 있어 이를 기억하면 미션을 쉽게 해결할 수 있지만 실제 자율주행차가 달려야 하는 상황은 그렇지 않다. 따라서 미션의 순서나 위치에 관계없이 미션을 파악하는 다양한 조건을 두어 어떤 상황에서든 미션을 수행할 수 있도록 하였다.

실제 주행 환경에서는 자동차가 달리고 있는 차선뿐만 아니라 다른 많은 차선이나 교차선등이 존재한다. 이 때 교차선이나 인식하지 않아야 하는 다른 차선들을 거르기 위해 범위 내 차선 재탐색 방식을 채택하였다. 이는 기존에 인식 중이던 차선 외에 다른 차선이 카메라에 잡히더라도 기존 차선의 범위 내에 있지 않기 때문에 원하는 차선만 잡아낼 수 있는 방식으로 자동차가 차선을 벗어나는 일이 절대 일어나지 않도록 돕는다.

이 외에도 흔히 쓰는 RGB색상표 대신 HSV색상표를 사용하여 실제 경기장에서 영향을 줄 빛에 의한 명도를 쉽게 조절할 수 있도록 했으며, 비순차적 미션 스레드를 구성하여 실제 경기장이 예시처럼 나오지 않더라도 미션을 판단하여 수행할 수 있게 했다.

6. 단계별 개발계획 및 실제 참여인원 및 업무 분장

6.1. 업무분장

No.	구분	성명	소속	학과	담당 업무
1	팀장	강기훈	한양대학교 (Erica)	로봇공학과	▪ 개발환경 구축 ▪ 구현 기술 문서화 ▪ 알고리즘 총괄 ▪ 영상처리 및 시스템 총괄
2	팀원	김지일	한양대학교 (Erica)	로봇공학과	▪ 센서 필터링 및 맵핑 ▪ 터널 코스 구간 구현 ▪ 신호등 분기점 구간 구현 ▪ 우선정지 장애물 구간 구현
3	팀원	송희원	한양대학교 (Erica)	로봇공학과	▪ 구현 기술 문서화 ▪ 수평 및 수직 주차 구간 구현 ▪ 회전 교차로 구간 구현 ▪ 신호등 분기점 구간 구현
4	팀원	이지화	한양대학교 (Erica)	로봇공학과	▪ 수평 및 수직 주차 구간 구현 ▪ 우선정지 장애물 구간 구현 ▪ 센서 필터링 및 맵핑
5	팀원	정은수	한양대학교 (Erica)	로봇공학과	▪ 개발환경 구축 ▪ 오르막 및 내리막 구간 구현 ▪ 차로 추월 구간 구현 ▪ 곡선(S)자 구간 구현

6.2. 개발 진행 절차

		6월	7월	8월	9월
개발환경 구축	개발환경 구축	① ⑤	① ⑤		
	경기장 제작		All	All	All
영상 처리	차선 검출 및 주행	① ⑤			
	내리막길 진입 인식			⑤	⑤
	우선정지 장애물 인식		② ④	② ④	
	곡선(S)자 정밀 주행		① ⑤	① ⑤	
	회전 교차로 진입 인식			③	
	터널 진입 인식		② ③		
	추월구간 진입 인식				③ ⑤
	신호등 인식			② ③	② ③
센서 및 제어	수평 및 수직 주차		③ ④	④	④
	선행 차량 인식				③
	터널 주행			② ③	② ④
	추월구간 주행				③ ⑤
실험 및 완성	알고리즘 최적화		① ③	① ⑤	① ⑤
	개발완료보고서 작성				① ③
	완성				All