2019 Spring ME492 Final Report

Team A
Jongwon Kim, 20140135
Gijeong Kim, 20160067
Inchul Chung, 20160597
Minjae Cho, 20160624

2019.06.16

1. Philosophy:
path generation을 통한 가장 효율적인(빠른) path의 설정과 불필요한 움직임의 최소화

1.1 개요 (코드를 이해하기 위해 알아야하는 하드웨어의 특징 및 코드의 전반적인 흐름):

아래 사진이 우리의 모델이다. 공을 먹기 위해서 별 다른 pick up motion이 필요 없으며, 직진 motion만으로 파란 공을 먹을 수 있다. 지나가는 모든 공을 먹기 때문에 빨간 공의 회피 모션이 반드시 필요했다. 하지만, 회피 모션의 경우 근처의 공을 치거나 다른 생각하지 못한 변수를 만들 위험이 있어 최대한 배제하고자 했고, 그렇게 나온 것이 path generation이다.

우리는 처음 시작할 때, 모든 6개 공의 위치를 가지고 최적의 경로를 탐색한다. 빨간 공이 3개이고 파란 공이 3개 일 때 나올 수 있는 총 경로의 수는 6개 이기에, 각각의 상황에 대한 예상 시간을 구하고 이 중 가장 시간이 짧은 경로를 선택한다.

우리가 path로부터 얻고 싶은 정보는 rough한 angle 1,2,3,4이다. Angle1은 처음 시작 지점에서 첫번째 공을 바라보기 위해 돌아야 하는 각도이고, Angle 2(3)는 첫번째(두번째) 공을 먹은 자리에서 두번째(세번째) 공을 바라보기 위해 돌아야 하는 각도이다. 마지막으로 Angle 4는 세번째 공을 먹은 자리에서 초록 색 공을 바라보기 위해 돌아야 하는 각도이다. 이 'rough' angle을 돌게 함으로써, 목적으로 하는 공이 캠의 가운데에 제일 '근접'하도록 만들 수 있는 것이다. 로봇이 Rough angle을 돈 이후에 가운데에 제일 근접한 공을 목표로 'tracking'해 나가는 것이 우리 코드의 전반적인 흐름이다.

하지만 처음에 공의 위치를 제대로 파악하지 못한다면, 위의 path generation에서 나온 rough angle은 의미가 없어진다. 기존의 RGB camera의 경우, 3m 이상만 되어도 공까지의 거리가 매우 부정확하게 나왔는데, 이는 공을 detect했을 때 나온 공의 contour와 실제 공의 크기를 비교하며 거리를 파악했기 때문이다. 따라서 정확한 공의 위치를 알아야 했기에, RGBD camera를 사용하게 되었다. 공의 detect는 RGB camera로 하고 공까지의 거리는 depth camera로 구했기에, 먼 거리의 공의 좌표도 훨씬 정확하게 잡을 수 있었다. 이는 path generation에서 나온 angle 1,2,3,4를 더 reliable하게 만들었기에, path generation의 효과를 매우 높이는데 기여했다.

1.2 Greedy algorithm vs Dynamic Programming:

Greedy Algorithm은 갈림길에서 선택의 순간마다 최선의 경로를 택한다. 때문에 도착점에 도달하기까지 가능한 모든 경우의 수를 계산할 필요 없이 선택된 경로의 하위 경로에 대해서만 지속적으로 계산을 해주면 된다. 이는 계산에서의 이점을 가져올 수 있으나, 이러한 방법으로 만들어진 최종 경로가 시작점에서 도착점까지 가능한 모든 경로 중 최고의 경로라고 장담할 수는 없다.

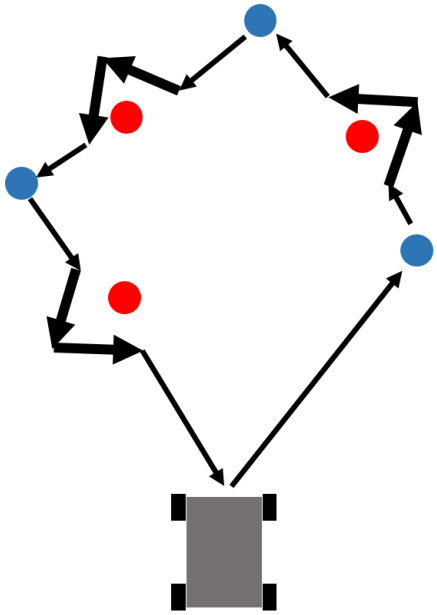
Dynamic Programming은 시작점에서 도착점까지 가능한 모든 경로를 계산한 뒤 그 중 최선의 경로를 택한다. 이는 시작할 때 모든 경우를 계산해야 하여 계산에 필요한 시간이 소요되

나 언제나 최선의 경로를 선택한다는 장점이 있다.

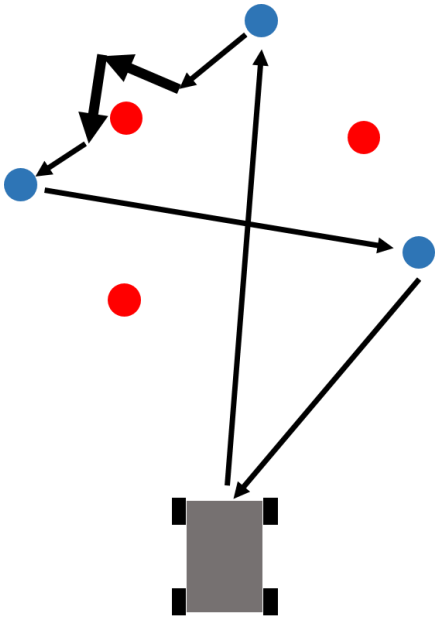
우리는 Dynamic Programming 알고리즘을 채택하였다. 3개의 공을 먹는 모든 경우의 수는 6가지 밖에 되지 않고 CPU의 연산속도는 차체의 이동 속도에 비하면 매우 빠르다. 즉, Greedy Algorithm에서 다음 공을 찾기 위해 소요하는 불필요한 시간이 6가지 경로를 계산하는데 필요한 시간 보다 매우 크다는 것이다. 따라서 Path generate를 통해 최적의 경로를 탐색하는 알고리즘을 사용함으로써 시간을 대폭 단축시키고자 했다.

예를 들어, 아래 그림과 같이 공이 놓인 상황에 대해 생각해 보겠다. 만약 왼쪽으로 회전하며 찾은 공을 먹어가는 greedy algorithm을 택한다면 (1)의 결과와 같을 것이다. 3번의 회피 모션이 들어가며, 각각의 회피 모션에서 대각선 이동과 회전을 하기에 많은 시간이 소요될 것이다. 이는 가장 빠른 path를 찾고자 하는 우리의 목표에 어긋났다.

우리는 회피 모션에 가장 큰 가중치를 두고 이를 최대한 피하고자 했다. 이렇게 만들어진 path generation의 실제 실험 결과는 (2)와 같다. 회피 모션을 한번만 하고 속도가 빠른 직진 모션의 비중을 높임으로써, 실제로 위의 경로에 대해서 30초대 초반 정도로 시간을 굉장히 줄일 수 있었다. 이처럼 우리는 어떤 상황이 나오든, 각각의 상황에서 가장 빠른 path를 path generation을 통해 찾고자 했다.



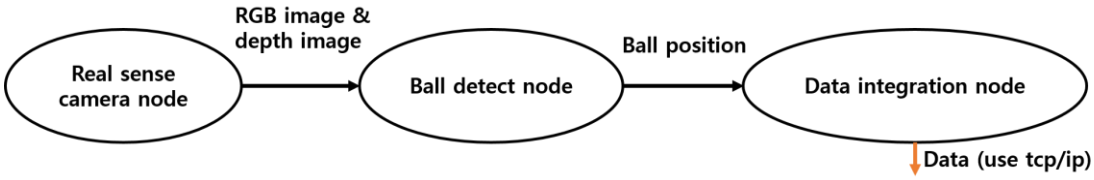
(1) Greedy algorithm



(2) Path generation

2. Ideas in our code:

2.1 About whole ROS programming (message 를 중심으로 설명)



Ball detect node 에선 Realsense 카메라 노드로부터 받은 rgb image 와 depth image 를 이용해, red ball 과 blue ball, green ball 각각의 숫자(r_size, b_size, g_size)와 각각의 차체 기준 x,y 좌표(r_img_x, r_img_y, b_img_x, b_img_y, g_img_x, g_img_y)를 ball position message 에 담는다.

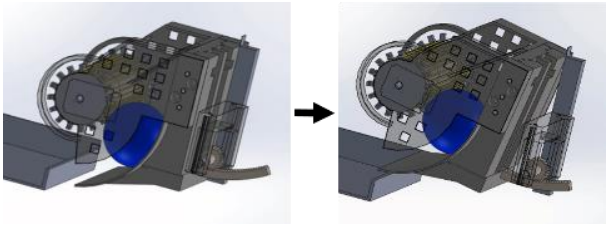
Ball position.msg

```
r_size
b_size
g_size
b_over

r_img_x
r_img_y

b_img_x
b_img_y

g_img_x
g_img_y
```

Depth cam 로부터 나온 거리 데이터(depth cam 기준 z 좌표)와 rgb image 에서 구한 ball 의 위치(rgb cam 기준 x,y 좌표)를 기반으로, 각각의 ball 의 '차체 기준' 상대 좌표 x,y 를 계산하게 된다. 양의 x 는 차체 기준으로 x 만큼 오른쪽에 ball 이 위치해 있음을 뜻하고, 양의 y 는 차체 기준으로 y 만큼 앞 쪽에 ball 이 위치함을 뜻한다.

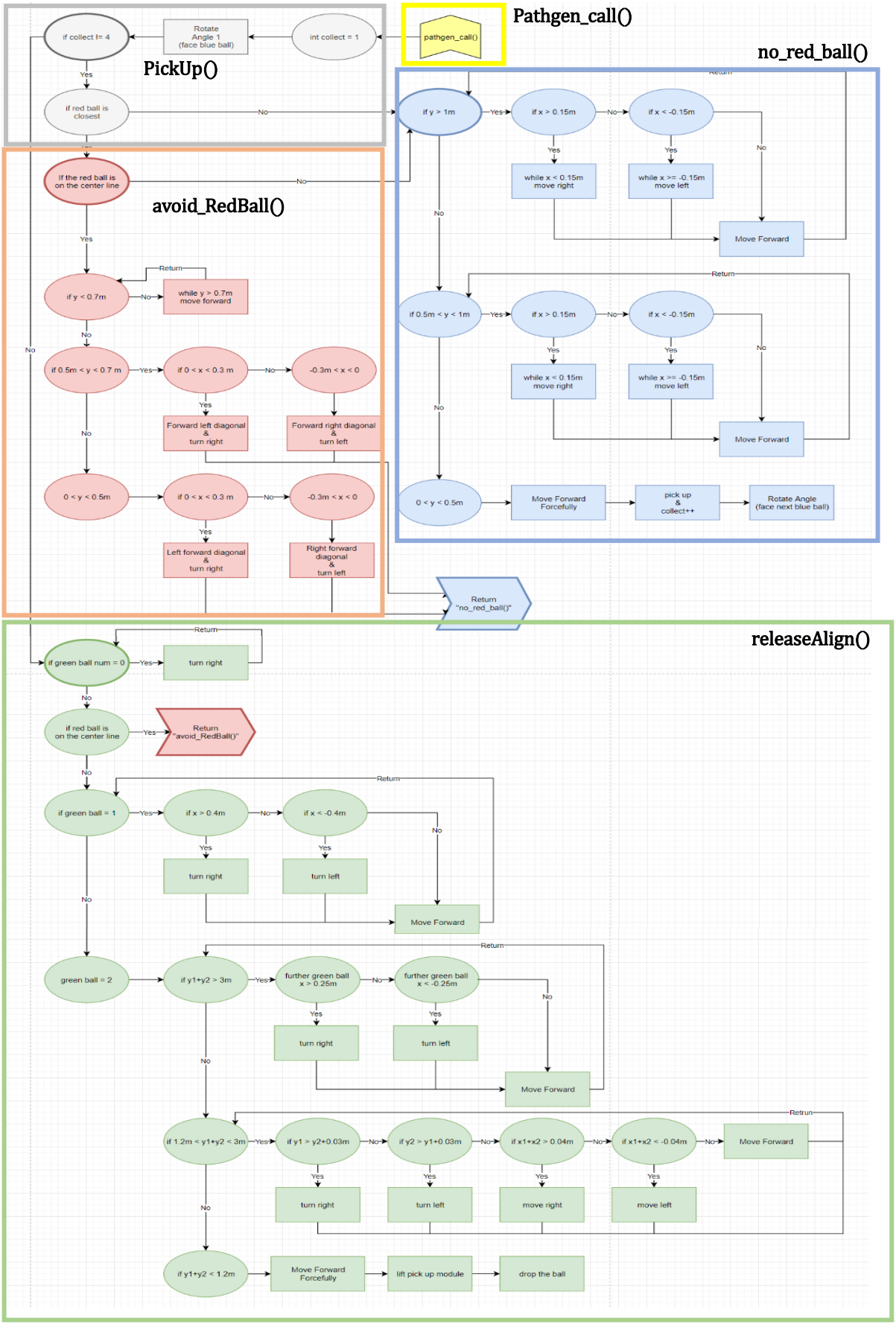
처음 path generation 할 때 우리는 반드시 blue ball 3 개는 detect 해야 한다. 대부분의 경우, 시작할 때 3 개의 blue ball 을 모두 detect 할 수 있었지만, blue ball 두개가 차를 기준으로 일직선상에 있는 경우 하나의 blue ball 로 detect 하는 경우가 간혹 있었다. 이 경우, blue ball num = 2 가 되어 path gen 을 할 때 문제가 생길 수 있다. 따라서 이런 경우 0 번 공 뒤에 또 다른 blue ball 이 있는 경우는 b_over=0, 1 번 공 뒤에 또 다른 blue ball 이 있는 경우는 b_over=1 로 b_over 의 변수를 사용하여 공이 겹쳤음을 알려주게 했다. (depth cam 기준 공의 z 좌표와 rgb cam 기준 공의 z 좌표를 비교하여 공이 겹쳤는지 알 수 있다. 공이 겹친 경우 rgb 에서 실제보다 크게 보이기 때문이다.) 3 개의 공이 정상적으로 detect 되는 보통의 경우엔 b_over=-1 이 되며 정상적으로 path gen 을 시작한다.

따라서 data integration node 에서는 ball detect node 에서 보내는 ball position message 를 이용해 현재 상황을 판단하고 그 상황에 맞는 차체의 이동 명령을 'data' 변수에 담아 myRIO 로 보내준다. Data 는 크기 24 의 array 로, data[0]이 차의 V_x, data[1]이 차의 V_y, data[4]가 차의 CW rotation ω, data[14]가 pick up roller 의 회전 방향, data[17]이 release 를 위해 pick up module 을 들지 말지를 결정한다. 전진이나 좌우 이동, 회전, 대각선 이동의 경우 data[0], data[1], data[2]의 값을 조정해 줌으로써 모두 구현할 수 있다. 그리고 release 는 오른쪽 사진과 같이 pick up module 을 들어올리고, roller 를 반대로 돌려 실행한다. 따라서 data[17]을 이용해 pick up module 을 들어올리고 data[14]를 이용해 roller 를 반대로 돌리면 된다.



2.2 Flow in data integration node

우리는 data integration node에서 다음과 같이 다섯 개의 함수를 중심으로 알고리즘을 구성하였다. 처음에 최적의 path를 찾는 pathgen_call() 함수, 이후 path gen이 완료되면 현재 상황 따라 적절한 함수를 호출하는 PickUp() 함수, 경로 상 빨간 공이 존재할 때 회피하도록 하는 avoid_RedBall() 함수, 빨간 공이 없을 때 파란 공을 수집하는 no_red_ball() 함수, 파란 공을 모두 수집한 뒤 바구니를 향해 돌아가는 releaseAlign() 함수가 있다. 그 흐름은 아래의 flow chart와 같다. 표에서 등장하는 x,y는 각 상황에서 판단의 기준이 되는 ball의 차체 기준 x,y 좌표이다.


Pathgen_call()
pathgen_call()
PickUp()
int collect = 1
Rotate Angle 1 (face blue ball)
if collect != 4
Yes
if red ball is closest
No
avoid_RedBall()
If the red ball is on the center line
No
Yes
Return
if y < 0.7m
No
while y > 0.7m move forward
if 0.5m < y < 0.7 m
Yes
if 0 < x < 0.3 m
No
-0.3m < x < 0
Yes
Forward left diagonal & turn right
Forward right diagonal & turn left
No
0 < y < 0.5m
if 0 < x < 0.3 m
No
-0.3m < x < 0
Yes
Left forward diagonal & turn right
Right forward diagonal & turn left
no_red_ball()
if y > 1m
Yes
if x > 0.15m
No
if x < -0.15m
Yes
while x < 0.15m move right
Yes
while x >= -0.15m move left
No
Move Forward
No
if 0.5m < y < 1m
Yes
if x > 0.15m
No
if x < -0.15m
Yes
while x < 0.15m move right
Yes
while x >= -0.15m move left
No
Move Forward
No
0 < y < 0.5m
Move Forward Forcefully
pick up & collect++
Rotate Angle (face next blue ball)
Return "no_red_ball()"
releaseAlign()
if green ball num = 0
Yes
turn right
No
if red ball is on the center line
Yes
Return "avoid_RedBall()"
No
if green ball = 1
Yes
if x > 0.4m
No
if x < -0.4m
Yes
turn right
Yes
turn left
No
Move Forward
No
green ball = 2
if y1+y2 > 3m
Yes
further green ball x > 0.25m
No
further green ball x < -0.25m
Yes
turn right
Yes
turn left
No
Move Forward
No
if 1.2m < y1+y2 < 3m
Yes
if y1 > y2+0.03m
No
if y2 > y1+0.03m
No
if x1+x2 > 0.04m
No
if x1+x2 < -0.04m
No
Move Forward
Yes
turn right
Yes
turn left
Yes
move right
Yes
move left
No
if y1+y2 < 1.2m
Move Forward Forcefully
lift pick up module
drop the ball

3. Explanation of code

3.1 Settings [글로벌 변수들 및 기본 함수들]

3.1.1 글로벌 변수 [1~124]

1~11 : 같은 이름의 파일이 많아 혼동을 방지하기 위해 수정날짜와 내용, 담당자를 표시함

12~37 : 코드를 실행시키기 위한 헤더 파일들 ROS 구동을 위한 파일들 + 수학연산 및 알고리즘 구동을 위한 파일들 + 기본 cpp 구동을 위한 파일들 + opencv를 위한 파일들 등등이 포함되어 있다.

40~51 : void avoid_RedBall(); void no_red_ball(); void releaseAlign(); 함수 실행을 위한 글로벌 변수 선언

54~89 : 코드 전체를 위한 글로벌 변수 선언

54 : 각도 변환(rad -> deg) 함수 define

56~57 : 소켓 통신을 위한 주소와 포트 넘버

61~65 : lidar 데이터를 위한 변수 (우리 조는 사용하지 않았다)

67~77 : camera 데이터 수신을 위한 변수 (색깔 별 공 개수 및 좌표 등)

79~83 : 코드 실행 중 실행단계 및 path planning 모드를 표시하기 위한 변수

85~89 : 소켓 설정 및 데이터 전송을 위한 변수들

92~120 : pathgen()을 위한 변수

99 : 6종류의 path의 time cost를 저장

102~106 : 속도 및 움직임 변환 가중치. 이 변수들을 통해 회피 동작을 최소화하는 혹은 회전을 최소화하는 등의 경로 설계가 가능해진다.

110~112 : pathgen()에서 사용할 파랑/빨간색 공들의 좌표;

114~118 : pathgen() 알고리즘의 최종 출력값이다. 이를 바탕으로 실제 움직임에서 한 지점에서 다음 목표지점을 바라봐야 할 때 어느 정도 회전해야 하는지 알려준다.

angle 1 : 시작 위치에서 첫 번째 공을 바라볼 때 회전해야 하는 각도

angle 2 : 첫 번째 공을 먹은 후 두 번째 공을 바라보기 위해 회전해야 하는 각도

angle 3 : 두 번째 공을 먹은 후 세 번째 공을 바라보기 위해 회전해야 하는 각도

angle 4 : 세 번째 공을 먹은 후 출발지점을 바라보기 위해 회전해야 하는 각도

angle 5 : 출발지점으로 돌아온 후 바구니를 바라보기 위해 회전해야 하는 각도

123~124 : 오류를 쉽게 확인하고 수정하기 위해 코드들이 실행되는 과정들을 result.txt 파일에 저장하여 확인하였다.

3.1.2 void dataInit() [126~152]

소켓 통신으로 보내는 데이터를 초기화하는 함수

3.1.3 void move_forward() 등의 움직임 함수 [154~428]

특정 움직임을 위해 data[]를 수정하는 함수들. 전진, 후진, 좌우 이동, 대각선 좌우 이동, 좌우 회전, 정지 등이 속도별로 정의되어 있으며 픽업 모듈을 들어 올리거나 내리고 회전 방향을 바꾸는 함수 또한 포함되어 있다.

3.1.4 void camera_Callback() [430~507]

ball_detect 노드에서 들어오는 메시지를 처리해 전역변수에 저장하는 콜백 함수이다.
433~441: 메시지에 있는 색깔 별 공의 개수를 받아와 변수에 저장해준다. 새로 받은 공들 중 공이 아님이 판별되었거나 너무 가깝거나 먼 공을 제거해 필요한 데이터만 사용할 수 있도록 하기 위한 변수 또한 선언한다.
443~455: 빨강 공 데이터 선별 및 저장. 공의 크기로 측정한 거리와 depth 정보의 차이가 커 공이 아님으로 판명된 경우 공의 y좌표가 -100으로 설정되어 넘어오게 된다. 이러한 경우를 제거해 x,y 좌표를 저장하고 실제 공의 개수를 red_ball_number에 저장한다.
457~469: 위에서 빨간 공에게 해준 과정과 비슷하게 파란 공도 처리해준다. 다만 벽이나 바닥에 비치며 잡힌 공이나 픽업 모듈 안에 있는 공이 잡히는 등의 추가적으로 발생하는 오류 또한 제거해주기 위해 거리가 32 이하 이거나 (-100포함) 500 이상일 때 또한 거르게 조건을 설정해 올바른 x y 값과 공의 개수를 저장한다.
472~484: 위에서 빨간 공에게 해준 과정과 같게 초록 공도 처리한다.
486~492: 가장 가까운 빨간 공의 인덱스 값을 찾는 부분이다. 이를 통해 찾은 빨간 공의 위치를 바탕으로 회피를 여부를 결정하게 된다.
493~499: 차체가 바라보는 방향(카메라의 중간)에서 x 좌표(좌우)가 가장 가까운 파란 공을 찾는다. 이를 통해 찾은 파란 공을 계속 트래킹하며 나아간다.
500~506: 가장 가까운 파란 공을 찾는 코드이다.

3.1.5 void cor_show() [509~528]

카메라 콜백 함수를 통해 처리되어 들어온 공의 좌푯값을 출력해주는 함수이다. 이를 통해 콜백 함수를 통해 데이터가 어떻게 처리되었고 결국 우리가 사용하게 될 데이터가 어떤 것인지를 쉽게 확인할 수 있다.

3.1.6 void init_move() [530~542]

처음 공의 위치를 파악하고 path planning을 하기 전에 그 정확도 향상을 위해 차체를 앞으로 조금 이동시키기 위해 사용하려 했던 함수이다. 그러나 카메라의 시야각 등의 이유로 실제로 사용하지는 않았다.

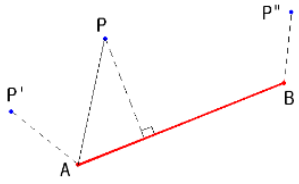
3.1.7 double pdis() [771~774]

두 점 사이의 거리를 반환해주는 함수이다.

3.1.8 double inner() [776~779]

두 벡터 사이의 내적값을 반환해주는 함수이다.

3.1.9 double mindis() [781 ~ 792]



선분과 점 사이의 거리를 반환해주는 함수이다. 직선과 점 사이의 거리는 식 하나로 구할 수 있지만 선분과 점 사이의 거리는 점의 위치에 따라 3가지로 구분하여 거리를 구하여야 한다.

3.1.10 bool b_ball_in_path(int b1, int b2, int b3) [794~806]

경로 중에서 한 파란 공에서 다음 파란 공으로 가는 사이에 다른 파란 공이 있으면 이 경로는 비효율적이고 이보다 다른 순서로 공을 수거하는 것이 더 효율적임이 확실하므로 이를 거르기 위한 함수이다. 파란 공을 수거하는 순서(b1, b2, b3: 공이 저장된 배열의 인덱스이다.)를 입력 받으면 이를 이동하는 경로 사이에 파란 공이 있는 경우가 있는지 판단하여 비효율적인 경우 true를 반환하고 그렇지 않을 경우 false를 반환한다.

3.1.11 double angle() [808 ~ 825]

출발 위치, 회전 위치 목표 위치를 입력하면 회전 위치에서 회전해야 하는 각도를 도 단위로 반환하는 함수이다. 각도를 계산할 때 acos나 atan은 0~180도까지만 계산할 수 있기에 360도 가능한 atan2를 사용하였다. 이 함수의 출력값이 음수면 왼쪽으로 양수이면 오른쪽으로 회전해야 한다는 의미를 지닌다.

3.1.12 void rotate(int angle) [1042~1096]

로봇을 angle 만큼의 각도로 openloop 회전시키는 함수이다.

1043~1062: rotated값이 0일 경우(angle1을 회전하는 경우) 이전 빨간공 회피에 의한 보정 값이 없기에 angle의 부호에 따라 오른쪽 혹은 왼쪽으로 천천히 회전한다. openloop 회전이기 때문에 for문의 반복횟수를 실험적으로 설정하였다.

1063~1080: rotated 값이 0이 아닐 경우 이전의 회피 여부에 따라 lTheta, rTheta의 보정 값이 존재하게 된다. angle 값에 이를 보정해준 후 openloop으로 로봇을 회전시킨다.

1081~1086: 보정 값 변수들을 초기화 시키고 회전이 끝난 로봇을 멈춘다.

1095: openloop 회전이 한번 진행되었으므로 rotated를 ++ 해준다.

3.1.13 int main(int argc, char **argv) [1672~1731]

1675~1679: ros 노드 실행 및 subscriber 설정을 위한 코드

1684~1691: 코드를 실행한 후 ball_detect 노드에서 데이터가 넘어올 때까지 잠깐의 딜레이가 존재한다. 이때 이후의 코드가 실행되면 문제가 생기기에 데이터가 넘어올 때까지 기다린다.

1692: cor_init()을 통해 path planning에 필요한 좌푯값을 계산한다.

1695~1700: pathgen_call()을 통해 최적의 경로 및 회전각을 계산한다. 함수가 제대로 실행되었는지 확인한다.

1702~1711: myrio와의 통신을 위한 소켓을 연결하고 연결이 잘되었는지 확인한다.

1713~1728: collect가 5가 될 때까지(공 전부 수거 후 반환까지 완료한 상태) Pickup() 함수를 반복해서 실행한다.

1729: 실행정보를 입력하던 파일을 닫는다.

1730: 전체 코드를 종료한다.

3.2 Settings [글로벌 변수들 및 기본 함수들]

3.2.1 중요 변수

3.2.1.1 가중치

직진 속도, 회전 속도 대각선 이동속도(회피속도), 움직임 변환의 가중치를 두고 path planning의 time cost 계산에 사용하였다. 속도 가중치 값이 클수록 특정 동작으로 일정거리를 이동하는데 적은 시간이 들어 특정 동작을 비교적 선호하는 경로가 계산되게 된다. 반대로 속도 가중치 값이 작을 경우 특정동작을 실행하는 데에 많은 시간이 들기에 이를 최소화하는 경로가 선정되게 된다. 이를 이용하여 회피에 필요한 대각선 가중치 값을 낮게 두었고 회피를 최소화하는 경로를 얻어 시간 단축 및 정확성을 높이는 결과를 얻을 수 있었다.

3.2.1.2 각도(angle1~5)

Path planning의 최종 결과가 담기는 변수이다. 각 각도는 한 지점에 도착한 이후 다음 지점을 바라보기 위해 회전해야 하는 각도를 일컫는다. 이 각도를 바탕으로 우선 빠른 회전을 진행한 후 중앙에서 가장 가까운 목표를 향해 나아가게 된다. (angle1: 출발지점 -> 1번째 공, angle2: 1번째 공 -> 2번째 공, angle3: 2번째 공 -> 3번째 공, angle4: 3번째 공 -> 출발지점, angle5: 출발지점 -> 바구니)

3.2.2 void cor_init() [544~769]

한 개의 프레임에서 인식되는 공의 정보는 오차를 포함하고 있을 확률이 높기에 여러 프레임의 정보를 처리하여 좀더 정확도가 높은 값을 반환해 path planning에 사용될 수 있도록 한다.

- 몇 개의 공이 인식되는지 파악하기 위한 변수들이다. [554~563]
- 좌표값의 평균을 얻기 위해 임시로 값을 저장해 놓은 배열이다. [566~569]
- 조건의 만족하여 좌표값이 계산되기 전까지 딜레이를 갖고 반복적으로 데이터를 받아들이고 처리한다. [571~574]

3.2.2.1 파란색 공

파란색 공의 경우 3가지의 상황으로 분류된다. 첫번째의 경우는 3개의 공이 인식되고 가려져 있는 공이 없다고 판단된 경우, 두번째 경우는 2개의 공이 인식되고 가려져 있는 공이 있다고 판단된 경우, 세번째의 경우는 위 두 경우에 포함되지 않는 나머지의 경우이다. 데이터를 지속적으로 갱신하여 현재 상태가 어떠한 경우인지 파악하여 pathmode변수로 저장하고 3프레임의 값을 평균 내어 공의 좌표 또한 path planning에 사용될 수 있도록 한다. 만약 세번째의 경우에 해당되면 path planning의 계산이 이루어질 수 없기에 path planning의 과정을 뛰어넘고 greedy한 알고리즘으로 로봇을 움직일 수 있게 한다.

3.2.2.1.1 파란색 공이 3개가 인식되고 가려져 있는 공이 없다고 판단되었을 경우 좌표값을 누적해서 더해주는 부분이다. [577~590]

3.2.2.1.2 파란색 공 2개가 인식되고 한 개의 공이 다른 공 뒤에 가려있다고 판단되었을 경우 좌표값을 누적해서 더해주는 부분이다. 이전에 공 3개가 잡혔다가 이 경우로 넘어온 것이라면 임시저장 배열을 초기화한 후 현재 데이터를 입력한다. [591~621]

3.2.2.1.3 위 두 상황에 해당하지 않는 상황이 지속할 경우 path planning이 불가능하므로 while 함수를 탈출한다. [622~640]

3.2.2.2 빨간공

인식되는 개수에 오류가 생겨도 path planning에서 문제는 생기지 않는다. 따라서 몇 개의 공이 인식되는지 파악한후 그 개수에 맞추어 3프레임 동안의 값을 평균 낸다. 만약 빨간 공의 개수가 3개보다 적은 경우 나머지 공의 좌표는 (0,-100)으로 지정해 path planning에 영향을 주지않게 한다.

3.2.2.2.1 빨간 공이 3개 인식될 경우 데이터를 누적하는 함수이다. [644~655]

3.2.2.2.2 빨간 공이 2개 인식될 경우 데이터를 누적하는 함수이다. 남은 한 개의 공은 (0,-100)으로 설정한다. [656~670]

3.2.2.2.3 빨간 공이 1개 인식될 경우 데이터를 누적하는 함수이다. 남은 한 개의 공은 (0,-100)으로 설정한다. [671~687]

3.2.2.2.4 빨간 공이 인식되지 않을 때 모든 좌표를 (0, -100)으로 설정한다. [688~700]

3.2.2.2.5 3개 이상의 빨간 공이 인식될 경우 이는 오류임이 분명하므로 다음 프레임으로 넘어간다. [701~705]

3.2.2.2.6 같은 조건을 지닌 프레임의 데이터가 충분히 누적되었다고 판단되면 while 문을 탈출한다. [708]

3.2.2.2.7 누적된 빨간 공의 좌푯값을 프레임 수로 나눈다. [715~719]

3.2.2.2.8 파란 공이 3개일 때 좌푯값을 프레임 수로 나누고 pathmode를 2로 설정한다. [722~730]

3.2.2.2.9 파란 공이 2개일 때 좌푯값을 프레임 수로 나누고 pathmode를 0혹은 1로 설정한다. 나머지 한 공의 좌표는 (0, 0)으로 설정한다. [731~742]

3.2.2.2.10 파란 공을 탐지하지 못하는 경우 모든 공 좌표를 (–100,-100)으로 설정해준다. [743~751]

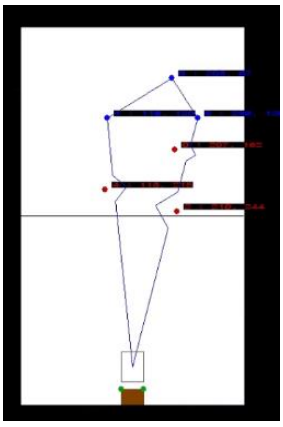
3.2.2.2.11 최종 계산 결과를 출력한다. [758~765]

3.2.3 pathgen() [827~900]

파란 공 3개가 cor_init()에서 인식된 경우 실행되는 path planning 함수이다. Input 값으로 공을 수거하는 순서 (좌표가 저장된 배열의 인덱스)가 주어지며 이 경로에 대한 time cost를 반환한다.

* Calculate time cost

출발지점, 첫번째 공, 두번째 공, 세번째 공, 출발지점을 5개의 지점으로 잡고 한점에서 다른 점으로 가는 단계를 한 단계로 가정한다. 한 단계 안에서는 현재 위치에서 다음 지점을 바라보기위한 회전, 다음 지점까지의 직진거리, 경로에 빨간 공의 유무에 및 거리에 따른 회피 움직임의 거리, 움직임 전환시간 등을 계산하고 각 값들을 각 가중치로 나누어 준다. 가중치로 나누어진 값들을 전체 경로에 대하여 모두 더하여 주면 그 경로에 대한 time cost 값을 얻을 수 있다.



3.2.3.1 b_ball_in_path(b1, b2, b3)를 이용하여 time cost를 계산하기 전에 경로에 파란 공이 있는지 확인한 후 비효율이라고 판단하면 시간을 max로 설정한 후 pathgen()을 탈출한다. [832~834]

3.2.3.2 시작지점과 첫 번째 공의 사이의 거리를 직진속도 가중치 값으로 나눈 값을 time cost에 더한다. [836]

3.2.3.3 시작지점에서 첫 번째 공을 바라보기 위해 회전해야 하는 각도를 회전속도 가중치 값으로 나는 값을 time cost에 더한다. 움직임의 순서로는 836번째 코드보다 앞에 있어야 한다. [837]

3.2.3.4 빨간색 공 3개가 시작지점에서 첫 번째 파란 공을 가는 사이 경로 근처에 있는지 판단한다. mindis()를 통해 이동 경로(선분)와 빨간 공 사이의 거리를 구하고 이 거리가 안전거리보다 가까우면 회피 이동을 time cost에 추가한다. 회피 이동의 경우 계획단계에서는 두 번의 대각선이동으로 피한다고 가정했기에 빨간 공과의 거리가 안전거리가 될 때까지 대각선 이동하고 다시 대각선 이동하여 파란 공을 트렉킹 할 수 있는 위치로 돌아올 수 있도록 하는 거리를 계산하고 이를 대각선 이동 가중치 값으로 나눠준 후 더한다. 회피 이동을 하는 경우 그 위치에서의 직선 이동은 없어지기에 그에 대한 cost는 빼준다. 또한, 움직임 간의 전환 딜레이를 가중치로 고려해 넣어주었다. [838~849]

3.2.3.5 (838~849)의 부분에서 했던 과정을 첫 번째 공을 수거한 후 두 번째 공을 수거하러 가는 경우로 좌표를 바꾸어 반복한다. [851~864]

3.2.3.6 (838~849)의 부분에서 했던 과정을 두 번째 공을 수거한 후 세 번째 공을 수거하러 가는 경우로 좌표를 바꾸어 반복한다. [866~879]

3.2.3.7 838~849의 부분에서 했던 과정을 세 번째 공을 수거한 후 출발했던 위치로 돌아가는 좌표를 바꾸어 반복한다. [881~894]

3.2.3.8 원위치로 도착한 후 release해야 하는 바구니를 바라보기 위한 회전값을 가중치에 더해준다. pathgen 에서는 이후 공을 release하는 모션 및 경로는 고려하지 않았다(어떠한 경로이든 release에 걸리는 시간을 거의 동일하기 때문). [896]

3.2.4 tcal()[930~959]

3개의 공을 수거할 수 있는 6가지의 모든 경우의 수에 대하여 pathgen()함수를 실행하고 그중 최솟값을 찾아 그 경로를 반환해주는 함수이다.

3.2.4.1 각 경로에 따라 pathgen 함수를 통해 time cost를 구한 후 tcost[] 배열에 담는다. [932~937]

3.2.4.2 tcost[]의 값을 출력한다. [939~946]

3.2.4.3 tcost[]의 값을 비교하여 최솟값을 찾는다 그 최솟값을 지닌 경로를 반환해준다(0~5의 숫자). [948~959]

3.2.5 pathgen2() [902~928]

파란 공 2개가 인식되고 나머지 한 공이 다른 공에 가려 있다고 판단되는 경우 실행되는 path planning 함수이다. 어느 공 뒤에 다른 공이 가려져 있는지 ball_detect 노드에서 파악하여 전달해주기에 이를 바탕으로 대략적인 경로를 설계할 수 있다. 한 공의 좌표를 모르는 상황에서 최적의 경로는 우선 가리고 있는 공을 수거한 후 가려진 공을 수거하고 나머지 공을 마지막으로 수거하는 결로라고 판단하였다.

3.2.5.1 가려져 있는 공이 0번째 공 뒤에 있는지 1번째 공 뒤에 있는지는 cor_init 함수를 통해 pathmode에 저장된다. 이 변수를 바탕으로 공을 수거하는 순서를 정해주게 된다. 기본적으로 다른 공을 가리고 있는 공을 수거하고 그 뒤에 가려져 있던 공을 두 번째로 먹은 후 나머지 공을 세 번째로 먹는 방식을 취한다. [906~913]

3.2.5.2 위에서 정한 공의 수거 순서를 바탕으로 회전각을 정해준다. angle1과 angle5는 다른 경우와 동일하게 구해준다. angle2의 경우 처음 공을 먹고 뒤에 가려져 있던 공은 거의 로봇이 바라보는 방향에 있을 것이기 때문에 0도로 지정하였다. angle3의 경우 두 번째 공의 정확한 위치를 모르기 때문에 정확한 각도를 정할 수 없다. 그러나 첫 번째 공과의 상대적 위치를 통해 회전 방향은 알 수 있기에 이에 따라 1000 혹은 –1000으로 지정해준다. 이후 이 값의 부호를 통해 정해진 방향으로 회전하며 공을 찾게 된다. angle4의 경우 앞과 똑같이 정확한 값은 구하지 못하지만, 방향은 알 수 있기에 1000 혹은 –1000의 값을 지정해준다. [915~920]

3.2.5.3 계산된 각도 값을 확인을 위해 출력해준다. [922~927]

3.2.6 pathgen_call() [961~1039]

실질적으로 path planning을 진행하는 메인 함수이다. cor_init()에서 파악한 모드를 바탕으로 tcal() 혹은 pathgen2()를 호출한다. Pathgen2()의 경우 이 함수 자체에서 각도를 계산하는 부분이 있기에 Pathgen_call() 함수는 종료가 되고 tcal()을 호출한 경우 반환된 경로를 바탕으로 각도들을 계산하고 보정하여 전역변수에 저장해준다.

3.2.6.1 pathmode가 2이면(정상적인 상황) 뒤의 코드를 계속하고 0혹은 1이면 () pathgen2()를 호출하여 angle 변수들을 계산한다. [965]

3.2.6.2 tcal()을 호출해 6개의 경로 중 가장 time cost가 적은 경로를 얻는다. [966]

3.2.6.3 tcal()으로 얻은 경로 정보를 바탕으로 파란 공을 수거하는 순서를 b1, b2, b3에 저장한다. [970~994]

3.2.6.4 파란 공을 수거하는 순서와 좌표들을 바탕으로 angle1~5의 값을 계산한다. [996~1000]

3.2.6.5 작은 각도를 강제 회전하는 경우 상황에 따라 오차가 큰 경우가 발생하여 5도 이하의 작은 각도는 0으로 축소한다. [1002~1005]

3.2.6.6 3개의 파란 공이 출발지점 기준으로 어느 각도만큼 떨어져 있는지 계

산한다. 이 값을 통해 처음 먹는 공이 가장 오른쪽에 있는지 중앙에 있는지 가장 오른쪽에 있는지 알 수 있다. [1008~1010]

3.2.6.7 처음 수거하는 공이 가장 오른쪽에 있는 경우 angle1 값을 5 증가시켜 중앙에 있는 공을 잘못 인식할 확률을 낮추어 준다. [1012~1015]

3.2.6.8 처음 수거하는 공이 가장 오른쪽에 있는 경우 angle1 값을 5 감소시켜 중앙에 있는 공을 잘못 인식할 확률을 낮추어 준다. [1016~1019]

3.2.6.9 (1002~1005)의 과정을 한번 더 반복한다. [1020~1023]

3.3 void PickUp() [1534 – 1670]

3.3.1 초기값이 1인 collect 라는 정수형 변수를 만들어 공을 하나씩 수집할 때마다 increment한다. 또한 초기값이 0인 rotated 라는 정수형 변수를 만들어 공 수집 후 다음 공을 바라보도록 회전할 때마다 increment한다.

3.3.2 If collect = 4 -> 공 3개를 수집한 상태이므로 release motion을 수행한다. [1535-1537]

3.3.3 collect=4가 아닐 때는 collect의 값에 해당하는 stage에 따라 다음과 같이 행동한다.

3.3.3.1 Stage 1: 시작하자 마자 첫 번째 공을 향해 바라보도록 회전한. 이때 rotated=0, collect=1이다. Angle 1 회전 후 rotated=1이 되어 rotated=collect 조건을 만족하므로 no_red_ball() 함수가 실행되고 첫 번째 파란 공을 향하여 출발한다. [1560-1599]

3.3.3.2 Stage 2: 첫 번째 공을 먹은 뒤 stage 2로 넘어간다. 이때 rotated=1, collect=2이다. Angle 2 회전 후 rotated가 increment 되며 두 번째 파란 공을 바라본다. 마찬가지로 rotated=collect 조건을 만족하여 no_red_ball() 함수가 실행되고 두 번째 공을 향하여 출발한다. [1600-1633]

3.3.3.3 Stage 3: 두 번째 공을 먹은 뒤 stage 3로 넘어간다. 이때 rotated=2, collect=3이다. Angle 3 회전 후 rotated가 increment 되며 세 번째 파란 공을 바라본다. 마찬가지로 rotated=collect 조건을 만족하여 no_red_ball() 함수가 실행되고 세 번째 공을 향하여 출발한다. [1634-1667]

3.3.3.4 Stage 4: 파란 공 세 개를 모두 먹은 뒤 stage 4로 넘어간다. 세 번째 파란 공을 수집한 직후 collect 변수가 increment되어 변수 값이 4가 되면 초록 공을 향해 돌아가는 releaseAlign() 함수가 실행된다. [1535-1537]

3.3.4 collect = rotated 인 조건이 만족된 상태에서는 다음과 같이 움직인다. [1538-1557]

3.3.4.1 빨간 공이 가장 가까운 경우

3.3.4.1.1 가장 가까운 빨간 공의 x좌표가 -0.21m < x < 0.21m 인 경우 회피해야 하므로 avoid_RedBall() 함수를 호출한다.

3.3.4.1.2 그 외의 경우 빨간 공이 가장 가깝더라도 회피해야 할 필요가 없기 때문에 no_red_ball() 함수를 호출하여 파란 공을 향해 다가간다.

3.3.4.2 빨간 공이 가장 가깝지 않은 경우

3.3.4.2.1 파란 공이 가장 가깝기 때문에 빨간 공을 회피할 필요 없어 위와 같이 no_red_ball() 함수를 호출한다.

3.4 void no_red_ball() [1281 - 1371]

3.4.1 시야에 들어온 파란 공이 없을 경우 “no blue ball”을 출력하며 공을 찾을 때까지 왼쪽으로 회전한다. 이는 Path generation이 잘못되거나, path generation 경로대로 따라가지 않은 경우 발생한다. [1282-1290]

3.4.2 파란공이 시야에 포착된다면 가장 중심에 가까운 공 목표로 tracking하여 수집한다.

3.4.2.1 Distance > 1m : [1290-1320]

3.4.2.1.1 -0.15m < X_position < 0.15m : move forward [1293-1300]

3.4.2.1.2 X_position < -0.15m : move left [1303-1310]

3.4.2.1.3 X_position > 0.15m : move right [1313-1320]

3.4.2.2 Distance < 1m : 파란 공의 위치를 더 정확하게 맞춘다. [1324-1348]

3.4.2.2.1 -0.07m < X_position < 0.07m : move forward [1342-1348]

3.4.2.2.2 X_position < -0.07m : move left at medium speed [1324-1331]

3.4.2.2.3 X_position > 0.07m : move right at medium speed [1334-1341]

3.4.2.3 Distance < 0.5m : 카메라가 공을 포착할 수 있는 최소 거리 (0.1m)를 고려하여 일정 거리 이내로 가까워지면 강제 전진을 통해 파란 공 수집. 즉, 이 ‘강제 이동’을 해야 파란 공을 수집했다고 인식한다.

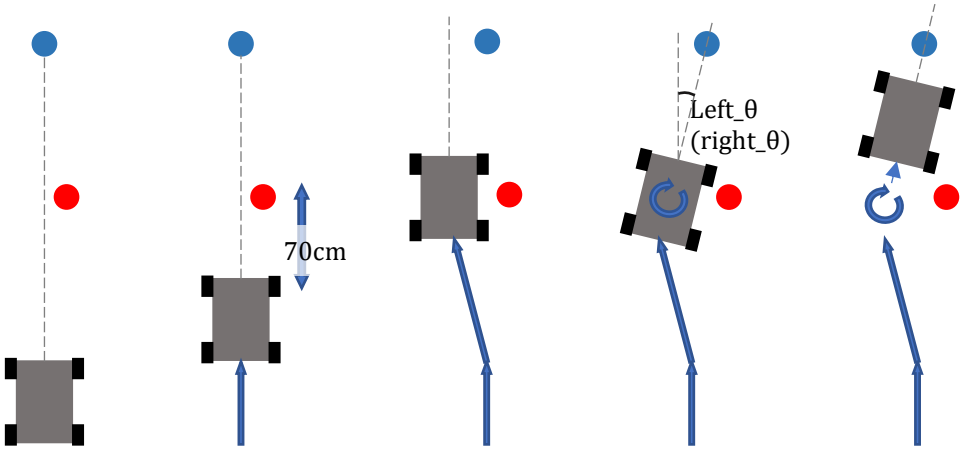
3.4.2.3.1 Move forward forcefully [1349-1359]

3.4.2.3.2 Increment variable ‘collect’ [1361]

3.4.3 파란 공을 향해 다가가는 도중에 빨간 공이 경로 상 존재하게 된다면 회피를 해야 한다. 이는 PickUp() 함수에서 결정하는 사안으로써 avoid_Redball() 함수가 실행된다.

3.5 void avoid_RedBall() [1373 - 1532]

3.5.1 본 차체에는 사선 이동을 통한 회피 모션을 사용했다. 다음은 이를 도식화한 것이다.



Red ball avoid motion

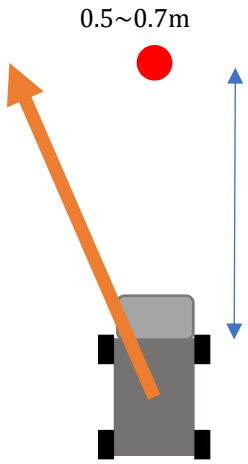
3.5.2 전체 프레임에서 가장 가까이 있는 공이 빨간 공인 경우, 그 공의 좌표가 중심부에 존재한다면 회피 모션이 개시된다. 만약 빨간 공이 가장 가깝지만 중심부에서 멀리 떨어져 있다면 굳이 피할 필요가 없다. 이는 for loop을 blue ball number, red ball number 만큼 돌리면서 sorting 할 수 있다.

3.5.3 파란 공을 향해 다가가는 도중 빨간 공이 경로 상 존재할 때

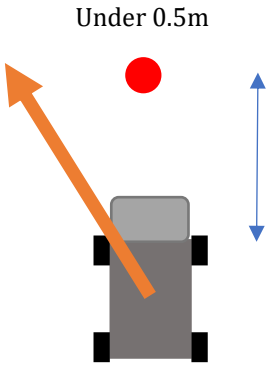
3.5.3.1 Red ball distance > 0.7m [1382-1389]

3.5.3.1.1 Move forward

3.5.3.2 Red ball distance < 0.7m [1392-1531]



Red ball avoid motion 1



Red ball avoid motion 2

빨간 공의 거리가 0.7m다 가깝지만 0.5m보다 먼 경우 좌우로는 조금 이동하고 앞으로 크게 이동한다. 빨간 공의 거리가 0.5m보다 가까운 경우가 있는데 이는 파란 공을 수집한 직후 회전하자 마자, 또는 빨간 공을 회피하자 마자 새로운 빨간 공이 앞에 있을 때에 해당한다. 이때는 앞으로 조금 이동하고 좌우로 크게 이동한다. 이를 통해 두 모션 모두 끝났을 때의 차체 위치는 빨간 공으로부터 동일한 거리만큼 옆에 위치하게 한다.

3.5.3.2.1 0 < X_position < 0.3m : 빨간 공이 차체 기준 오른쪽에 치우쳐져 있으므로 왼쪽으로 회피하는 것이 효율적이다. [1392-1461]

3.5.3.2.1.1 Left avoid

3.5.3.2.1.1.1 만약 Red ball Y_position < 0.5m 라면 Red ball avoid motion 2로 동작한다. [1396-1404]

3.5.3.2.1.1.2 0.5m < Red ball Y_position < 0.7m 라면 Red ball avoid motion 1로 동작한다. [1405-1413]

3.5.3.2.1.2 Turn right [1420-1426]

3.5.3.2.1.2.1 정면을 바라보는 상태에서 왼쪽으로 사선 이동하였으므로 기존에 align을 맞춰 두었던 파란 공이 아닌 다른 공이 중심부에 가장 가깝게 위치할 수 있다. 파란 공을 수집하는 메커니즘이 가장 중앙에 있는 파란 공을 향해 가는 것이기 때문에 기존에 따라가던 파란 공이 가장 중심에 오도록 오른쪽으로 아주 조금 회전한다. 이때 회전한 각도를 다음 차례의 rough angle 값에 더하여, 다음 stage에서 회전할 angle을 보정해준다. 또한 회전 후 파란 공이 중심 범위 내에 없다면 추가적인 보정 회전을 통해 중심에 오도록 한다.

3.5.3.2.2 -0.3m < X_position < 0 : 빨간 공이 차체 기준 왼쪽에 치우쳐져 있으므로 오른쪽으로 회피하는 것이 효율적이다. [1464-1531]

3.5.3.2.2.1 Right avoid

3.5.3.2.2.1.1 만약 Red ball Y_position < 0.5m 라면 Red ball avoid motion 2로 동작한다. [1468-1476]

3.5.3.2.2.1.2 0.5m < Red ball Y_position < 0.7m 라면 Red ball avoid motion 1로 동작한다. [1478-1485]

3.5.3.2.2.2 Turn left [1491-1496]

3.5.3.2.2.2.1 위와 마찬가지로 왼쪽으로 아주 조금 회전하여 기존의 파란 공을 바라볼 수 있도록 한다.

Left avoid 와 right avoid를 나눈 이유는 빨간 공이 왼쪽으로 치우쳐 져 있을 때 오른쪽으로, 오른쪽으로 치우쳐 져 있을 때 왼쪽으로 회피하여 최대한 효율적으로 적게 움직임을 가져가면서 회피하기 위함이다.

Avoid_RedBall() 함수 실행 도중 회피 모션이 끝났을 때 빨간 공 회피가 파란 공을 향해 가는 도중 발생했는지 초록 공을 향해 가던 중 발생했는지 구분하는 조건문이 있다. 초록 공을 향해 가고 있던 경우는 언제나 collect의 변수 값이 collect = 4일때이므로 if collect = 4라면 회피 이후 회전 없이 releaseAlign() 함수로 돌아간다. 그 외의 경우에는 no_red_ball() 함수로 돌아가 기존에 향하고 있던 파란 공을 수집하기 위한 동작을 수행한다.

3.6 void releaseAlign() [1098 - 1279]

3.6.1 Green ball 2개 중 왼쪽 공의 x,y 좌표를 각각 x1, y1, 오른쪽 공의 x,y 좌표를 각각 x2, y2라고 지정한다. [1107-1118]

3.6.2 세 번째 파란 공을 수집하고 PickUp() 함수에서 angle 4를 회전하여 초록 공을 바라보게 한다. 만약 path generation 대로 움직이지 않았다면 angle 4를 회전하였을 때 초록 공이 시야에 들어오지 않을 수 있다. 만약 green ball number = 0이면 green ball number = 1이 될 때까지 오른쪽으로 회전하도록 한다. 만약 270도 회전하는 동안 초록 공을 찾지 못한다면 그 이후로는 더 천천히 돌며 초록 공을 찾는다. 이는 빠르게 회전하면 카메라에서 멀리 있는 초록 공을 잘 디텍팅하지 못하는 경우가 있어 이를 해결하고자 함이다. [1120-1138]

3.6.3 초록 공을 향해 가는 도중 빨간 공이 시야에 들어온다면, 빨간 공을 회피해야 하는 조건에 들어간 경우 avoid_RedBall() 함수를 호출한다. avoid_RedBall() 함수 내에서 회피 이후 collect = 4일 때는 releaseAlign() 함수로 돌아오는 조건이 있어 회피 후 다시 현 위치로 돌아온다. [1141-1143]

3.6.4 Green ball number = 1인 경우 그 초록 공 하나를 중심에 오도록 바라보며 전진한다. 바구니에서 4m이상 떨어져 있을 때 가끔 초록 공 하나만을 인식하는 경우가 있었다. 한 번의 명령어 실행 이후 ros::spinOnce()를 통해 계속해서 새로운 정보를 받아들여, 하나의 공만 바라보며 다가가다 일정 거리 이내로 가까워지면 반드시 두 개의 초록 공이 모두 잡히게 된다. [1145-1164]

3.6.4.1 X_position < -0.4m : Turn left. [1145-1151]

3.6.4.2 X_position > 0.4m : Turn right [1152-1157]

3.6.4.3 -0.4m < X_position < 0.4m : move forward [1158-1163]

3.6.5 Green ball number = 2인 경우 원하는 path대로 움직이고 성공적으로 파란 공들을 모두 수집하였을 때 마지막에 angle 4를 회전하면서 초록 공 두 개를 모두 바라보게 된다. [1166-1279]

3.6.5.1 3m < y1+y2 : 바구니로부터 약 1.5m 이상 떨어진 거리에서는 둘 중 상대적으로 더 멀리 있는 공을 향해 중심을 맞춰 다가간다. 가까이 가서 정렬을 위해 회전할 때 시야에서 한 공이 사라지는 경우를 방지하기 위함이다.

y1>y2인 경우 x1의 좌표가 -0.25m <x1< 0.25m에 오도록 한 상태로 왼쪽 초록 공을 향해 다가간다. 반대로 y1<y2인 경우 x2의 좌표가 -0.25m < x2 < 0.25m에 오도록 한 상태로 오른쪽 초록 공을 향해 다가간다.[1166-1210]

3.6.5.2 0.6m < y1+y2 < 3m : 회전을 통해 차체가 벽을 수직으로 바라보게 한 뒤, 두 공의 중심선 상에 차체가 올 수 있도록 좌우로 이동하여 정렬한다. [1212-1250]

먼저 y1>y2+0.03m 일 때는 왼쪽 공이 멀리 있으므로 오른쪽으로 회전한다. 반대로 y2>y1+0.03m일 때는 오른쪽 공이 멀리 있으므로 왼쪽으로 회전한다. 두 공의 y좌표 차이가 0.03m이내로 되도록 회전한다는 조건을 통해 차체의 전진방향이 벽과 수직을 이루도록 한다. [1214-1227]

그 다음 두 공의 x좌표들을 이용하여 x1+x2<-0.04m인 경우 왼쪽으로 이동한다. 반대로 x1+x2>0.04m인 경우 오른쪽으로 이동하여 차체가 바구니 앞에 오도록 조정한다. 앞의 두 조건이 충족된 상태에서 바구니로부터 0.3m 앞까지 전진한다. [1229-1241]

3.6.5.3 y1+y2 < 0.6m : 차체가 바구니로부터 0.3m 앞에 왔다면 pick up module을 들어올리고 강제 전진하여 바구니 앞으로 오도록 한다. 그 다음 pick up module의 roller를 반대로 회전시켜 파란 공들을 바구니 안에 drop 한다. [1252-1279]

4. Desired coding schedules

Week 1 : Ubuntu 설치 및 ROS 설치

Week 2 : ROS system 이해와 창시구를 위한 개발환경 만들기

Week 3 : Terminal 명령어 익히고 C++ 연습

Week 4 : C++ 연습 및 Package를 만들기 위한 연습

Week 5 : C++ 연습 및 Node 간 메시지 전달 숙지

Week 6 : C++ 연습 및 Node 간 메시지 전달 숙지

Week 7 : Capstone 1 ROS에서 사용하는 기기 간 통신 방법 이해, 전진 및 후진과 같은 명령어를 통해 차체 움직여 보기.

Week 8 : Sensor data 처리 숙지 및 webcam data processing. 파란 공을 향해 회전, 좌우이동 및 전진을 통해 다가가는 것 구현, 빨간 공 회피 동작 구현.

Week 9 : Sensor data 처리 숙지 및 lidar data processing. SLAM 사용법 숙지.

Week 10 : RVIZ 및 Gazebo를 이용하여 simulation 구현.

Week 11 ~ Final : 실제 데모를 해보며 발생하는 여러 오류들을 수정하고 완성도를 높인다.

5. 시행착오

위에서 설명한 알고리즘을 토대로 데모 시뮬레이션을 여러 차례 시도했을 때 매번 예상치 못한 오류가 발생하였다. 빨간 공을 제대로 피하지 못하거나, 파란 공을 보지 못하고 지나치거나, 회전이 제대로 되지 않거나 또는 초록 공을 보고 바구니로 제대로 돌아오지 못하는 경우들이 있었다.

다음은 발생한 문제점들을 해결하기 위한 방법이다.

5.1 Move forward, move left, move right, turn left, turn right 등 모든 이동 명령어를 속도를 매우 세분화하여 각각의 상황에 따라 알맞은 속도를 지정해주었다. 각각의 명령어는 fast, medium, slow, very slow의 4단계로 나누어 사용하였다.

5.2 각각의 움직임이 실행되도록 하는 빨간 공, 파란 공의 좌표 값에 대한 조건들을 수정하였다. 예를 들어 빨간 공의 x좌표에 따라 전진 또는 좌우 이동 명령어가 결정되는데 그 기준에 해당하는 값을 거듭 수정하며 최적의 값을 찾기 위해 노력하였다. Pick up module이 지속적으로 동작하며 공을 수집하기 때문에 빨간 공을 피하지 않고 지나가게 된다면 그 또한 pick up module에 들어가게 된다. 혹은 정확하게 회피하지 못하고 공을 건드리면 건드려진 공이 움직이면서 파란 공들을 건드리는 문제가 발생할 수 있는데 이는 path generation을 토대로 움직이는 우리의 알고리즘 상 치명적인 문제를 발생시킨다. 이러한 이유로 더욱 정교한 움직임을 가져가기 위한 수치를 찾는 것이 매우 중요하다.

5.3 여러 함수에서 공통적으로 사용되는 조건 변수를 하나의 변수로 지정하여 한 번에 컨트롤 할 수 있게끔 만들었다. 예를 들어 빨간 공의 y좌표에 따라 차체가 전진 또는 회피 동작을 수행하는데 그 y값에 해당하는 수치를 lineRed 라는 변수에 지정하여 모든 함수에는 lineRed를 사용하고 int lineRed에 대한 기본 값을 가장 앞에 선언해주었다. 값의 수정이 필요할 때는 가장 앞에 선언한 int lineRed의 값을 수정함으로써 빠트리는 부분 없이 빠르게 수정할 수 있었다.

5.4 실행된 함수 및 명령어를 텍스트파일로 저장하여 디버깅할 때 사용하였다. cout을 사용하면 터미널창에 현재 실행중인 명령어를 알 수 있지만 NUC의 터미널창을 실시간으로 보기 쉽지 않고 또한 다음 시뮬레이션 실행 시 소멸되는 단점이 있다. 따라서 특정 명령어가 실행될 때 원하는 출력값을 .txt 파일로 저장하게끔 하고 촬영한 영상과 비교해보면서 디버깅하여 어떤 순간에 어떤 명령어가 실행되었고 어느 부분에서 문제점이 발생했는지 쉽게 알 수 있었다. .txt 파일로 저장하는 명령어는 다음과 같다.

```
string filePath = "/home/”username”/result.txt";
ofstream txtFile(filePath);.
```

5.5 명령 실행 이후 파란 공, 빨간 공의 정보가 초기화 되지 않고 그 전 값이 메모리에 남아있는 경우가 있었다. 두 번째 파란 공을 수집하고 나서도 두 번째 공의 위치 정보가 여전히 남아있어 세 번째 공을 보고 있음에도 두 번째 공의 위치 정보를 토대로 움직임을 가져가는 문제가 발생하였다. 이러한 경우는 주로 무한루프에 갇혀서 발생하였는데 4)번의 .txt파일로 저장하는 방법을 통해 어느 부분에서 무한루프에 갇혔는지 쉽게 알 수 있었다. 그리고 항상 조건문에서 지금 보이는 공이 0개인지 아닌지를 체크하도록 하여 이전 값을 사용해 함수가 실행되지 않도록 하였다.

5.6 Ball detect node에서 매 프레임마다 RGB image와 Depth image를 비교하는 과정을 거치다 보니 연산이 생각보다 오래 걸렸다. Real sense node에서 보내주는 RGB image와 Depth image는 30Hz로 ball detect node로 들어오는데, ball detect node에서 처리 후 보내는 ball position message는 6~7Hz로 나갔었다. ROS의 loopRate은 20Hz로 작동하지만 ball detect node에서 송신하는 데이터가 6~7Hz 정도로 낮으니 공의 위치 정보가 빠르게 업데이트 되지 않고 지연되었다. 이로 인한 delay는 system의 stability를 떨어트렸고, 예를 들어 Angle1~3을 회전하고 난 뒤 파란 공이 중앙에 들어왔음에도 불구하고 그 전 값을 여전히 가지고 있어 추

가로 더 회전하는 문제를 야기했다. 이를 해결하기 위해 애초에 ball detect node로 보내는 image의 해상도를 1920*1080에서 1280*720으로 변경하여 연산 처리량을 대폭 감소시켰다. 그러자 ball detect node에서 메시지가 15Hz의 속도로 data integration node로 송신되어 위의 문제를 해결할 수 있었다.