2019 Spring ME492 Final Report

Team F

정민기, 20130582

이인표, 20140735

염동훈, 20150475

이진섭, 20160524

2019.06.16

1. introduction

ROS는 Robot Operating System의 약자로 robot의 많은 하드웨어들, 즉 하드웨어 드라이버들을 연결해 통합 관리하는 system이다. 그래서 이 하드웨어들 사이의 통신이 매우 중요한데 이 통신 상태 및 관계를 나타내어주는 rqt_graph를 시작으로 전체 알고리즘 설명 후 공을 인식하여 position 정보를 나타내는 ball detect node, 자율 주행 알고리즘이 구현되어 있는 data integration node 순으로 보고서를 진행하려고 한다.

2. rqt graph 및 Overall Algorithm

2.1 rqt graph

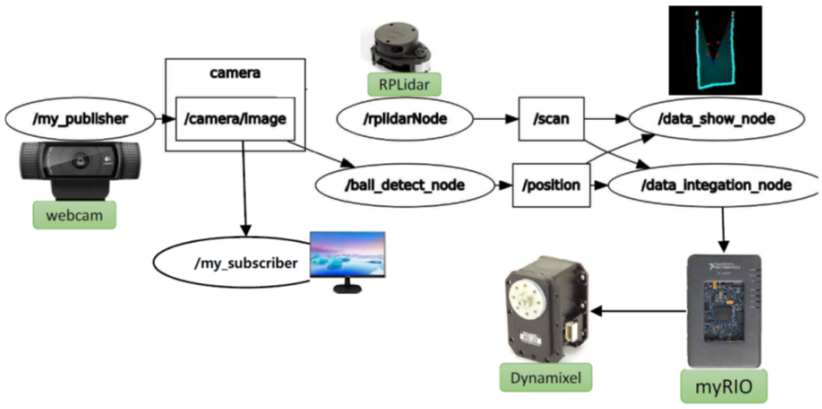


Figure 1 기본 코드의 rqt_graph

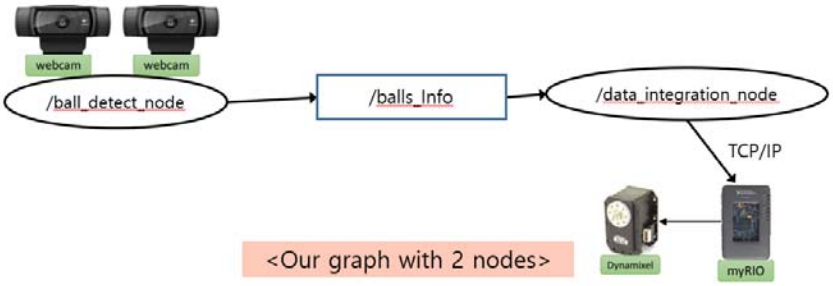


Figure 2 우리 코드의 rqt_graph

나눠준 기본 코드의 rqt_graph는 Figure1과 같다. Webcam의 영상을 받아오는 /my_publisher node와 이를 확인하는 /my_subscriber node, 이로부터 영상 이미지를 받아와 이미지 프로세싱을 하는 /ball_detect node, RPLidar를 사용할경우 추가되는 /rplidarNode 그리고 Lidar와 webcam의 이미지를 받아와 로봇의 행동을 제어하는 /data_integration_node, 두 데이터를 받아와 데이터지도를 보여주는 /data_show_node가 존재한다. 하지만 우리의 경우 /ball_detect_node와 /data_integration_node만 사용하기로 하였다. 그 이유는 웹캠의 영상을 받아와 다시 msg로 보내 다른 node에서 굳이 이미지 프로세싱을 할 필요가 없다. 영상을 받아오는 대로 바로 그 노드에서 이미지 프로세싱을 하면 되기 때문에 /my_publisher node와 /my_subscriber node를 제외하여 /ball_detect_node에서 영상을 받아오자마자 바로 이미지 프로세싱을 하도록 하였다. 또한 우리의 설계철학에 맞게 최대한 simple하게 RPLidar를 사용하지 않고 webcam의 상대적 위치만으로 모든 행동을 제어할 수 있도록 하였다. 따라서 /rplidarNode 또

한 사용하지 않아 결과적으로 Figure2에서 볼 수 있듯이 2개의 Node만을 사용하는 코드를 작성하였다.

2.2 Overall Algorithm

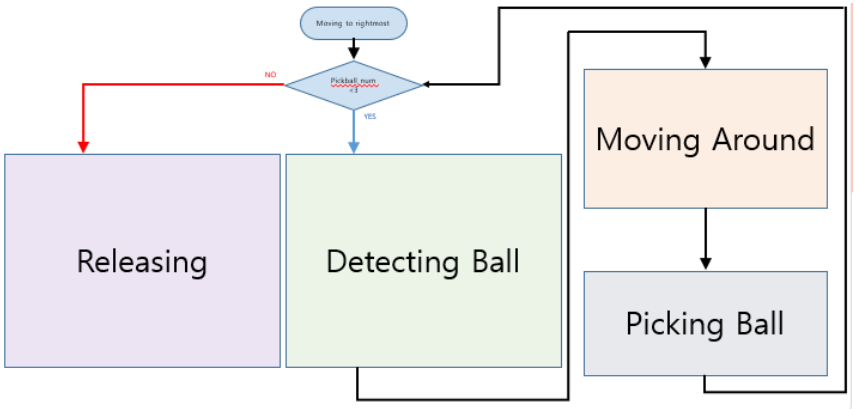


Figure 3 Overall Algorithm

전체 알고리즘은 Figure 3과 같다. 우리 조는 가장 오른쪽에 있는 공부터 시계 반대방향으로 돌며 pick up하도록 알고리즘을 생각했기 때문에 시작하면 시계방향으로 살짝 회전한다. 회전한 후 (pick_ball_num)이라는 정수의 크기를 비교하여 detecting ball로 넘어간다. Detecting ball에서는 파란 공이 탐지될 때까지 회전하며 탐지되면 align한다. Moving around에서는 align한 파란 공을 향해 이동하며 만약 경로에 빨간 공이 있다면 회피한다. 마지막으로 Picking ball에서는 두번째 webcam을 통해 파란 공을 줍고 (pick_ball_num)을 1 증가시킨다. 마지막으로 (pick_ball_num)이 3이상이 되면 Release 단계로 넘어가서 바구니로 돌아간 후 공을 내려놓는다.

2.2.1. Detecting ball

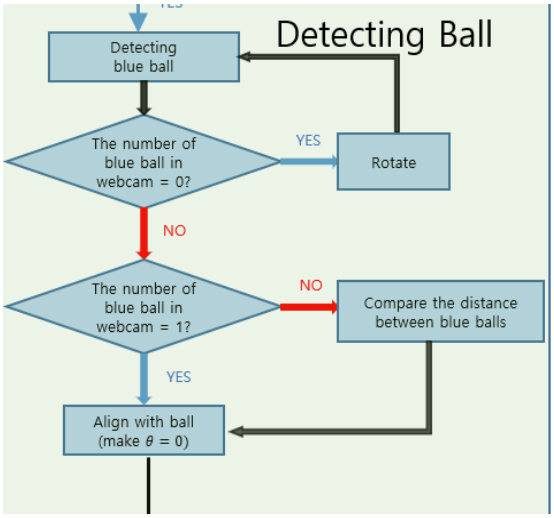


Figure 4 Detecting Ball

공을 탐지하는 과정은 위와 같다. 만약 webcam에서 탐지되는 파란 공이 없다면 webcam에 공이 탐지될 때까지 회전한다. Webcam에서 파란 공이 한 개만 인식되면 그 공에 align을 한다. 만약 2개 이상의 파란 공이 인식된다면 때에 따라 원하는 공을 목표로 align한다.

2.2.2. Moving Around

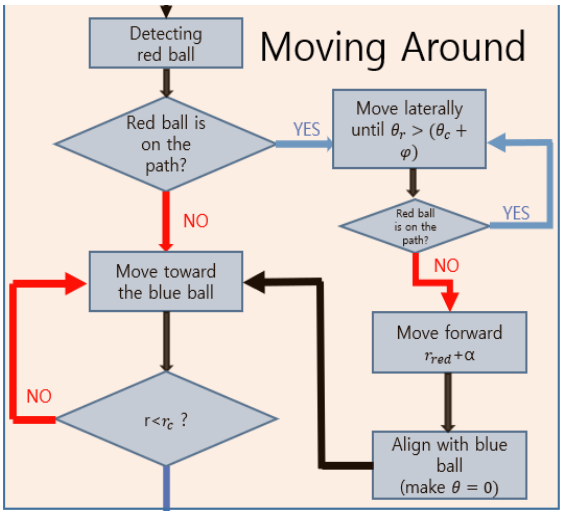


Figure 5 Moving Around

Moving around에서는 detecting ball 단계에서 align한 공을 향한 이동과 빨간 공을 피하는 움직임이 있다. 우선 경로상에 빨간 공이 없다면 공과의 거리가 일정 거리가 될 때까지 이동한다. 만약 경로상에 빨간 공이 있다면 좌우로 회피하여 움직인다.

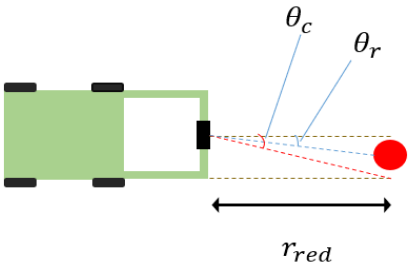


Figure 6

공을 회피하는 거리는 Figure 6와 같이 빨간 공과의 각도인 θ_r과 θ_c를 비교하여 θ_r이 θ_c보다 일정 각도 이상 커질 때까지 회피하도록 만들었다. 공을 회피한 후에는 r_{red}만큼 움직인 후 원래 목표했던 파란 공과 다시 align한 후 다시 파란공을 향해 움직인다.

2.2.3. Picking Ball

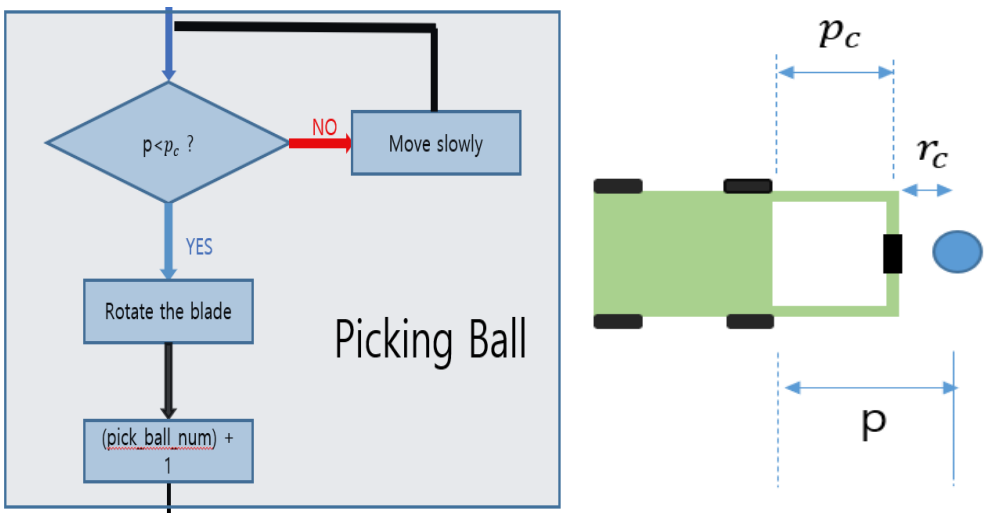


Figure 7 Picking Ball

Picking ball 단계에서는 원래 보던 webcam이 아닌 두번째 webcam을 사용한다. p_c 와 P의 크기를 비교하여 P가 p_c보다 크면 천천히 움직인 후 P가 더 작아지면 blade를 회전해 공을 pick up 하고 (pick_ball_num)을 1 증가시킨다.

2.2.4. Releasing

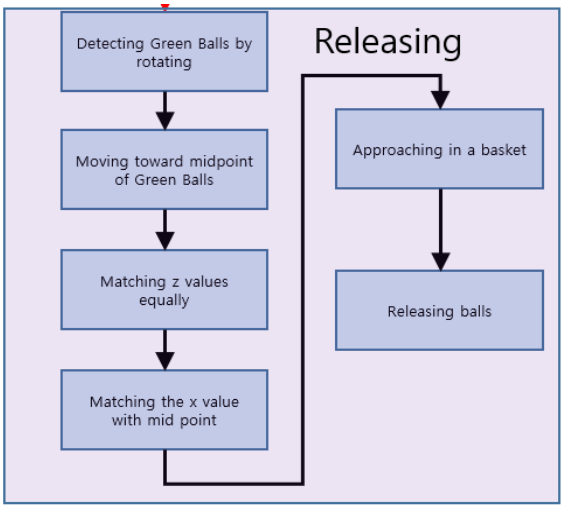


Figure 8 Releasing

마지막 releasing 단계에서는 초록색 공 두개가 모두 detect될 때까지 회전한다. 공 두개가 detect되면 두 공의 중심을 맞추고 일정 거리만큼 이동한다. 바구니와 어느정도 가까워지면 두번째 webcam을 이용하여 하나의 초록색 공에 다시 align을 하고 이동하여 파란 공을 release한다. 이때 우리는 초록색 공 하나에 맞춰 release하기 때문에 간혹 실패할 때가 있어서 release하는 중에 천천히 앞으로 움직인 후 다시 뒤로 이동하도록 만들었다.

3. ball_detect_node

3.1. webcam의 위치 및 좌표축 변환

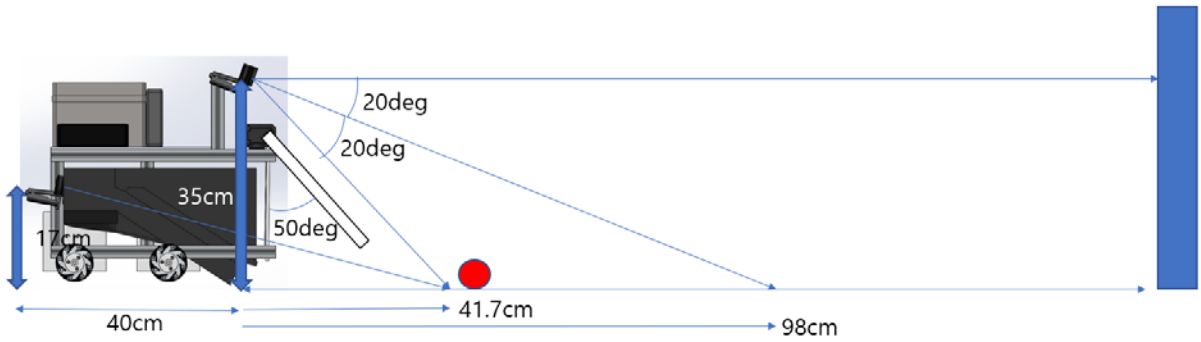
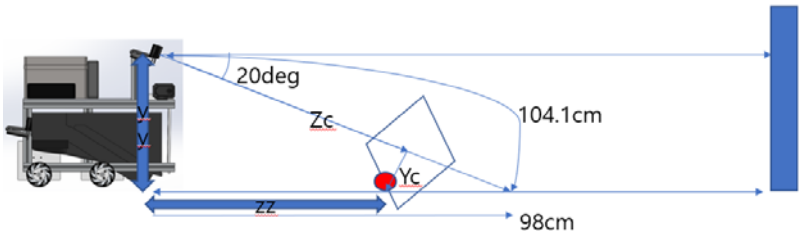


Figure 9 webcam의 위치

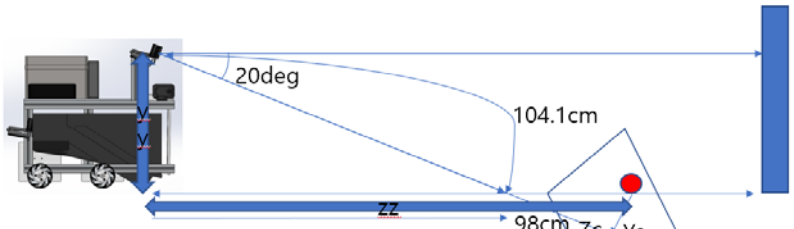
위 그림은 우리 모델의 옆모습이고 webcam이 위에 1개, 아래 1개 달렸음을 알 수 있다. webcam1의 경우 아래로 20도만큼 각도를 내렸기 때문에 원래 webcam의 좌표계와 달라지게 된다. 따라서 다음 그림들과 같이 좌표축 변환을 해줬다.



$$yy = Zc \times \sin(tan^{-1}(\left(\frac{Yc}{Zc}\right) + 20°)$$

$$zz = Zc \times \cos(tan^{-1}(\left(\frac{Yc}{Zc}\right) + 20°)$$

Figure 10 좌표축 변환



$$yy = \sqrt{Zc^2 + Yc^2} \times \sin(20° - tan^{-1}(\left(\frac{Yc}{Zc}\right))$$

$$zz = \sqrt{Zc^2 + Yc^2} \times \cos(20° - tan^{-1}(\left(\frac{Yc}{Zc}\right))$$

Figure 11 좌표축 변환

3.2. webcam으로 얻어낸 위치 정보 반환하기

웹캠을 통해 위치 정보를 얻어낸 후 여러 공의 위치 정보를 모두 가지고 있는 벡터를 만들어야 한다. 이 함수를 구현한 것이 Figure12이다.

```
vector<vector<float>> position_info( vector<float> radius, vector<Point2f> center, Scalar color, int &ball_num, Mat &result){
    ball_num = 0;
    vector<vector<float>> ball_positions;
    for( size_t i = 0; i< center.size(); i++ ){
        if (radius[i] > iMin_tracking_ball_size){
            vector<float> ball_position;
            ball_position = pixel2point(center[i], radius[i]);
            ball_positions.push_back(ball_position);

            //여기서부터는 공의 위치를 영상에 나타내는 코드로 중요하지 않다.
            circle( result, center[i], (int)radius[i], color, 2, 8, 0 );
            float isz = ball_position[2];
            string sz = floatToString(isz);
            text = "DISTANCE:" + sz;
            for( size_t i = 0; i< ball_positions.size(); i++ ){ //거리가 너무 먼 공에 대해서는 표시안하는 코드이다.
              if(ball_positions[i][0]>3 || ball_positions[i][1]<0 || ball_positions[i][2]>6){
              }
              else{
                putText(result, text, center[i],2,1,Scalar(255,255,255),2);
              }
            }
            ball_num++;
        }
    }
    return ball_positions;
}
```

Figure 12 position info를 얻어내는 코드

이 함수를 통해 각각의 중심과 반지름에 대해 위치 정보를 받아내고 공의 xyz위치 정보를 각각의 요소로 가지고 있는 vector<vector<float>> ball_positions에 저장해 이를 반환한다.

Webcam1과 2가 서로 다른 좌표를 사용하므로 이 또한 웹캠에 따라 서로 다른 position_info함수를 사용한다. 각각의 공의 위치 정보를 모두 얻었으니 이를 msg로 보내는 함수만 작성하면 된다.

3.3. msg로 공의 위치 정보 보내기

```
void pub_ball_info(int ball_num, vector<vector<float>> ball_position, ros::Publisher markers, int color){
    core_msgs::ball_position ball_list;  //declare marker

    //웹캠1의 경우 x가 3m보다 크거나, y가 0보다 작거나, z가 0.01m보다 작거나 6m보다 클 경우가 없기 때문에 이 경우 생기는 에러를 보내지 않는코드이다.
    for( size_t i = 0; i< ball_position.size(); i++ ){
      if(ball_position[i][0]>3 || ball_position[i][1]<0 || ball_position[i][2]>6 || ball_position[i][2]<0.01){
        ball_position.erase(ball_position.begin()+i);
        ball_num--;
      }
    }
    if(ball_num){
        for(int k=0;k<ball_position.size();k++){
            ball_list.img_x.push_back(ball_position[k][0]);
            ball_list.img_y.push_back(ball_position[k][1]);
            ball_list.img_z.push_back(ball_position[k][2]);
            ball_list.size = ball_num;
        }
    }
    markers.publish(ball_list);
}
```

Figure 13 공의 위치정보를 보내는 코드

여러 공의 위치정보를 가지고 있는 벡터인 ball_position과 Publisher를 받아와 함수를 실행한다. Webcam 1의 경우 x값이 3m이상, y값이 0이하, z값이 6m이상일 경우 데모장 밖이므로 이 경우는 정보를 보내지 않도록 벡터에서 erase함수를 통해 지워주고, z값이 0.01보다 작은 에러가 detect됫을 경우에도 마찬가지로 지워준다. 그리고 core_msgs 형식의 ball_list를 정의해 x,y,z값과 detect된 공의 개수를 msg를 통해 publish하게 된다.

```
void pub_ball_info2(int ball_num, vector<vector<float>> ball_position, ros::Publisher markers, int color){
    core_msgs::ball_position ball_list;  //declare marker
    //x가 0.5m보다 크거나 y가 0보다 작거나 z가 0.2m보다 작거나 3m보다 큰 공에 대해서는 메세지를 보내지 않도록 벡터에서 지우는 코드이다.
    for( size_t i = 0; i< ball_position.size(); i++ ){
      if(ball_position[i][0]>0.5 || ball_position[i][1]<0 || ball_position[i][2]>3 || ball_position[i][2]< 0.2){
        ball_position.erase(ball_position.begin()+i);
        ball_num--;
      }
    }
    if(ball_num){
        for(int k=0;k<ball_position.size();k++){
            if(ball_position[k][2]<3){
            ball_list.img_x.push_back(ball_position[k][0]);
            ball_list.img_y.push_back(ball_position[k][1]);
            ball_list.img_z.push_back(ball_position[k][2]);
            cout<<ball_position[k][2]<<endl;
            ball_list.size = ball_num;
          }
          else{
          }
        }
```

Figure 14 웹캠2의 공의 위치정보를 보내는 코드

Webcam2의 경우 왼쪽에 공을 저장하는 부분이 있으므로 x값이 0.5m보다 클 경우, y값이 0보다 작을 경우, z값이 3m보다 클 경우, 0.2m보다 작을 경우 공의 정보를 지우도록 한다. 그리고 위와 마찬가지로 이를 msg에 담아 publish하는 것이다. 이렇게 함수로 작성된 코드 말고 실제 공의 x, y, z정보를 구하고 publish하는 코드의 경우 Figure15을 보면 된다.

정리하자면 ball_detect_node에서는 camera calibration을 통해 camera matrix를 구한다. 다음 데모 환경에 맞는 각각의 색깔의 공의 hsv값 범위를 구한다. 다음 이미지 프로세싱을 통해 공의 컨투어를 찾아내 이 외접원의 중심과 반지름을 구하고 이를 camera matrix를 이용해 x,y,z 값을 구해내 msg에 담아 publish하는 것이 ball_detect node에서 하는 모든 것이다.

```
vector<vector<float>> ball_position_r = position_info(radius_r, center_r, color_r, ball_num, result);
pub_ball_info(ball_num, ball_position_r, redball_info, RED);

vector<vector<float>> ball_position_b = position_info(radius_b, center_b, color_b, ball_num, result);
pub_ball_info(ball_num, ball_position_b, blueball_info, BLUE);

vector<vector<float>> ball_position_b2 = position_info2(radius_b2, center_b2, color_b, ball_num, result2);
pub_ball_info2(ball_num, ball_position_b2, blueball_info2, BLUE);

vector<vector<float>> ball_position_g = position_info(radius_g, center_g, color_g, ball_num, result);
pub_ball_info(ball_num, ball_position_g, greenball_info, GREEN);

vector<vector<float>> ball_position_g2 = position_info(radius_g2, center_g2, color_g, ball_num, result2);
pub_ball_info(ball_num, ball_position_g2, greenball_info2, GREEN);
```

Figure 15 각 색깔의 공의 정보를 보내는 코드

4. data_integration_node

data_integration_node에는 우리가 구현하고자 하는 모든 알고리즘, 함수, 변수 등등이 있다. 먼저 우리가 만든 함수, 변수 등을 먼저 설명하고 알고리즘은 그 다음에 설명하도록 하겠다.

4.1. Ball Data Function

공 인식을 위해 open cv에서 이미지 처리를 해 공의 position 정보를 보내주면 이를 받는 함수가 callback함수들이고 로봇이 과제를 수행할 수 있게 가장 가까운 공의 정보를 골라주는 find_nearest_ball_position 함수, 가장 오른쪽 공의 정보를 골라주는 find_rightest_ball_position 함수 마지막으로 거리가 0이라고 뜨는 error를 잡아주는 remove_error_nearest함수를 정의해 사용했다.

4.1.1. Global Variables

Line : 42~90가 Open CV에서 이미지 처리를 통해 얻은 공들의 position 정보들을 받는 변수들이다. 순서대로 인식 되는 공의 개수, x좌표, y좌표, z좌표, x축과의 각도, 거리 변수들이고 이 정보들을 담고 있는 벡터 변수 posi, 가장 가까운 공의 벡터 변수 nearest posi 순으로 선언되어 있다. 이러한 변수들이 빨강, 파랑, 초록 그리고 2번째 카메라의 파랑, 초록 볼에 대하여 총 5개의 묶음으로 선언되어 있다. 중간에 다른 배열의 크기는 다 20인데 blue ball2만 90인 이유는 pick up module 안쪽에 설치되어 있는 웹캠 2 때문인데 공을 주우면 공이 안으로 들어와 카메라와 매우 가까워지는 순간이 존재하고 이때 opencv에서 매우 많은 공으로 인식하는 경우가 있어 segmentation error를 방지하기 위해 크기를 90으로 잡았다.

4.1.2. Callback Function

Red ball, blue ball, blue ball2(by 2nd camera), green ball, green ball2(by 2nd camera), 총 5개의 callback함수로 매 loop마다 공의 position 정보들을 갱신한다. Line 287~313을 주석과 함께 보면 잘 알 수 있듯이 msg를 통해 받은 x, y, z, theta, distance 정보를 순서대로 벡터에 넣는 함수인데 먼저 갱신전에 posi과 nearest posi 벡터를 clear함수로 초기화 하고 ball_number만큼 for문을 돌리며 p 벡터에 넣고 이렇게 만들어진 p 벡터를 벡터로 이루어진 posi 벡터에 넣는다. 마지막으로 가장 가까운 공의 정보가 필요한 경우 find_nearest_ball_position 함수를, 가장 오른쪽 공의 정보가 필요한 경우 find_rightest_ball_position 함수를 사용해 정보를 저장해둔다. 이후 callback function의 구조는 모두 같다.

4.1.3. Nearest & Rightest

Callback function에서 쓰이는 nearest 그리고 rightest 함수는 모두 우리가 정의한 것이고 이 내용은 line 184부터 확인할 수 있다. Callback 함수에서 만든 공들의 정보가 담겨있는 posi 벡터를 입력 변수로 넣어주면 공의 개수만큼 for문을 돌리면서 앞에서부터 차례대로 두 공의 distance를 비교하여 더 작은 값의 공을 저장하는 방식으로 nearest 공을 골라낸다. 또한 거리가 0으로 뜨는 에러를 거르기 위해 거리가 0.01보다 클 때 실행되도록 if문을 중간에 추가했다. Rightest 함수도 같은 알고리즘으로 각도를 비교하여 큰 값의 공을 저장하여 rightest 공을 골라낸다. 웹캠 2 함수의 경우 카메라가 뒤집혀서 설치되어 있기 때문에 부호가 수정된 것 이외에는 같은 코드이다.

4.2. Moving Function

모터의 동작을 위해서 랩뷰에 동작 정보를 알려줘야 하는데 우리는 원래 xbox가 보내는 신호 체계를 그대로 이용하여 data[24]에서 필요한 data값만 변경시키는 것으로 모터를 조작했다.

4.2.1. Mechanism

이를 위해 data[24]를 float형으로 전역변수로 선언하였다. 또한 동작 종류마다 각 바퀴의 회전 조합이 다르므로 각 상황에 필요한 data값을 바꿔주는 이동함수들을 만들어 사용했는데 초기에는 여러 data를 사용하였지만 속도 값을 ROS에서 보내주는 등 여러 변화를 거쳐 최종적으로 data[0], [1], [4], [6], [8], [9], 총 6개를 사용했고 순서대로 좌우 직선 이동, 전진 후진, 좌우 회전, 속력, release, pick up 상태를 나타낸다. 속력의 최대 값은 55로 여러 단계로 나눠서 함수를 만들기도 하고 전진과 회전을 합친 함수를 만드는 등 총 24개의 이동함수를 만들었다.

4.2.2. Trajectory Funciton

로봇의 움직임을 부드럽게 만들기 위해 경로함수를 만들어 사용했다. 속력이 제어값이므로 처음과 끝의 가속도가 0이 되도록 하기 위해 3차 다항식을 경로로 이용해 path_3함수를 만들었다. 입력 변수는 v0, vf, kf, k로 순서대로 초기 속력, 목표 속력, 도달 시간(루프 수), 현재 시간(루프 수)을 나타낸다. 그리고 속력 제한을 위해 k가 kf를 넘으면 k값을 kf로 고정시키는 if문을 추가했다. 모든 동작에 넣을 생각이었지만 생각보다 시각적으로 크게 보이지 않았고 속력이 점진적으로 변하면서 기존 코드의 상수들과 맞지 않아 실제 데모에서 pick up에서만 경로함수를 사용했다.

4.3. Extra Function

마지막으로 dup_go_straight 함수는 파란공 2개가 일렬로 있어 카메라 상에서 커다란 공 1개로 인식되어 회피한 후 일정 거리 앞으로 가기 위해 while문으로 50회의 루프동안 go_straight_slowly를 실행시키도록 한 함수이며 open loop이므로 while문 안에서 write함수로 data를 매 루프마다 랩뷰에 송신했다.

4.4. Loop Rate

Loop rate.sleep 은 loop의 진동수가 지정된 진동수(주기)를 유지시켜주는 함수인데 정확히는 지정된 진동수보다 높아지는 것을 막아주는 것이다. 따라서 loop rate가 20일 때 sleep은 loop가 초당 20회이하가 되도록 유지시켜주는 역할이다. 중간 계산이 길어지거나 코드가 매우 길어지게 되어 loop time이 커지면 진동수가 낮아지게 된다. 그러므로 loop rate가 너무 크면 매 loop마다 loop time이 달라져 의미가 없고 loop rate가 너무 작으면 data의 갱신이 너무 느려져 로봇의 반응이 느려진다. 따라서 우리는 실험을 통해 20이 우리 코드에 적합하다고 보고 open cv와 함께 loop rate을 20으로 정하였다.

4.5 알고리즘

우리는 알고리즘을 위에서 이미 간단하게 설명하였다. 이제 여기서는 알고리즘을 실제로 코드로 적용할 때 어떻게 적용하였는지. 특징은 무엇인지. 코드는 어떻게 구현되었는지 자세히 설명하려고 한다.

4.5.1. 알고리즘 특징

먼저, 우리의 알고리즘은 앞에서 설명한 것과 같다. 코드도 그 알고리즘을 바탕으로 작성했다. 우리 코드의 가장 큰 특징은 switch문을 써서 각 단계를 나누고 그 단계의 것만 실행되게 했다는 점이다. 픽업 해야 하는 상황이면 detecting ball → moving around → picking ball 순으로 단계가 진행되는데 우리는 이것을 case1, case2, case3로 나눠서 각 단계에서 문제가 생기면 그 전 단계 혹은 그 전전단계로 다시 돌아가게끔 만들었다. 이 것은 우리가 코드를 작성할 때 편리하고 오류 찾을 때도 어디서 오류 났는지 cout을 통해 쉽게 알 수 있게 해주었다. 예를 들어 detecting ball 단계에서 공을 찾고 align까지 하여 moving around 단계까지 넘어갔는데 카메라에서 공을 찾지 못하는 상황이 오면 다시 case1으로 돌아가 공을 다시 찾게 한다. 마찬가지로 picking ball 단계에서도 공을 잃어버리면 다시 case1으로 보낸다. 이런 유연성이 우리 모델의 성공 가능성을 높여주었다.

두 번째 특징은 움직일 때 직선운동 하다가 회전해야 하는 상황이 오면 멈춰 서서 회전만 하는게 아니라, 실제 차가 좌회전 우회전하듯이 앞으로 가면서 회전하는 것이다. 처음에는 정확성을 높이기 위해 회전해야 하는 상황이 오면 멈춰 서서 회전하였지만, 진행하면서 충분히 코드가 안정성이 높음을 확인하였고, 수행시간을 줄이기 위해 앞으로 가면서 회전하는 것을 넣었다. 이것은 시간을 단축해주었을 뿐만 아니라 앞으로 가던 모터가 갑자기 멈추고 다시 작동하는 문제를 없앴고 모터에 무리가지 않게 하였다.

세 번째 특징은 차가 움직이면서 align이 맞지 않으면 계속 좌회전 우회전 하면서 앞으로 가 계속 align하면서 간다는 점이다. 이는 회전할 때 멈추지 않기에 발생하는 또다른 장점이고, 이로 인해 멀리서 align을 하더라도 다가갈 때 공을 잃어버리는 현상을 없앨 수 있었다.

네 번째 특징은 비전 센서 즉, 카메라만으로 모든 알고리즘을 구현했다는 점이다. 우리는 20Hz로 카메라가 이미지를 받아들이고 있는 것을 이용하여, 매 루프마다 공과 카메라의 상대

적인 좌표를 갱신하였고 이는 꽤나 정확한 공의 좌표를 이용할 수 있었다. 따라서, 다른 센서는 필요하지 않았고 우리의 철학인 simple & compact는 필요하지 않은 것은 부착하지 않는 것이기에 우리 철학에도 부합했다.

다섯 번째 특징은 카메라2를 써서 picking ball 단계에서는 여기서 받아온 데이터를 썼다는 점이다. 우리는 카메라2를 아래쪽에 달고서 가까이 공이 있을 때도 계속 비전으로 확인할 수 있게 하였고 실제로 blade를 돌리거나 올려서 pickup 또는 release할 때 공을 끝까지 볼 수 있게 하였다.

여섯 번째 특징은 수많은 변수 값들, 예를 들어 어느 정도의 거리에서 blade를 돌려 pickup을 할지 등등 모두 실험을 통해 수치를 정했다는 점이다. 따라서 많은 실험이 필요하였고 미세하게 1cm 단위로 조정하였다.

일곱 번째 특징은 바퀴를 구동하는 모터는 open loop이 하나도 없다는 점이다. 우리는 이를 위해 카메라2를 pickup 또는 release하기 직전까지 쳐다보게끔 했고 바퀴는 항상 비전 센서를 통해 움직이기 때문에 정확성을 높일 수 있었다.

4.5.2. main 함수 내부 큰 while문 내부 코드 스케치

다음은 메인 함수 내부에 있는 while문을 크게 스케치하고 간단하게 설명한 것이다. 실제로 코드가 이런 형식으로 구현이 되어 있고 움직일 때는 정의했던 움직임 함수를 사용하였다. 자세한 코드, 예들 들어 MOVING A 코드, moving b 코드, pickup 코드 등은 스크린샷을 통해 뒤에서 자세히 설명한다.

픽업한 공의 개수가 0일 때와 1~2개 일 때를 나눠서 작성했는데, 이는 픽업한 공의 개수가 0일 때는 목표 공이 가장 오른쪽에 있는 파란공이고 1~2개 일 때는 가장 가까운 파란공이기 때문이다. 이는 전체적으로 반시계방향으로 돌면서 파란공을 픽업하기 위해 만든 알고리즘이다.

```
while(ros::ok)
ros::spinOnce()
if (픽업한 공의 개수가 0일 때)
    case1 (가장 오른쪽 파란공으로 align하는 것이 목적)
        if (카메라1에 파란공이 안보일 때) → 회전
        else (카메라1에 파란공이 보일 때) → 가장 오른쪽 공으로 align
    case2 (case1에서 align한 공 바로 앞까지 moving하는 것이 목적)
        if (카메라1에 파란공이 안보일 때) → case1으로 돌려보냄
        else if (가장 오른쪽 파란공까지 가는 경로 안에 다른 파란공 있을 때)
```

→ (이 경우 가장 오른쪽 공이 목표가 아니라 가는 경로 안에 있는 파란공이 목표가 됨)

if (카메라1에 빨간공이 보일 때) MOVING A 코드

if(가장 가까운 빨간공이 가장 가까운 파란공보다 가까이 있을 때)

→ 경로에 빨간공이 있으면 옆으로 피하고, 없으면 moving b 코드 실행

else(가장 가까운 빨간공이 가장 가까운 파란공보다 멀리 있을 때)

→ moving b 코드 실행

else (카메라1에 빨간공이 안보일 때) → moving b 코드 실행

else (가장 오른쪽 파란공까지 가는 경로 안에 다른 파란공이 없을 때)

→ (이 경우 case1에서 align한 그대로 가장 오른쪽 파란공이 목표가 됨)

MOVING A 코드

→ (다만 목표공이 가장 오른쪽 공이기 때문에 nearest_b_posi를 모두 rightest_b_posi로 바꿔줌)

case3 (Blade를 돌려서 실제로 공을 픽업하는 것이 목적)

→ pickup 코드 실행

else if (픽업한 공의 개수가 3개 미만일 때) (즉, 1~2개 픽업 했을 때)

case1 (가장 가까운 파란공으로 align하는 것이 목적)

if (카메라1에 파란공이 안보일 때) → 회전

else (카메라1에 파란공이 보일 때) → 가장 오른쪽 공으로 align

case2 (case1에서 align한 공 바로 앞까지 moving하는 것이 목적)

if (카메라1에 파란공이 안보일 때) → case1으로 돌려보냄

else (카메라1에 파란공이 보일 때)

MOVING A 코드

case3 (Blade를 돌려서 실제로 공을 픽업하는 것이 목적)

→ pickup 코드 실행

else (공을 3개 모두 픽업 했을 때) (즉, release모드 실행)

case1 (초록공 2개 detect 하고 g_dis까지 간 다음 x,z align)

case2 (align 된 상태로 g_c까지 전진)

case3 (relase)

4.5.3. 스크린샷을 통한 자세한 코드 설명

스케치한 부분과 스크린샷을 통해 코드를 자세히 설명하겠다. 스크린샷에서 주석부분은 길어서 안보일 수 있으니 여기서는 코드 위주로 본다. 코드에서는 주석을 매우매우 자세하게 달아 놓았으니 참고해주시면 됩니다

4.5.3.1

if (픽업한 공의 개수가 0일 때)

case1 (가장 오른쪽 파란공으로 align하는 것이 목적)

if (카메라1에 파란공이 안보일 때) → 회전

else (카메라1에 파란공이 보일 때) → 가장 오른쪽 공으로 align

```
if(pick_ball_num == 0){ // 픽업한 공의 개수가 0개 일 때
  switch(phase){ // phase의 초기값은 글로벌 변수 선언에서 1로 설정되어 있음

    case 1: // align 단계, 즉, 차체가 가장 오른쪽에 있는 파란공을 파라보게끔 만드는 단계
    if(b_ball_number == 0){ // 카메라1에 보이는 파란공의 개수가 0일 때, 카메라1은 위쪽에
      if(m == 0){ // case2에서 빨간공을 회피할 때 차체가 오른쪽 또는 왼쪽으로 움직이게 ㅁ
                  turn_ccw(); // 시계반대방향으로 회전
      }
      else{ // m이 1일 때
                  turn_cw(); // 시계방향으로 회전
      }
    }
    else{ // 파란공의 개수가 0이 아닐 때
      if(abs(rightest_b_posi[3]) > theta_c_i){ // 가장 오른쪽에 있는 파란공과 차체가 바라
        if(rightest_b_posi[3] > 0){ // 파란공이 차체가 바라보고 있는 방향 기준으로 오른쪽
                    turn_cw_slowly(); // 시계방향으로 천천히 회전
        }
        else{ // 파란공이 차체가 바라보고 있는 방향 기준으로 왼쪽에 있을 경우
                    turn_ccw_slowly(); // 시계반대방향으로 천천히 회전
        }
      }
      else{ // 위의 두 경우가 아닌 경우, 즉, align이 끝난 경우
        phase = 2; // phase를 2로 설정하여 case2가 실행되도록 한다
      }
    }
    break; // break하여 swith문을 빠져나온다
```

Figure 16

먼저 픽업한 개수가 0일 때와 1~2개일 때를 나눈 것은 맨처음에는 가장 오른쪽 공을 집어 크게 반시계방향으로 공을 최단시간으로 줍기 위해서이다.

여기서는 픽업한 공의 개수가 0일 때 phase가 글로벌 변수에서 1로 선언되어 있기에 가장 먼저 case1이 실행된다. 이 단계는 공을 탐지하고 align 하는 단계로 카메라1에 보이는 파란공이 없을 때는 시계방향 또는 반시계방향으로 회전하게 된다. 그 방향을 정하는 것은 변수 m에 의해 결정되는데 이는 case2에서 빨간공을 피할 때 왼쪽 또는 오른쪽으로 피하는데 이 때 m의 값을 바꿔주어 왼쪽으로 피하면 시계방향으로 회전하고 오른쪽으로 피하면 반시계방향으로 회전하게끔 만든다. 이로 인해 case2에서 공을 잃어버린 뒤 case1으로 다시 오더라도 맨처음 case1에서 align한 공을 향해 다시 align하게 되는 것이다.

카메라1에서 보이는 파란공이 개수가 0이 아닐 때는 가장 오른쪽에 있는 공으로 align을 한다. 정확히 각도를 0으로 맞출 수는 없기 때문에 theta_c_i라는 상수를 둬서 각도가 이 각도 안으로 들어오면 align 되었다고 판단한다.

모든 과정이 끝나면 phase를 2로 설정하여 다음 loop가 돌 때 case2가 실행되도록 한다.

4.5.3.2.

MOVING A 코드 내

→ 경로에 빨간공이 있으면 옆으로 피하고, 없으면 moving b 코드 실행

```
theta_car1 = RAD2DEG(atan(car_half_width1/nearest_r_posi[2])); // (가장 가까운 빨간공의 z좌표)와 (웹캠중심과 차 왼쪽 끝의 거리)를 이용하여 각
theta_car2 = RAD2DEG(atan(car_half_width2/nearest_r_posi[2])); // (가장 가까운 빨간공의 z좌표)와 (웹캠중심과 차 오른쪽 끝의 거리)를 이용하여

if(abs(theta_car2 + theta_avoid)>abs(nearest_r_posi[3]) && nearest_r_posi[3]>0){ // 빨간공이 차체가 파란공까지 가는 직선경로 안에 들어와있고
    go_left(); // 왼쪽으로 차를 움직인다
    m = 1; // m을 1로 설정하여 만약 공을 잃어버렸을 경우 case1로 갔을 때 시계방향으로 회전하여 공을 찾게 한다
}
else if(abs(theta_car1 + theta_avoid)>abs(nearest_r_posi[3]) && nearest_r_posi[3]<0){ // 빨간공이 차체가 파란공까지 가는 직선경로 안에 들어와
    go_right(); // 오른쪽으로 차를 움직인다
    m = 0; // m을 0로 설정하여 만약 공을 잃어버렸을 경우 case1로 갔을 때 반시계방향으로 회전하여 공을 찾게 한다
}
else{ // 가장 가까운 빨간공이 가장 가까운 파란공보다 가까이 있지만 파란공까지 가는 직선경로 안에 빨간공이 없는 경우 및 빨간공을 옆으로 피한 경
    if(nearest_b_posi[4] > b_aa){ // 가장 가까운 파란공과의 거리가 b_aa보다 클 경우
        go_straight(); // 앞으로 전진한다
    }
    else if(nearest_b_posi[4] > b_a){ // 가장 가까운 파란공과의 거리가 b_aa보다 작고 b_a보다 큰 경우. 즉, 빨간 공 피했는데 바로 얼라인 하면 안
        if(nearest_b_posi[0] < 0 && abs(nearest_b_posi[3])>theta_c_i){ // 가장 가까운 파란공이 왼쪽에 있으면서 각도가 theta_c보다 클 때
            turn_ccw_go_r(); // 앞으로 전진하면서 반시계방향으로 조금 회전한다. 다만 turn_ccw_go 함수보다 회전하는 속도가 느려서 주로 앞으로
        }
        else if (nearest_b_posi[0] > 0 && abs(nearest_b_posi[3])>theta_c_i) { // 가장 가까운 파란공이 오른쪽에 있으면서 각도가 theta_c보다 클
            turn_cw_go_r(); // 앞으로 전진하면서 시계방향으로 조금 회전한다
        }
        else{ // 위의 두 경우가 아닐 때, 즉, 파란공이 얼라인 되어 있을 때
            go_straight(); // 앞으로 전진한다
        }
    }
    else if(nearest_b_posi[4] < b_a && nearest_b_posi[0]<0 && abs(nearest_b_posi[3])>theta_c){ // 가장 가까운 파란공과의 거리가 b_a보다 작고,
        turn_ccw_go(); // 앞으로 전진하면서 반시계방향으로 회전한다. 회전하는 각도가 turn_ccw_go_r보다 더 크다
    }
    else if(nearest_b_posi[4] < b_a && nearest_b_posi[0]>0 && abs(nearest_b_posi[3])>theta_c){ // 가장 가까운 파란공과의 거리가 b_a보다 작고,
        turn_cw_go(); // 앞으로 전진하면서 시계방향으로 회전한다
    }
    else if(nearest_b_posi[4] > r_c){ // 위의 경우가 모두 아니면서, 가장 가까운 파란공과의 거리가 r_c보다 클 때
        go_straight(); // 앞으로 전진한다
    }
    else{ // 위의 경우가 모두 아닐 때, 즉, 빨간공 다 피하고 얼라인 되어있는 상태로 파란공 앞 거리 r_c까지 왔을 때
        dup_go_straight(); // case3로 가게 되면 카메라2를 쓰게 되고 카메라2가 아래쪽에 달려 있어서 파란 공이 2개가 붙어있으면 1개의 큰 공으로
        phase = 3; // phase를 3으로 설정하여 case3로 넘어가게 한다
    }
```

Figure 17

여기가 MOVING A 코드 내의 앞부분이고 큰 else 안의 if~else 부분이 moving b 코드가 된다. 이 두 코드는 뒤에서도 반복되어 쓰이기에 여기서만 설명을 하겠다.

먼저, (가장 가까운 빨간공의 z좌표)와 (웹캠중심과 차 왼쪽 끝의 거리)를 이용하여 각도를 구한다. 즉 이 각도는 빨간공의 z좌표를 가지고 자 왼쪽 끝의 x좌표를 가지는 좌표의 각도이다. 웹캠이 차의 중심에 달려있지 않아 두 개의 각도를 구한다. 따라서 이 각도 안에 빨간공의 각도가 있다면 빨간공은 파란공으로 가는 경로에 존재하여 피해야 한다.

빨간공이 차 중심보다 오른쪽에 있으며 경로 안에 존재하면, 왼쪽으로 피해야만 조금만 움직여도 빨간공을 피할 수 있다. 따라서 이 두 경우를 나눠서 각각 왼쪽 오른쪽으로 피하고 변수 m을 조절하여 공을 잃어버려 다시 case1으로 가더라도 원래 align했던 공을 찾을 수 있게끔 만든다.

이제 공을 피하거나 원래 빨간공이 경로에 없었을 경우 실행되는 else 안의 moving b 코드를 살펴보자. 먼저 파란공과의 거리가 b_aa가 될 때까지 앞으로 전진한다. 이 코드로 인해 빨간공을 옆으로 회피하더라도 바로 align하는 것이 아니라 옆으로 피하고 앞으로 전진하여 빨간공을 지나치게 만들어 준다. 거리가 b_aa가 되면 align하면서 앞으로 회전하며 전진하고 b_a가 되면 회전하는 속도를 더 높여서 align하는 속도를 빠르게 만들어준다. 이 단계를 나눈 이유는 align 할 때 아직 빨간공을 지나치지 않았을 가능성이 있기에 빨간공과 차가 부딪히지 않게 처음에는 회전속도를 느리게 하고 나중에는 높여준 것이다. turn_ccw_go_r() 함수는 전진하면서 회전하는 속도가 느리고, turn_ccw_go() 함수는 전진하면서 회전하는 속도가 빠르다. 이렇게 align하면서 파란공에 다가가고 이제 파란공과의 거리가 r_c가 되면 dup_go_straight()함수를 실행한다. 이 함수는 카메라2가 바닥에서 별로 높지 않게 달려있어 공이 두개가 가까이 있게 되면 하나의 큰 공으로 식별하는 문제를 없애기 위해 실행하는 함수이다. 이 함수를 실행하고 난 뒤에는 phase를 3으로 바꿔 case3를 실행한다.

위의 moving b 코드는 MOVING A 함수 안에서도 총 3번 쓰인다. 즉, (가장 가까운 빨간공이 가장 가까운 파란공보다 가까이 있을 때)에는 위의 캡쳐된 코드처럼 빨간공을 피하고 moving b 코드가 실행되고 (가장 가까운 빨간공이 가장 가까운 파란공보다 멀리 있을 때)와 (카메라1에 빨간공이 안보일 때)는 moving b 코드만 실행된다.

그리고 MOVING A코드는 뒤에서 2번더 쓰이게 된다. 위에서 작성한 스케치를 참고하면 좋을 것 같다.

4.5.3.3.

case3 (Blade를 돌려서 실제로 공을 픽업하는 것이 목적)

→ pickup 코드 실행

```
case 3: // picking ball 단계이다. 즉, case 1,2를 통해 파란공 앞까지 왔고 이 단계에서는 밑에 달린 카메라2를 통해 공을 탐지한다
        if(b_ball_number2 == 0){ // 카메라2에서 파란공이 보이지 않을 때
            j = 1; // 밑의 open loop을 돌리기 위해 j값 초기화
            while(j<15){ // 일정시간 동안 차를 뒤로 가게 만든다. 즉, case1로 보내도 카메라1에서 파란공이 안보일 수가 있기에 그 전에 차를
                go_back(); // 차를 뒤로 가게 하는 함수 실행
                write(c_socket, data, sizeof(data)); // 랩뷰로 통신할 데이터를 write한다
                j++; // j=j+1
                ros::Duration(0.02).sleep(); //duration을 0.02로 설정하여 일정한 시간동안 while문이 돌아가게 만든다
            }
            phase = 1; // phase를 1로 설정하여 case1으로 보낸다
        }
        else if(abs(nearest_b_posi2[3]) > theta_c_ii && nearest_b_posi2[0] > 0){ // 카메라2에서 파란공과의 각도가 theta_c_ii보다 크고 파
            turn_ccw_so_slowly(); // 천천히 반시계방향으로 회전
        }
        else if(abs(nearest_b_posi2[3]) > theta_c_iii && nearest_b_posi2[0] < 0){ // 카메라2에서 파란공과의 각도가 theta_c_ii보다 크고 파
            turn_cw_so_slowly(); // 천천히 시계방향으로 회전
        }
        else if(nearest_b_posi2[4]>p_c){ // 위의 세 경우가 아닌 경우, 즉, 얼라인 되고 파란공과의 거리가 p_c보다 클 때
            go_straight_slowly(); // 앞으로 천천히 전진
}
        else if(nearest_b_posi2[4]<0.4){ // 갑자기 노이즈로 인해 파란공과의 거리가 0이 뜨는 경우가 있어서 그 경우 픽업이 실행되지 않게하
            go_straight_slowly(); // 앞으로 천천히 전진
        }
        else{ // 얼라인 된 상태에서 파란공과의 거리가 p_c가 되었을 때
                    int i =1; // 밑의 open loop을 실행하기 위해 i값을 1로 선언한다
            while(i<45){ // 일정시간 동안 blade를 돌려서 공을 픽업하게 한다. 여기서 i<45는 실험으로 정한 값이다
                        pick_up(); // blade를 돌리는 픽업함수 실행
                        write(c_socket, data, sizeof(data)); // 랩뷰로 통신할 데이터를 write한다
                        i++; // i=i+1
                        ros::Duration(0.02).sleep(); // duration을 0.02로 설정하여 일정한 시간동안 while문이 돌아가게 만든다
            }
            phase = 1; // phase를 1로 초기화 한다
            m = 0; // m을 0으로 초기화 한다
            pick_ball_num++; // 픽업한 공의 개수를 하나 추가한다.
            dataInit(); // data 값을 초기화하여 랩뷰로 통신하는 data를 초기값으로 돌려놓는다
```

Figure 18

picking ball 단계로 들어오면 이제는 밑에 달린 카메라2로 목표 공을 탐지한다. 만약 카메라2에서 보이는 파란공의 개수가 0이면 open loop으로 일정시간동안 모델을 후진시킨 다음 case1으로 돌려보낸다. 후진시키는 이유는 바로 case1으로 보냈을 시 카메라1에서도 이 파란공을 탐지 못할 가능성이 있기 때문이다.

파란공이 탐지된다면 pickup을 위해 정밀하게 align을 한다. 허용되는 각도 theta_c_ii, theta_c_iii를 실험을 통해 정하고 각도가 안 맞을 시 시계 또는 반시계 방향으로 회전시킨다. align이 되었다면 이제 공을 향해 천천히 전진한다. 만약 가다가 align이 어긋났을 시 다시 align을 하고 전진하게 된다. 이제 파란공과의 거리가 p_c가 되었다면 blade를 open loop으로 돌려 공을 픽업하게 된다. 또한 갑자기 공과의 거리가 0이 뜨는 노이즈가 가끔 있기에 노이즈를 제거하기 위해 공과의 거리가 어느정도 이하이면 그냥 노이즈로 생각하고 계속 앞으로 가게끔 만들었다.

Pickup이 끝나면 픽업한 공의 개수를 하나 추가하고 phase 등을 초기화하여 case1부터 다시 시작하게끔 만든다.

여기까지의 과정이 끝나면 if (픽업한 공의 개수가 0일 때)가 완료되고 이제 else if (픽업한 공의 개수가 3개 미만일 때) (즉, 1~2개 픽업 했을 때)가 실행되게 된다. else if 안의 코드는 if (픽업한 공의 개수가 0일 때) 안의 case2에서 else if (가장 오른쪽 파란공까지 가는 경로 안에 다른 파란공 있을 때) 코드를 빼고 rightest_b_posi를 모두 nearest_b_posi로 바꾸면 된다. 따라서 코드가 일부 변형된 것을 제외하고는 반복되기에 cpp파일에서는 코드가 변형된 것만 설명했고 주석을 반복하여 달지 않았다. 또한 여기서도 설명을 생략하겠다. 따라서 다음 설명은 release이다.

4.5.3.4.

else (공을 3개 모두 픽업 했을 때) (즉, release모드 실행)

case1 (초록공 2개 detect 하고 g_dis(실험으로 구한거리)까지 간 다음 x,z align)

```
case 1: // case1 실행. 이 단계에서는 초록공 2개를 탐지하고 g_dis 거리까지 간 뒤에 z와 x를 얼라인 하여 틀
  if(g_ball_number < 2){ // 카메라1에 보이는 초록공의 개수가 2개 미만일 경우
          if (p == 0){ // p의 값이 0일 때 실행. p의 초기값은 0으로 설정되어 있다. 즉, 처음으로 case1에
             turn_ccw_fast(); // 빠르게 반시계방향으로 회전
          }
          else if (p == 1){ // p의 값이 1일 때 실행. 즉, 밑의 else에서 얼라인하다가 카메라에 공이 2개
             turn_cw_go(); // 앞으로 전진하면서 시계방향으로 회전
          }
          else{ // p의 값이 2일 때 실행. 즉, 밑의 else에서 얼라인하다가 카메라에 공이 2개가 안보이면
             turn_ccw_go(); // 앞으로 전진하면서 반시계방향으로 회전
          }

  }

  else{ // 카메라1에 보이는 초록공의 개수가 2개일 경우
          if(g_mid[1] > g_dis){ // 초록공 2개 사이의 중심과의 거리가 g_dis 이상일 경우
             go_straight(); // 앞으로 전진
          }
          else if(g_mid[2] > z_diff){ // 오른쪽 초록공 z좌표에서 왼쪽 초록공 z좌표를 뺀 값이 양수일 경
             turn_ccw_go(); // 앞으로 전진하면서 반시계방향으로 회전
             p = 1; // p를 1로 설정. 따라서 반시계방향으로 회전하다가 공을 하나 일어버리면 전 if문으
          }
          else if(g_mid[2] < -z_diff){ // 오른쪽 초록공 z좌표에서 왼쪽 초록공 z좌표를 뺀 값이 음수일
             turn_cw_go(); // 앞으로 전진하면서 시계방향으로 회전
             p = 2; // p를 2로 설정. 따라서 시계방향으로 회전하다가 공을 하나 일어버리면 전 if문으로
          }
          else if(rightest_g_posi[0]> g_x){ // 가장오른쪽에 있는 초록공의 x좌표가 g_x보다 큰 경우. 즉
             turn_cw_go(); // 앞으로 전진하면서 시계방향으로 회전
             p = 2; // p를 2로 설정. 따라서 시계방향으로 회전하다가 공을 하나 일어버리면 전 if문으로
          }
          else if(rightest_g_posi[0]< -g_x){ // 가장오른쪽에 있는 초록공의 x좌표가 -g_x보다 작은 경우
             turn_ccw_go(); // 앞으로 전진하면서 반시계방향으로 회전
             p = 1; // p를 1로 설정. 따라서 반시계방향으로 회전하다가 공을 하나 일어버리면 전 if문으
          }
          else{ // 초록공과의 거리가 g_dis 이내에서 x,z가 얼라인 되었을 경우
             phase = 2; // phase를 2로 설정하여 case2를 실행하게 한다
             p = 0; // p의 값을 0으로 초기화한다
```

Figure 19

case1에서는 초록공 2개를 탐지하고 g_dis거리까지 간 뒤 z와 x를 align하여 수평을 맞추게 된다. p라는 변수를 이용하여 align하다가 초록공 둘 중에 하나라도 잃어버리게 되면 잃어버린 방향으로 다시 회전하여 공을 찾게끔 만든다. 만약 초록공과의 거리가 g_dis이내로 들어오고 align도 끝났다면 phase를 2로 설정하여 다음 loop 때 case2가 실행되도록 한다.

4.5.3.5.

case2 (align 된 상태로 g_c까지 전진)

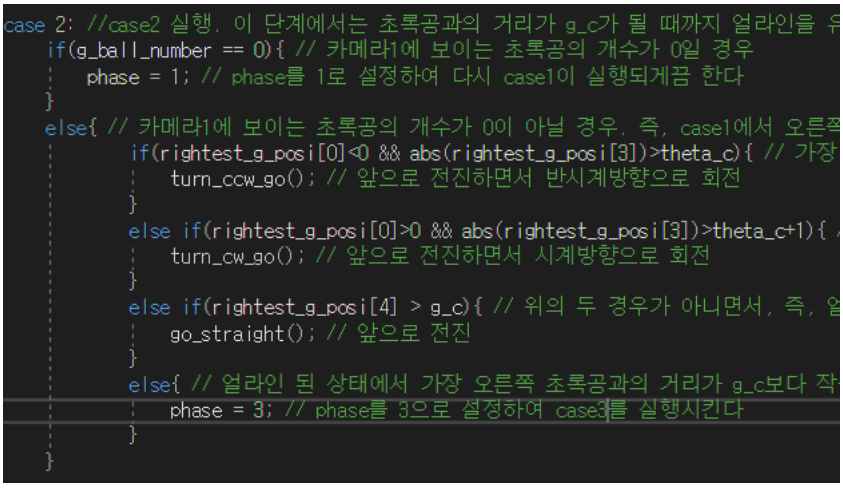


Figure 20

case2에서는 초록공과의 거리가 g_c가 될 때까지 얼라인을 유지하면서 앞으로 전진하게 한다. 여기서 우리는 오른쪽 초록공을 향해 전진하게 되는데 그 이유는, 모델의 release하는 부분이 왼쪽에 달렸기 때문에 오른쪽 초록공을 향해 모델의 중심이 다가가야 release가 바구니 안으로 될 수 있기 때문이다. g_c 거리 이내로 들어오면 case3로 보내게 된다.

4.5.3.6.

case3 (relase)

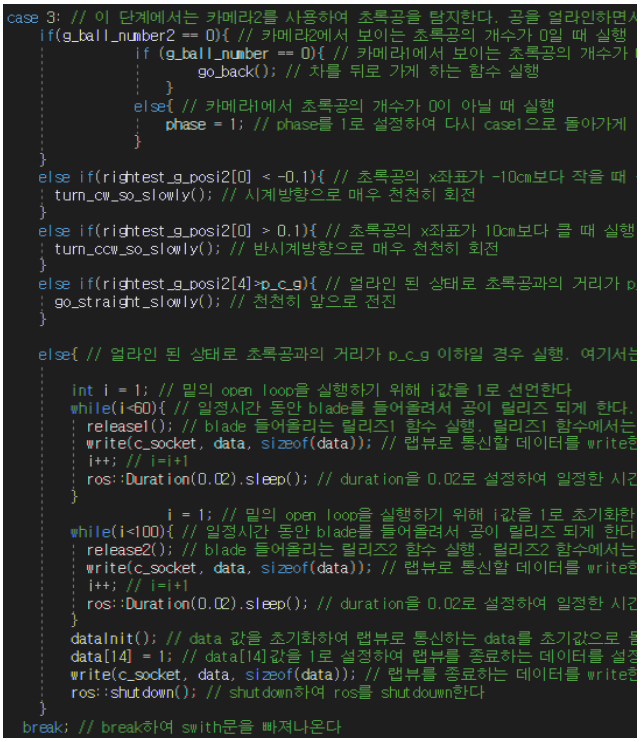


Figure 21

이 단계에서는 픽업하기 위해 카메라2로 파란공에 천천히 align하면서 다가갔던 것처럼 이제는 카메라2를 이용해 오른쪽 초록공을 향해 천천히 align 하면서 다가간다. 만약 초록공이 카메라2에서 안보이면 카메라1에서 초록공이 보일 때까지 후진하게 만들고 보이게 되면 case1로 다시 보내게 된다. 거리가 p_c_g 안으로 들어오면 open loop 2개를 사용하여 공을 release하게 된다. 2개 사용하는 이유는 처음에는 blade를 들면서 앞으로 천천히 전진하고, 마지막에는 천천히 후진하게끔 만들어 releas할 때 공이 바구니에 끼지 않도록 한 것이다. 이로 인해 안정성이 더 높아졌다.

4.6. 코드 요약

우리는 2.2.에서 설명한 알고리즘을 바탕으로 4.5.2.의 스케치 코드를 완성하였고 그 코드를 더 발전시켜서 실제로 구현가능한 코드를 작성하였다. 수많은 실험과 시행착오를 겪으면서 많은 에러를 제거했고 완성된 코드는 성공 가능성이 매우 높은 코드이다. 우리는 우리의 철학인 simple & compact를 하드웨어를 포함한 코드에 접목시켰고 최대한 간단하게 작성하려고 노력했다. 따라서 MOVING A, moving b, pickup 코드를 만들고 반복하여 사용되게끔 했다. 따라서 전체적으로 코드가 조금씩 변형되어 반복되어 쓰이는 구문이 많기에 코드가 정리되어 보이는 느낌이 든다.

5. 마무리

ROS에 대한 개념이 하나도 없는 상태에서 창시구 프로젝트를 시작하였고, 잘 모르기에 앞길이 막막했으나 교수님들과 조교님들의 도움으로 많은 난관을 헤쳐 나갈 수 있었다. 성공적인 결과를 얻을 수 있었고 모든 분들에게 감사드린다.

이 보고서를 보게 될 미래의 창시구 및 ROS 수강생분들에게 팁을 드리자면, segementation error는 허용되지 않은 메모리 영역에 접근하려 할 때 뜨는 에러인데 보통 ros에서는 배열 사용 중에 발생한다. 예를 들어 3의 크기의 배열을 사용하는데 4번째 배열을 사용하려 하면 segementation error가 뜨며 코드가 중단된다. 공이 하나 밖에 없어서 posi 크기가 1인데 조건문에서 posi에 저장된 2개의 공의 정보를 비교하는 등의 오류가 없는지 확인해야 한다.

또한 에러가 뜰 때 어디서 에러가 났는지 모르겠다면 cout을 실행되는 함수 등 안에 넣어서 확인해보고 터미널에서 마지막 cout이 어디서 되었는지 확인하면 에러가 난 지점을 확인할 수 있다.

스케쥴을 짤 때에는 먼저 실제로 코드를 실행시켜 볼 수 있을 만큼 하드웨어를 먼저 완성시켜야 한다. 하드웨어가 완성이 된다면 코드를 실제로 실행시킬 수 있기에 코드 발전 속도가 비약적으로 빨라진다. 다시 말하자면, 코드를 작성하여 바로바로 실행시켜 볼 수 없다면 그 코드는 손봐야 할 곳이 너무나도 많다. 항상 바로바로 실험하는 것이 코드를 빨리 완성하는데 도움이 될 것이다.

한학기 같이 고생해주신 교수님, 조교님들께 다시 한 번 감사드립니다.