2019 Spring ME492 Final Report

Group D – Team Kirby
조기태, 20110858
류상우, 20130223
한동훈, 20140619
엄용, 20150471
정태욱, 20150700
문희주, 20150866
김형준, 20160198

2019.06.16

Contents

Ⅰ. 개요

1. 과제에 대한 접근

1) 고찰

주어진 과제는 빨강색 공을 피해 3 개의 파란 공을 수집한 후 초록 공 또는 바구니를 이용하여 출발지점으로 돌아와 바구니에 공을 집어넣는 것을 목표로 한다. 각 공은 다른 공 또는 벽과 최소 50cm 의 간격을 두고 있으며, 그림 1 과 같이 출발지점으로부터 2.5m 이상 떨어진 곳에 배치된다.

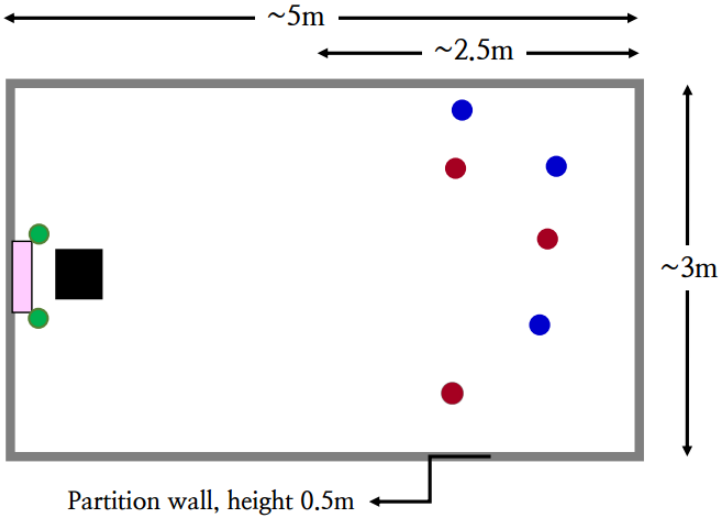

그림 1 Capstone Design Mission1 Track

공과 공 사이, 그리고 공과 벽 사이에 최소한의 거리 50cm 가 확보 되어있다는 사실은 빨간 공과 파란 공이 인접해 있을 때 구별하여 집을 필요가 없다는 사실 외에도, 빨간 공을 회피하는 알고리즘을 보다 간단히 만들 수 있는 단초가 된다. 로봇 차체를 일정 크기 이하로 줄인다면 공 사이를 빠져나가게 하는 것이 가능하다는 의미인 바, 벽과 충돌한다든가 공과 부딪혀 공의 배치가 복잡해지는 일을 고려하지 않아도 되는 것이다.

기본적으로 주어지는 ball_detection_node 에서는 공의 크기를 이용하여 공과 webcam 의 거리(z 값)를 계산해주는데, 실제 주어지는 모든 공의 크기가 일정하여 이러한 기능의 사용이 가능하다는 점 또한 중요하다. 다만 공의 크기가 일정한 경우 z 값과 더불어 y 값 또한 (노이즈가 없다면) 거리의 지표로서 사용이 가능하다. 우리는 거리 지표로써 y, z 값을 혼용하였는데, 그 이유는 후술한다.

바구니와 바구니에서 가장 가까운 공 사이의 거리가 2.5m 라는 것, 그리고 그 공이 벽에서 최소 50cm 떨어져 있다는 것은 마지막 공을 수집한 후 곧바로 바구니의 중심으로 향할 때 최악의 경우라도 약 35 도의 진입각을 확보할 수 있다는 것을 뜻한다. 이는 release step 에서 불필요한 과정을 줄여 주행시간을 단축시킬 수 있는 포인트가 된다.

2) 과제의 해결

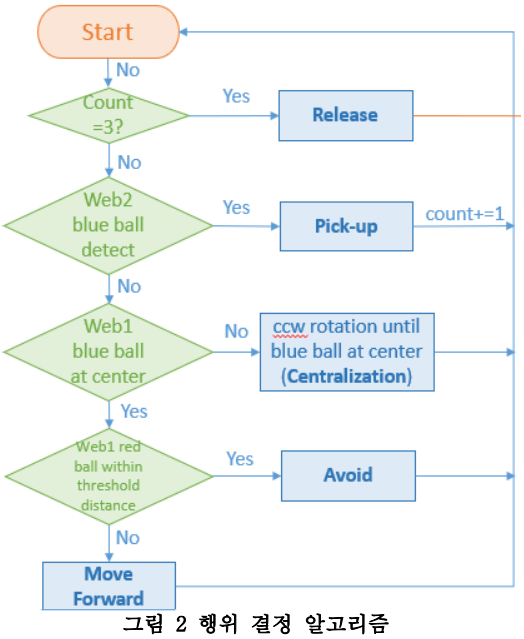

그림 2 행위 결정 알고리즘

공을 수집하여 과제를 해결하는 대략적인 알고리즘은 그림 2 와 같다.

파란 공이 보이지 않는 경우 Counter clockwise(이하 CCW)로 회전하며, webcam 에서 인식하는 복수개의 파란 공 중 '가장 오른쪽 공' 을 향해 움직인다. 움직이는 도중 빨간 공은 충돌하지 않는 범위 내에서 회피하며, webcam2 에 파란 공이 들어오는 경우 파란 공 수집을 우선적으로 실행한다. 모두 수집한 경우에는 회전하여 초록 공을 기준으로 바구니를 찾아 그 중심을 향해 접근하여 공을 release 한다.

2. 경로 결정 알고리즘

1) 폐루프 알고리즘

소위 말하는 개루프(open loop) 제어와 폐루프(closed loop) 제어는 과제를 수행함에 있어 공의 정보를 지속적으로 업데이트하여 로봇의 움직임에 매 순간 반영할 것인가의 문제로 귀결된다.

개루프 제어를 선택하는 경우, 즉 처음 한 번 또는 중간중간 공의 위치정보를 업데이트하여 로봇이 움직이는 와중에는 실시간 업데이트 하지 않는 방식은 코딩이 쉽다는 장점이 있다. 다만 이는 결정적으로 공 위치의 정확도가 높아야 하는 바 Vision processing 에 무게를 두는 동시에 하드웨어의 움직임 또한 정밀하여야 하며, 공의 위치를 업데이트하지 않고 있는 동안 돌발적으로 일어나는 상황에 대처하기 힘들다는 단점이 있다.

따라서 Team Kirby 는 폐루프 제어를 선택하였으며, 실제 모든 과정에 있어 필요한 공의 위치정보를 webcam 으로부터 업데이트 받아 로봇의 움직임을 판단하는데 사용하였다. 다만 폐루프 제어는 노이즈에 취약해지는 단점이 있는데, 이는 후술할 debouncing 등의 방법을 사용하여 해결하였다.

2) 상대 좌표계

상술한 과제를 수행함에 있어 상대 좌표계는 절대 좌표계에 비해 열등한 것으로 보이기 십상이다. 이는 우리가 로봇이 아닌 절대적 관찰자로서 트랙, 공과 로봇을 바라보고 있기 때문이다. 즉 우리가 바라보는 공의 위치와 로봇이 관찰하는 공의 위치는 필연적으로 달라질 수밖에 없으며, 제한된 시야각과 부정확한 공의 위치정보를 바탕으로 절대좌표계를 구성하여 path planning 을 하는 것은 과도한 노력을 필요로 하게 된다.

실제 우리는 Rplidar 나 방향 센서 등을 사용하여 절대좌표를 실현하는 방안 또한 충분히 고려하였다. 다만 의논한 결과 주어진 트랙이 비교적 간단한 형태로 되어 있는 점에 비추어 보아

상대좌표를 사용하는 것과 절대좌표를 이용하는 것이 큰 차이가 없다고 보았으며, 절대 좌표를 실현함에 있어 추가적으로 필요한 장비나 노력이 그러한 차이에 비해 다소 크다고 판단하였는 바 상대좌표를 이용하여 경로결정을 하게 되었다.

3. Vision

1) Camera positioning

Kirby 의 크기를 줄이기 위해 공의 흡입부를 차체의 아래에 둘 수밖에 없었고, 따라서 단일 webcam 만으로는 파이프의 바로 앞부터 전방 4.5m 까지의 시야를 확보하는 것이 불가능했다. 따라서 우리는 그림 2 와 같이 두 개의 카메라를 이용하여 전방으로 넓은 시야를 확보하고자 하였다.

webcam1 은 보다 넓고 먼 시야를 확보하며, 공이 멀리 있는 경우 처리해야할 일을 webcam1 의 시야를 통해 해결한다. 따라서 빨간 공을 회피하는 기작이나 파란 공, 초록 공을 대략적으로 추적하는 기능을 webcam1 에서 수행하게 된다.

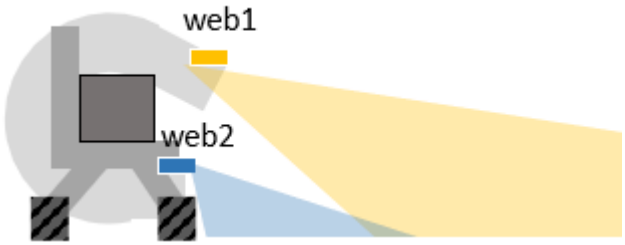

그림 3 Webcam positioning

webcam1 이 목표 되는 공을 정밀하게 추적하지 않는 이유는 목표 된 공을 webcam2 의 시야에 들어오도록 하면 되기 때문이다. 즉, webcam2 에 목표 된 공이 들어오는 경우 pick up 등을 위한 정밀한 추적을 시작하며, 실제 결정적인 과정들은 이 때 이루어지기 때문에 webcam2 에서 수집되는 공의 정보가 매우 중요해진다. 따라서 노이즈를 줄이고 안정장치를 추가하는 것이 필요한데, 이를 위한 Vision processing 과 Debouncing process 에 대하여는 후술하도록 한다.

2) Vision processing

상술한 Vision processing 의 중요성과 더불어 기본적으로 '창의적 시스템 구현' 의 과제는 공을 인식한 후 해당 정보를 기반으로 움직이는 것을 요체로 하기 때문에, 정확하게 공을 인식하고 그 위치를 파악하는 것은 프로젝트의 근간이 되는 작업이다. 실제 webcam 으로부터 공의 정보를 전달받아 확인해보면 수집되는 정보들이 비교적 부정한 것을 확인할 수 있다. 우리는 크게 세가지 문제를 직면했는데, 1) 공이 멀리 있는 경우 depth 정보의 부정확함, 2) 그에 따른 x, y 좌표의 부정확함 그리고 3) 공이 가까이 있는 경우 '공 안의 공' 이 인식되는 문제였다.

1) 공의 depth 정보가 부정확하다는 것은 다만 페루프, 상대 좌표를 이용한 제어를 선택한 우리로서 큰 문제는 되지 않았다. Kirby 는 상대좌표를 이용하기 때문에 절대적인 거리는 필요한 경우가 없었기 때문이었다(즉 상대적 거리의 측정은 비교적 정확하였으며, 다만 후에 releasing mode 에서 문제가 된 바 있으나 해당 문제에 대한 논의는 본문에서 다루도록 한다). 다만 이 문제는 2)에서 다룰 x, y 좌표 보정과 관련하여 문제가 된다.

2) 기본적으로 주어진 Ball detection node 에 의해 계산된 공의 x, y 좌표는 z 좌표를 기반으로 보정된 수치이다. 이는 물론 절대좌표계를 사용하는 경우 공의 위치를 파악하는 데에 보다 용이하겠으나, 그림 4 과 같이 공과 webcam 의 거리가 멀수록 z 좌표가 부정확해지는 경우 수집된 x, y 좌표 정보를 사용하기 어려워지게 된다.

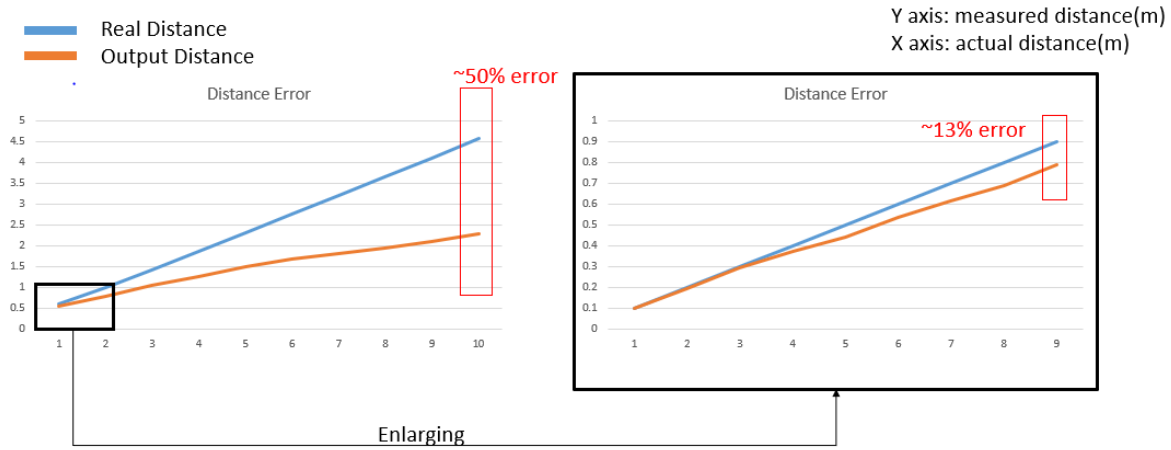

그림 4 거리에 따른 공의 z좌표 오차

더불어 우리는 Kirby 가 공을 추적, 회피하는 알고리즘에 있어 방향 전환 판단을 공의 x 좌표가 webcam 의 중심으로부터 얼마나 떨어져 있는지에 따라 결정하도록 코딩을 한 상태였다. 따라서 ball detection node 에서 공의 x, y 좌표를 화면상 pixel 값 기준으로 받도록 코드를 수정하였다. 즉, 이하 사용되는 공의 x, y 좌표는 거리에 따라 보정된 x, y 좌표가 아닌 webcam 상에서 pixel 값을 기준으로 결정되는 x, y 좌표값이다.

3) 공이 webcam 에 너무 가까이 위치해 있는 경우(통상적으로 그 거리가 10cm 이하 인 경우) 그림 5 과 같이 '공 안의 공' 이 인식되며 실제 공의 윤곽도 부정확하게 잡는 문제도 종종 발생한다. Kirby 의 경우 webcam2 가 suction pipe 상단에 위치하여 공과 짧은 거리를 유지하며 정밀 제어를 하는 것을 목표로 하였는 바, 이러한 문제를 직면하는 것은 불가피한 일이었다.

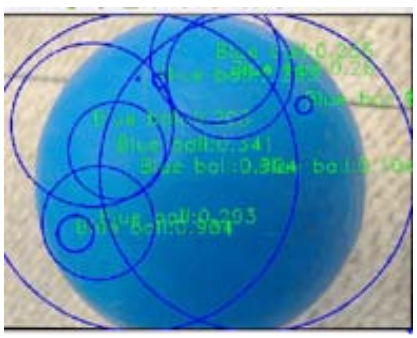

그림 6 Before vision processing

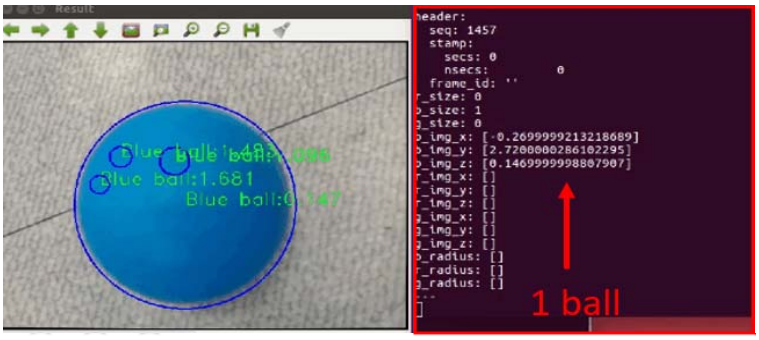

그림 5 After vision processing

따라서 우리는 ball detection node 외에도 그림 7 과 같이 Vision processing 을 위한 vision modifying node 를 추가하여 이러한 문제를 해결하였으며, 이를 통해 하나의 공을 복수의 공으로 인식하는 등의 부가적인 문제들도 함께 해결할 수 있었다.

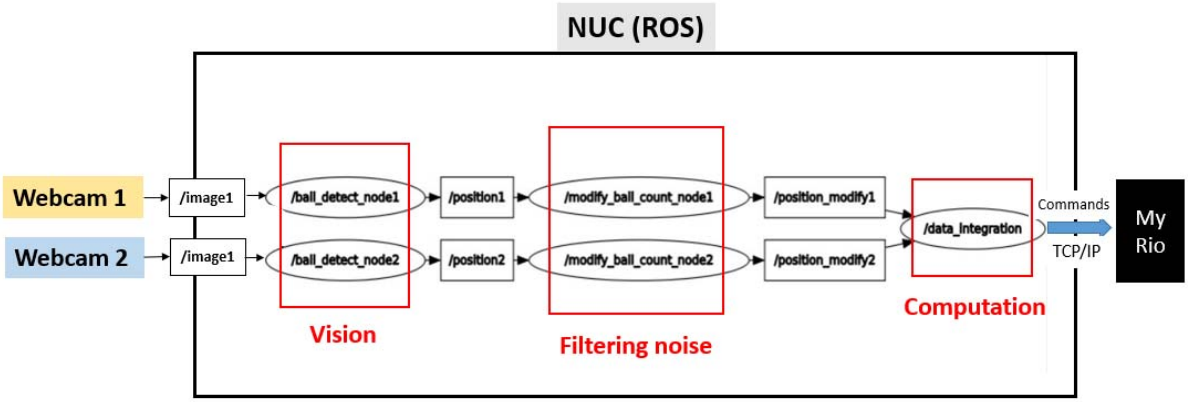

그림 7 System Integration Overview

Ⅱ. 변수 지정

1. webcam1

```
// webcam1 global variables
int web1_blue_number=0;//numb
int web1_red_number=0;//numbe
float web1_blue_X_array[20];/
float web1_blue_X = -100;//x
float web1_blue_Z_array[20];/
float web1_blue_Z = -100;//z
float web1_red_X_array[20];//
float web1_red_X = -100;//x p
float web1_red_Y_array[20];//
float web1_red_Y = -100;//y p
float web1_red_Z_array[20];//
float web1_red_Z = 100;//z po
int web1_green_number=0;//num
float web1_green_X_array[20];
float web1_green_X=-100;//x p
float web1_green_Y_array[20];
float web1_green_Y=100;//y po
float web1_green_Z_array[20];
float web1_green_Z=-100;//z p
float web1_green_X_min;//x va
float web1_green_X_max;//x va
float web1_green_X_closest;//
float web1_green_Y_closest;//
float web1_green_Z_closest;//
float web1_green_X_target;//x
float web1_green_X_average;//
float web1_green_Y_left;//y v
float web1_green_Y_right;//y
float web1_center=0; // x pos
```

그림 8 webcam1의 변수

webcam1에서 인식되는 빨강, 파랑, 초록 공의 개수와 각 x, y, z 좌표 이름을 설정한다. 같은 색의 공이 여러 개 인식되는 경우 모두 받기 위해 먼저 array로 정보를 받은 후, 필요한 공의 정보만을 web1_blue_X, web1_blue_Z 등에 저장하게 된다. 초록 공 정보는 release mode에서 필요하며, 초록 공의 위치 정보 외에도 바구니를 찾아가는 과정에서 필요한 값을 별도로 변수로 지정하여 계산, 저장하게 된다. Web1_green_X,Y,Z_closest는 가장 가까이 이는 초록 공의 x, y, z 좌표 값을 의미하며, web1_green_X_average는 두 초록 공의 x 좌표 평균값, web1_green_X_target은 특정 단계에서 판단하여 Kirby가 목표로 정한 초록 공의 x 좌표를 의미한다.

Web1_green_X_min,max는 각각 인식하는 두 개의 초록 공 중 왼쪽, 오른쪽에 위치한 공의 x 좌표를 뜻하며, 이와 유사하게 web1_green_Y_left,right는 두 개의 초록 공 중 각 왼쪽, 오른쪽에 위치한 공의 Y 좌표를 의미한다.

Web1_center는 webcam의 방향을 정확히 로봇의 직진 방향과 일치시키기 힘든 하드웨어의 특성상 중심 조정을 위해 설정한 변수이다.

각 공의 위치정보를 글로벌 변수 지정 단계에서 초기화 한 것은 실제 integration node를 실행시킬 때 발생하는 문제에 기인한다. 그림 8 처럼 변수의 초기값을 설정하지 않고 Integration node를 처음 실행하는 경우, 각 webcam에서 정보를 채 받아오지 못하고 main 함수를 실행하게 되어 랜덤하게 설정된 공의 위치정보를 바탕으로 짧은 시간동안 명령을 내리게 된다. 따라서 예상치 못한 초기 움직임을

방지하고 공이 카메라에 인식되지 않는 경우 의도한 대로 명령을 내리게 하기 위해 위와 같이 초기값을 설정한다.

2. webcam2

```
// webcam2 global variables
int web2_red_number=0; //numbe
float web2_red_X_array[20];//x
float web2_red_X = -100;//x po
int web2_green_number = 0;//nu
int web2_blue_number=0;//numbe
float web2_blue_X_array[20];//
float web2_green_Y_array[20];/
float web2_blue_X = -100;//x p
float web2_green_X = -100;//x
float web2_green_Y = -100;//y
float web2_green_Z = 100;//z p
float web2_green_X_max;//x pos
float web2_green_X_min;//x pos
float web2_green_X_array[20];/
float web2_green_Z_array[20];/
float web2_center=0;//x positi
```

그림 9 webcam2의 변수

기본적으로 webcam2 에서 수집하고 설정하는 변수는 대부분 webcam1 의 그것과 유사하다. 다만 webcam2 는 초록 공과 파란 공의 정밀 접근을 주로 담당하며, 빨간 공 회피는 만일의 경우 이루어지는 것으로 후술하는 바와 같이 빨간 공의 x 좌표 위치정보만 수집하여 회피하게 된다.

또한, webcam1 에서는 release mode 에서 차체의 각도를 조절하기 위해 초록 공의 위치정보를 가공한 결과값들이 필요하였다. 다만 webcam2 에 초록 공이 들어온 이후로는 초록 공에 근접하여 바구니에 공을 집어넣는 과정만 필요한 바, webcam1 에서와 같은 부가적인 변수지정은 필요치 않다.

3. Data Initiation - Command format

Integration node 에서 공의 위치정보를 받아와 이를 바탕으로 로봇에서 어떤 명령을 내릴지 결정한 후, 해당 명령을 내리는 형식은 실제 각 조마다 상이했다. 우리는 xbox 컨트롤러로 로봇을 움직이는 방식을 그대로 차용하여 명령을 내리도록 하였다.

```
data[0] = 0; //lx*data[3]; data[4] = 0; //rx*data[7];
data[1] = 0; //ly*data[3]; data[5] = 0; //ry*data[7];
data[20] = 0; //GamepadButtonDown(_dev, BUTTON_LEFT_SHOULDER);//servo m
data[21] = 0; //GamepadButtonDown(_dev, BUTTON_RIGHT_SHOULDER);//suctio
```

그림 10 Xbox 컨트롤러 버튼의 데이터 값

Xbox 컨트롤러에는 두 개 스틱의 상하, 좌우 위치를 나타내는 데이터 값을 포함하여 각 버튼마다 데이터 값이 배정되어 있어 총 23 개의 데이터 값을 이용할 수 있다. 이 중 우리는 수평이동 명령을 내리기 위해 좌측의 스틱(x 방향: data[0], y 방향: data[1])을, 회전 명령을 내리기 위해 우측 스틱을 이용하였으며(data[4]), suction motor 와 servo motor 의 가동을 위해 좌측과 우측의 shoulder 버튼을 사용하였다(data[20], data[21]). 이때 각 스틱의 데이터 값은 float 으로 -1 에서 1 까지 해당 값에 비례하여 속도가 변화하게 만들었으며, 이외의 버튼은 0 또는 1 의 값을 배정하여 Boolean 형식으로 명령을 내리게 하였다.

Ⅲ. Callback

1. 개요

그림 6 에서 보았듯이, ball_detection_node 에서 공의 정보를 받아 modify_ball_count_node에서 노이즈를 제거한 후 '/position_modify' 라는 토픽 이름으로 가공된 공의 위치 정보를 publish하게 된다. Integration node에서는 이렇게 정제된 공의 위치 정보를 바탕으로 상황 판단을 하는데, 이 때 여러 공의 정보 중 필요한 정보만을 추출하는 과정을 callback 함수에서 거치게 된다. 즉 publish 되는 공의 정보를 받아 두고 있다가, main 함수에서 spinOnce()가 실행되는 경우 subscribe된 정보를 callback 함수에서 새로 업데이트하여 이를 바탕으로 제어가 이루어지는 것이다.

2. Camera1_callback

1) 초기값 설정

```
web1_blue_X = -100;
web1_blue_Z = -100;
web1_red_X = -100;
web1_red_Y = -100;
web1_red_Z = 100;
web1_green_X = -100;
web1_green_Y = -100;
web1_green_Z = 100;
```

그림 11 초기값 설정

callback 함수에서 초기값을 설정한 것은 글로벌 변수 설정 단계에서 초기값 설정한 것과는 그 이유를 달리한다. 글로벌 변수 설정 단계에서 초기값을 설정한 것은 callback 함수가 돌지 않았을 때 main 함수가 초기값을 기준으로 돌기 때문이라면, callback 함수 내에서 새로이 변수를 초기화하는 이유는 매 callback 함수마다 필요한 공의 정보를 추출해서 가지고 오기 위함이다. 예를 들어 빨간 공의 경우 항상 가장 가까운 빨간 공의 정보만을 추출하여 web1_red_X,Y,Z에 저장하는데, 이를 위해 각 빨간 공의 z좌표를 비교하여야 한다. 매 순간 빨간 공이 멀어질 수도, 가까워질 수도 있기 때문에 새로운 공의 좌표를 업데이트하기 위해서는 가상의 '매우 멀리 있는 빨간 공' 을 설정하여 실제 빨간 공과 비교하는 과정이 필요하기 때문에 web1_red_Z의 초기값을 100으로 설정하였다.

web1_blue_X의 초기값 또한 '가장 오른쪽에 있는 파란 공' 을 추출해내고 파란 공이 없는 경우 CCW로 회전시키기 위해 -100으로 설정하였으며, 이외 web1_green_X,Z의 초기값 또한 상술한 바와 마찬가지 이유로 초기값을 설정하였다.

2) 공 개수 지정

각 색깔의 공 개수는 후술하는 바와 같이 callback 함수 내에서 필요한 공을 찾는 과정에 있어 필요하다. 그림 12 에서 보듯이 포인터를 이용하여 각 토픽으로부터 공의 개수 정보를 받아온다.

```
web1_blue_number = position_modify1->b_size;
web1_green_number= position_modify1->g_size;
web1_red_number = position_modify1->r_size;
```

그림 12 공의 개수 정보

3) 파란 공 정보

callback함수를 실행할 때마다 파란 공의 개수만큼 for문을 실행, 디폴트 값 '-100' 을 포함한 실시간 파란 공들의 x축 좌표를 비교하여 가장 우측에 있는(가장 x축 좌표 값이 큰) 파란 공 정보를 받아온다.

```
for(int i = 0; i < web1_blue_number; i++)
{
    web1_blue_X_array[i] = position_modify1->b_img_x[i];
    web1_blue_Z_array[i] = position_modify1->b_img_z[i];
    if(web1_blue_X < position_modify1->b_img_x[i]){
      web1_blue_X = position_modify1->b_img_x[i];
      web1_blue_Z = position_modify1->b_img_z[i];
```

그림 13 파란 공 정보 업데이트

4) 빨간 공 정보

callback 함수를 실행할 때마다 빨간 공의 개수만큼 for 문을 실행, 디폴트 값 '-100'을 포함한 실시간 빨간 공들의 y 축 좌표를 비교하여 가장 가까이 있는(가장 y 축 좌표 값이 큰) 빨간 공 정보를 받아온다.

```
for(int i = 0; i < web1_red_number; i++)
{
    web1_red_X_array[i] = position_modify1->r_img_x[i];
    web1_red_Z_array[i] = position_modify1->r_img_z[i];
    web1_red_Y_array[i] = position_modify1->r_img_y[i];
    if(web1_red_Y < position_modify1->r_img_y[i]){
      web1_red_X = position_modify1->r_img_x[i];
      web1_red_Z = position_modify1->r_img_z[i];
      web1_red_Y = position_modify1->r_img_y[i];
```

그림 14 빨간 공 정보 업데이트

가장 가까이 있는 빨간 공의 정보를 z 값이 아닌 y 값을 기준으로 한 것은 바구니의 색이 빨간색에 가까웠기 때문이었다. 카메라는 공의 크기를 기준으로 z 값을 판단하는 바, 바구니는 공보다 다소 커 멀리 있는 경우에도 '빨간 공'이 가까이 있는 것으로 인식하는 문제를 해결하기 위함이었다.

5) 초록 공 정보

초록 공의 경우 두 공의 위치 정보를 이용하여 계산해야 하는 값이 몇 개 있어 그림 15 와 같이 일단 array 를 통해 각 공의 정보를 저장한다.

```
for(int i = 0; i < web1_green_number; i++)
{
    web1_green_X_array[i] = position_modify1->g_img_x[i];
    web1_green_Y_array[i] = position_modify1->g_img_y[i];
    web1_green_Z_array[i] = position_modify1->g_img_z[i];
}
```

그림 15 초록 공 array 정보 업데이트

초록 공 또한 가장 가까운 공의 x,y,z 좌표를 업데이트 하는데, 그림 15 에서 보듯이 x, z 값과 y 값을 받을 때 판단하는 기준이 각각 z, y 값으로 서로 다른 것을 알 수 있다. 이는 초록 공의 y 값은 로봇과 초록 공 사이의 거리를 판단하여 해당 거리별로 움직이는 알고리즘을 바꾸기 위한 것임에 반해, z 축은 두 초록 공의 상대적인 거리를 비교하여 '더 가까운' 초록 공을 찾아내기 위한 것에 비롯하였다.

```
for(int i=1; i<web1_green_number; i++){
    if(web1_green_Y_closest < web1_green_Y_array[i]){
      web1_green_Y_closest = web1_green_Y_array[i];
    }
for(int i=1; i<web1_green_number; i++){
    if(web1_green_Z_closest > web1_green_Z_array[i]){
      web1_green_Z_closest = web1_green_Z_array[i];
      web1_green_X_closest = web1_green_X_array[i];
```

그림 16 가까이 있는 초록 공의 x, y, z값 업데이트

즉 반복적인 실험 결과 'Kirby 와 공' 의 거리를 판단하는 데에는 y 값을 사용하는 것이 비교적 안정적이나, 초록 공이 일정 거리 이상에 있는 경우 두 초록 공의 상대적인 거리를 비교하는 데에는 z 값을 비교하는 것이 더 안정적이라는 결론에 기인한 것이었다.

Release mode 에서 쫓아가고 있는 공의 좌우를 판단하기 위해 x 축 기준으로 좌측(web1_green_X_min)과 우측(web1_green_X_max)의 초록 공 좌표를 받게 된다. 다만 초록 공이 시야에 한 개만 존재하는 경우 그림 17 과 같이 해당 초록 공의 x 좌표를 X_min, max 에 모두 입력한다.

```
web1_green_X_min = web1_green_X_array[0];
web1_green_X_max = web1_green_X_array[0];
```

그림 17 초록 공이 1개인 경우

초록 공이 시야에 2 개 존재하는 경우, 그림 17 과 같이 양 공의 x 좌표를 서로 비교하여 x 값이 더 큰 공의 x 좌표를 web1_green_X_max 에, x 값이 더 작은 공의 x 좌표를 web1_green_X_min 에 저장한다.

```
for(int i=1; i<web1_green_number; i++){
    if(web1_green_X_min > web1_green_X_array[i]){
      web1_green_X_min = web1_green_X_array[i];
      }
    if(web1_green_X_max < web1_green_X_array[i]){
      web1_green_X_max = web1_green_X_array[i];
      }
}
```

그림 18 web1_green_X_min, max 업데이트

Release mode 에서 중요한 역할을 하는 변수 중 하나는 'leftright' 이다. 차후 상세히 기술하겠으나 간략히 설명하자면, release 함수에서 가장 가까이 있는 초록 공이 좌측 공인지 우측 공인지를 판단하여 좌측 공인 경우 leftright 에 0 을, 우측 공인 경우 leftright 에 1 의 값을 저장한다. 따라서 그림 18 에서 보이듯이 leftright 값에 따라 쫓아가게 되는 초록

공(web1_green_X_target)이 변하며, 결국 가장 가까이 있는 초록 공을 목표 초록 공으로 지정하게 되는 알고리즘이다.

```
if(leftright == 0){
    web1_green_X_target = web1_green_X_min;
}
else if(leftright == 1){
    web1_green_X_target = web1_green_X_max;
}
```

그림 19 목표 초록 공의 x좌표 설정

3. Camera2_callback

1) 초기값 설정

```
web2_red_X = -100;
web2_blue_X = -100;
web2_green_Z = 100;
web2_green_Y = -100;
```

callback1 에서와 마찬가지 방식으로 빨간 공과 파란 공의 x 좌표 기본값은 '매우 왼쪽'에, 초록 공의 y, z 좌표 기본값은 '매우 멀리' 설정하였다.

그림 20 초기값 설정

2) 파란, 빨간 공 정보

Callback1 에서와 마찬가지로 가장 오른쪽에 있는 파란 공, 가장 가까이 있는 빨간 공의 정보를 받아온다.

3) 초록 공 정보

초록 공 정보 또한 callback1 에서와 마찬가지 방식으로 가져오며, callback2 에서는 가장 가까운 초록 공과 좌, 우측의 초록 공의 좌표 정보를 받아온다.

대부분의 구조가 callback1 과 동일하나, 가장 가까운 초록 공의 정보를 받는 과정은 조건문에 있어 상이함을 보인다.

```
if(web2_green_Z > position_modify2->g_img_z[i] && position_modify2->g_img_z[i] != 0){
```

그림 21 web2_green_Z 판단 조건문

webcam2 내에서 보이는 가장 가까운 초록 공을 z 값으로 판단한 것은 실제 초록 공이 webcam2 에 매우 근접하는 경우 작은 공들이 인식되는 노이즈가 발생하였으며, 이 경우 y 축을 기준으로 가장 가까운 초록 공을 찾는 경우 노이즈가 영향을 끼칠 수 있어 작은 크기의 노이즈에 영향을 받지 않는 z 값을 기준으로 가까이 있는 초록 공의 위치정보를 업데이트 하였다.

또한 webcam2 에 초록 공이 들어오는 순간 이따금 중심(x=0)에 z=0 인 초록 공이 인식되는 노이즈가 발생하였으며, 이러한 노이즈를 제거하기 위해 z=0 인 초록 공은 업데이트 하지 않도록 하는 조건문을 추가하였다(z=0 이라는 것은 webcam 과 공의 거리가 0 이라는 뜻으로, 실제 일어날 수 없는 일이다).

Ⅳ. Moving functions

1. Move_forward()

```
if(data[1] < v){
data[0] = 0;
data[1] = data[1]+acc;
data[4] = 0;
data[5] = 0;
}
else if(data[1] > v){
data[  0] = 0;
data[1] = data[1]-acc;
data[4] = 0;
data[5] = 0;
}
```

그림 22. 직진 속도

함수에 입력한 속도대로 직진신호를 넣어준다. 하지만 입력한 속도가 현재 차체의 속도보다 훨씬 크거나 작을 경우 차체 진동이 커지고 wheel slip 등이 발생할 수 있다. 특히 정지시에 속도가 급하게 줄어들어 slip이 발생하면 정확한 위치에 멈추지 못하므로 문제가 발생한다. 따라서 가속 및 감속 알고리즘을 추가하여 차체 진동을 줄이고 wheel slip을 줄이도록 구성했다. 만약 입력한 속도가 현재 속도(data[1])보다 작다면 move_forward function이 실행될 때 마다 속도에 acc라는 상수를 더하여 입력한 속도로 가속시킨다. 입력한 속도가 현재 속도보다 크다면 반대로 acc를 빼서 입력한 속도로 감속시킨다.

```
if(web1_red_Y > 2.52 && collection < 3){
  if(data[1] < 0.5){
   data[0] = 0;
   data[1] = data[1]+acc;
   data[4] = 0;
   data[5] = 0;
  }
  else if(data[1] > 0.5){
   data[0] = 0;
   data[1] = data[1]-acc;
   data[4] = 0;
   data[5] = 0;
  }
}
```

그림 23. 빨간 공 접근 속도

직진 시 빨간 공이 가까이 다가올 경우 회피를 하게 된다. 이 때 직진 속도가 빠르면 감속을 하더라도 빨간 공에 지나치게 가까이 접근하여 회피에 문제가 생겼다. 따라서 빨간 공이 어느정도 가까이 있을 경우 직진 속도를 낮춰서 회피의 정확성을 높였다.

```
suction_check();
write(c_socket, data, sizeof(data));
```

그림 24 suction check

Suction_check function을 통해 suction motor on/off를 결정하고 write function을 통해 data array 값을 myRio로 송신한다.

2. Turn_CW(), Turn_CCW()

```
data[0] = 0;
data[1] = 0;
data[4] = w;
data[5] = 0;
```

그림 25. 회전 신호

함수에 입력한 속도대로 회전 신호를 넣어준다. 나머지는 move_forward function과 같이 suction_check(), write function으로 구성된다. Turn_CCW()는 회전 신호의 부호만 다르고 Turn_CW()와 동일하다.

3. Move_left(), move_right()

```
if(data[0]>-v){
data[0] = data[0]-acc;
data[1] = 0;
data[4] = 0;
data[5] = 0;
 }
```

그림 26. 좌우 이동 속도

함수에 입력한 속도대로 좌우 이동 신호를 넣어준다. Move_forward function과 마찬가지로 감속 알고리즘이 추가되고 suction_check(), write function으로 구성된다. Move_left function의 경우도 부호만 다르고 move_right function과 동일하다.

Ⅴ. Main

1. 알고리즘 및 구조

앞서 언급한 바와 같이 webcam1에서는 파란 공의 대략적인 추적과 빨간 공의 회피를 하며, 그 와중 webcam2에 파란 공이 들어오는 경우 pick up을 시작하고, 빨간 공이 들어오는 경우 좌우 회피를 하는 등 webcam2에서 발생하는 이벤트에 대해 우선권이 주어져 있다.

우리의 코드는 main함수의 while문에서 공의 정보를 기반으로 한 움직임을 결정하며, 매 행위마다 spinOnce() 함수를 실행하여 새로운 공의 정보를 업데이트하는 것을 요체로 한다. 또한 main 함수는 단일 while 문으로 구성되어 있으며, while 문은 여러 개의 if 문으로 구성되어 있다(그림 2 참고). 즉 이는 조건을 판단하는 순서가 중요하다는 의미이며, webcam2의 이벤트에 대해 우선권을 부여한다는 것은 webcam2에서 발생하는 이벤트를 고려하는 if문을 webcam1의 그것보다 앞에 두어야 한다는 뜻이 된다.

따라서 조건의 판단 순서는 1. release 여부, 2. pick up 여부, 3. webcam2 빨간 공 회피 여부, 4. webcam1 파란 공 중심 여부, 5. webcam1 빨간 공 회피 여부이며, 이 중 release와 pick up은 차후 별도의 함수로서 소개할 것이므로, 나머지 기능에 대해 설명한다(641행~).

2. webcam2 빨간 공 회피

```
else{//when blue ball is not dete
 if(web2_red_number != 0){//when
   if(web2_red_X > 0){//when x po
     //move left until red ball d
     while(web2_red_number != 0){
       move_left(1);
       ros::spinOnce();
       sleep_count(t);
     }
   }
   else{//when x posotion of red
     //move right until red ball
     while(web2_red_number != 0){
       move_right(1);
       ros::spinOnce();
       sleep_count(t);
```

그림 27 webcam2 빨간 공 회피

빨간 공 회피는 webcam1에서 이루어지며, webcam2에 빨간 공이 인식되는 경우는 예상치 못하게 Kirby가 공과 충돌하여 빨간 공이 이동경로 중 webcam1에 들어오지 않은 채 webcam2에 들어오는 경우이다. 즉 webcam2에서 빨간 공을 회피하는 것은 일종의 안전장치에 불과한 바, webcam1에서 빨간 공을 회피하는 경우와는 달리 빨간 공이 좌측, 우측이 있는지 판단하여 좌, 우로 빨간 공이 보이지 않을 때까지만 수평 이동한다.

또한 만일의 사태에 대비한다 하더라도, webcam2에 파란 공이 있어 pick up을 해야함에도 빨간 공 회피를 위해 파란 공을 피하는 것은 더 치명적이라고 판단하여, webcam2에 빨간 공과 파란 공이 함께 인식되는 경우는 회피 기동을 하지 않도록 설계하였다(그림 22 상단 else문).

3. webcam1 파란 공 중심 조정

시야에 보이는 파란 공을 쫓아가기 위해서는 파란 공을 시야의 중심에 두어야 하는데, 이러한 파란 공 추적 과정을 설계함에 있어 몇 가지 고려해야 하는 점이 있다. 첫번째는 하드웨어상으로 로봇의 전진방향과 카메라의 중심이 완벽히 일치할 수 없다는 점이며, 두번째는 중심조정을 위해 회전을 하더라도 미끄러짐 등의 이유로 완벽히 공을 카메라상 중심에 위치시키기 어렵다는 점, 그리고 마지막으로는 공의 위치가 카메라상 중심이 아닌 한 직진을 함으로써 점점 그 중심에서 멀어진다는 것이다.

따라서 우리는 위와 같은 문제에 대하여 '중심'이라고 판단할 수 있는 x 축상의 threshold 값을 설정하였다(-1.3 ~ 1.3). 다만 위와 같은 threshold에서 벗어났을 때 다시 해당

threshold에 들어오게 회전하도록 하는 경우, 조금만 직진을 하면 다시 threshold에서 벗어나 적절한 주행을 보기 힘든 경우가 발생할 수 있다. 따라서 우리는 그림 23과 같이 큰 threshold(-1.3~1.3)와 작은 threshold(-1.1~1.1)로 나누어 debouncing을 통해 해당 문제를 해결하였다.

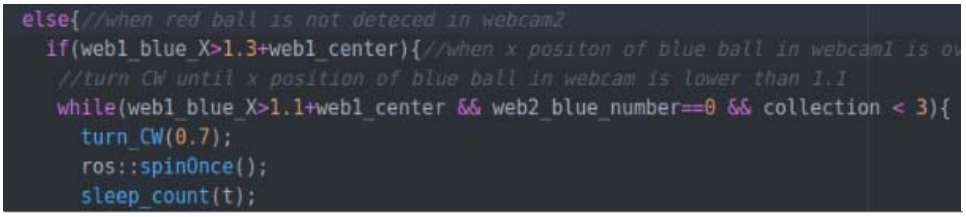

그림 28 파란 공 중심조정 – CW회전

즉 추적하는 파란 공의 x좌표가 1.3보다 큰 경우 CW로 회전하여 x좌표가 1.1이하가 될 때까지 회전하도록 하였다. 이는 회전을 정지하였을 때 생기는 미끄럼을 고려한 값으로, 실제 회전이 완전히 정지한 경우 파란 공의 x좌표는 0에 가까운 값이 되는 것을 실험적으로 알 수 있었다.

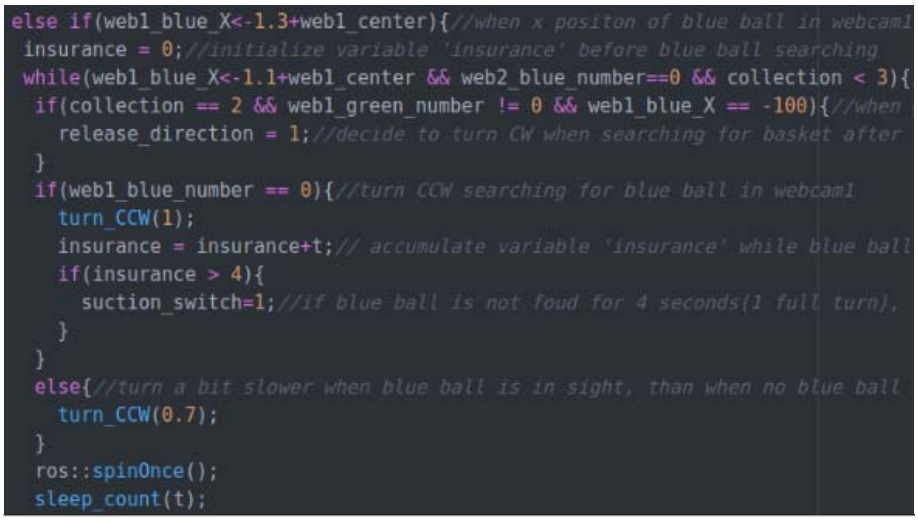

그림 29 파란 공 추적 - CCW 회전

상술하였듯이 Kirby는 파란 공이 보이지 않는 경우 CCW로 회전하는 것을 기본으로 하며, 따라서 CCW 회전시 발생하는 조건을 고려하여 안정장치를 다소 마련하였다.

파란 공의 x 좌표가 -1.3 보다 작을 때에는 -1.1 보다 클 때까지 CW 회전하며, 앞서 callback 함수에서 파란 공의 x값을 -100으로 초기화하는 것은 파란 공이 보이지 않는 경우 x값을 -100으로 받아 CCW로 회전하여 공을 찾게 하기 위함이었음을 알 수 있다.

그림 24에서 보이는 'release_direction'은 공을 3개 수집한 이후 바구니를 찾기 위해 무조건 CCW로 회전하는 것을 방지하기 위해 도입된 변수로, 공을 2개 수집하고 마지막 공을 찾기 위해 CCW로 회전하는 와중에 초록색공이 보이는 경우 release_direction에 1을 저장한다. 이 release_direction이 1인 경우 release mode에서 바구니를 찾기 위해 회전할 때 CW로 회전하게 된다.

그림 24에서 볼 수 있는 insurance는, webcam2에 파란 공 2개가 동시에 보여 수집한 공의 개수를 1개만 카운트하는 만일의 경우에 대비한 것이다. 즉, 매번 파란 공을 찾기 위해 돌기 시작할 때마다 Kirby는 insurance 변수를 0으로 초기화하며, 파란 공을 찾기 위해 회전하는

시간이 4초(약 한 바퀴)를 넘으면 suction_switch를 1로 바꾸면서 수집한 공의 개수에 임의로 1을 더하게 된다. 공을 다 수집하였음에도 수집한 공의 개수가 3개가 되지 않는 경우에 대비한 안전장치이다.

또한 파란 공을 찾기 위해 회전하는 시간을 줄이기 위해 파란 공이 시야에 보이지 않는 경우는 최대속도(1)로 회전하게 만들었으며, 파란 공이 시야에 보이는 경우는 회전 속도를 조금 줄여(0.7) 정확도를 높이도록 설계하였다.

4. webcam1 빨간 공 회피

기본적으로 빨간 공의 회피는 파란 공을 추적하고 있는 도중에만 의미가 있는 바, 그림25 상단의 else문은 파란 공이 시야의 중심에 있을 때를 의미한다. 빨간 공의 y좌표가 3.5라는 것은

실제 Kirby와 빨간 공 사이의 거리가 약 30cm일 때를 의미하며, 해당 거리에서 빨간 공이 보이지 않을 때까지 좌/우로 수평이동 한 후 그림26의 직진 기동을 일정 거리 하게 되는 경우 빨간 공과 충돌하지 않으면서 완전한 회피가 가능하다.

```
else{//webl_blue_X at middle
   if(webl_red_Y > 3.5){//start avoiding when red ball is close
      if(webl_red_X>0+webl_center){//when red ball is on right side
        //move left until red ball disappears from webcam1
        //do not avoid red ball far away from kirby
        while(webl_red_number!=0 && webl_red_Y > 3 && web2_blue_number==0 && collection < 3){
        move_left(1);
        ros::spinOnce();
        sleep_count(t);
        }
```

그림 30 빨간 공 회피 - 수평이동

수평이동 후 직진을 하는 경우 일정 거리 이상 직진 이동을 해야 이후 파란 공을 찾기 위해 회전을 할 때에 빨간 공이 다시 webcam1에 들어오지 않는다. 따라서 우리는 반복적인 실험을 통해 약 0.8초동안 최고속도로 직진을 하는 경우 적절한 거리의 직진을 한다는 것을 알아낼 수 있었다.

```
float k = 0;//initializing variable 'k
bool c = true;//initializing 'c' as tr
//go forward for 0.8 seconds
while(k<0.8 && c){
  ros::spinOnce();
  if(web2_blue_number != 0){//when blu
    pick_up();
    c = false;//stop and get out of th
  }
  data[0] = 0;
  data[1] = 1;
  data[4] = 0;
  data[5] = 0;
  sleep_count(t);
  suction_check();
  write(c_socket, data, sizeof(data));
  k=k+t;
```

그림 31 빨간 공 회피 – 직진

다만 일정 거리 직진을 하는 와중에도 혹여 파란 공이 webcam2의 시야에 들어왔을 때 이를 무시한다면 공을 쳐버려 이동 루트가 과도히 길어질 수 있으므로, 해당 직진 while loop를 폐루프로 구성하여 pick up을 진행한 후 Boolean c를 이용하여 직진 while loop를 탈출하게끔 만들었다.

좌측으로 수평 이동하며 빨간 공을 회피하는 경우, 직진한 후 파란 공을 찾기 위해 원래의 파란 공 탐색 과정대로 CCW 회전을 하는 경우 기존에 추적하던 파란 공을 놓칠 염려가 생기게 된다.

```
k=0;//initializing variable 'k'
//turn CW for 0.5 seconds
//do not turn CW when pick up was done
while(k<0.5 && c){
  data[0] = 0;
  data[1] = 0;
  data[4] = 0.8;
  data[5] = 0;
  suction_check();
  write(c_socket, data, sizeof(data));
  sleep_count(t);
  k=k+t;
```

그림 32 빨간 공 회피 – 직진 후 재회전

따라서 그러한 상황을 방지하기 위해 우리는 좌측 수평 이동을 통해 회피를 진행한 경우만 직진 이동 후 CW 회전을 0.5초 동안 행하게 만들었다(그림 27).

또한 이 경우도 마찬가지로, '회피 후 직진' 중 pick up 을 시행하였다면 0.5 초의 회전이 무의미하므로 Boolean c 를 통해 생략하도록 했다.

VI. Pick_up

1. Suction_check()

Pick up mode에서 suction motor를 켜고 공이 파이프로 들어간 후 1.5초 이후에 끄게 하는 함수이다. 1.5초 이후에 끄는 이유는 공이 파이프를 타고 올라가 저장소에 들어가는데 걸리는 시간이 약 1초 가량이기 때문이다.

```
  if(suc<=1.5){
    data[21]=1;
  }
  else{
    data[21]=0;
  }
}
```

그림 33. Suction 가동

Suc 변수는 pick up function이 실행될 때마다 0으로 초기화되고 pick up function을 탈출했을 때 (webcam 2에 파란 공이 사라졌을 때) 탈출한 후 지난 시간이 저장되므로, suc 변수가 1.5이하일 때 suction motor를 켜도록 설계하면 해당 알고리즘을 구현 가능하다. suc 초기값은 1.5보다 커야 ROS 실행 직후 suction motor가 켜지지 않으므로 10으로 설정한다.

2. Sleep_count()

```
ros::Duration(sleeprate).sleep();

suc = suc + sleeprate;
```

그림 34. 시간 update

Node를 지정한 sleep period만큼 sleep시키고, 해당 sleep period를 suc 변수에 더해 suc 변수가 pick up function을 탈출한 후 지난 시간을 저장하게 만든다. 또한 파란 공을 pick up한 경우 수집한 공의 개수를 지정하는 변수 collection을 1 증가시킨다.

```
if(collection<2){
  if(0.6<=suc && suction_switch ==1){
      collection = collection + 1;
      suction_switch =0;
      }
    }
//do the ball counting faster when las
else{
  if(0.1<=suc && suction_switch ==1){
    collection = collection + 1;
    suction_switch = 0;
  }
}
```

그림 35. Collection 추가

Webcam 2에서 파란 공이 사라졌다면(pick up function이 종료되었다면) pipe 안으로 들어와 수집된 것이므로 collection을 1 증가시켜야 한다. 이를 위해 0과 1의 값을 가지는 suction_switch라는 변수를 도입한다. Pick up function이 실행될 경우 suction_switch를 1로 초기화하고 Pick up function 밖에서 suction_switch가 1일 경우 collection을 증가시키고 suction_switch를 다시 0으로 꺼주면 pick up이 완료될 때 마다 collection을 1 증가시키게 만들 수 있다.

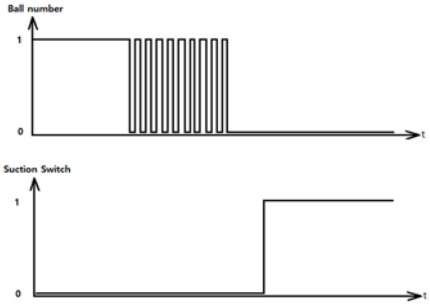

그림 36. Debouncing

하지만 webcam, 파란 공 진동, 노이즈 등이 심해 파란 공이 사라졌을 때 곧바로 collection을 증가시킨다면 collection이 1보다 많이 증가할 수 있다. 따라서 pick up function이 끝나고 0.6초 이후에 collection을 1 증가시키게끔 조건을 설정하여 debouncing하였다. Collection이 2일 때는 마지막 공에 대한 counting이므로 collection이 1보다 많이 증가해도 release mode로 넘어가는 것은 동일하므로 보다 빠르게 release mode로 변경시키기 위해 0.6초가 아니라 0.1초로 debouncing하였다.

3. Pick_up()

```
if(web2_blue_X > -2.0 && web2_blue_X < 1.7){
  suction_switch = 1; //turn on suction_switc
}
suc = 0;//initializing suc in order to turn
```

그림 37. Suction switch 초기화

Ball counting과 suction motor 가동을 위해 suction_switch를 1로 초기화한다. 다만 실험 중 가끔 파란 공이 webcam2의 가장자리에 아주 작게 위치했다가 사라져 pick up을 못했음에도 suction_switch가 1로 켜져 collection이 증가하는 case가 발생하여 webcam2의 가장자리에 파란 공이 위치할 경우는 suction_switch를 1로 켜지 않도록 처리했다. suc 변수는 0으로 초기화하여 pick_up function이 실행과 동시에 suction motor를 가동시킨다.

```
if(web2_blue_X>0.8+web2_center){//if x position of blue ball is
 while(web2_blue_X>0.5+web2_center && web2_blue_number != 0){//
   turn_CW(0.28);
   ros::spinOnce();
   ros::Duration(t).sleep();
 }
}
else if(web2_blue_X<-0.8+web2_center){//if x position of blue b
 while(web2_blue_X<-0.5+web2_center && web2_blue_number != 0){/
  turn_CCW(0.28);
  ros::spinOnce();
  ros::Duration(t).sleep();
  }
 }
else{//go forward if blue ball is in the middle
 move_forward(0.3);
}
```

그림 38. 파란 공 접근

Suction motor를 가동시킨 채로 webcam 2를 이용해 파란 공에 가까이 접근하면 파란 공이 pipe 안으로 빨려 들어간다. 파란 공에 대한 접근은 webcam2 상에서 파란 공이 중앙 범위 밖일 경우 회전하여 중앙 조정을 하고 중앙 범위 안일 경우 직진하도록 하여 구현했다.

Ⅶ. Release

우리의 release 알고리즘은 가까운 초록 공을 향해 충분히 가깝게 직진한 후, 좌우 바구니 방향으로 초록 공이 webcam 2에 안보일 때까지(혹은 둘다 보이지만 화면 양 끝에 존재할 때까지) 이동하여 pipe 출구를 바구니 중앙에 위치시킨 다음, servo motor를 가동해 cap을 열어 바구니 안으로 공을 release하는 것이다. 경기장 규격과 ball position 규칙을 토대로 각도를 계산한 결과 해당 알고리즘이 충분히 안정적이라고 판단했고 실제로 약 30번의 실험 중 공이 바구니 안에서 서로 부딪혀 탄성에 의해 튕겨 나가는 경우 한 번을 제외하고는 release에 실패한 적은 없었다. 해당 알고리즘의 장점은 release process가 매우 빠르다는 것인데 실제로 demo에서 release time은 다른 조의 40-70%밖에 되지 않았고 이는 1차시도 34초로 1등을 할 수 있었던 큰 요인으로 꼽힌다.

Release function은 바구니 중심 조정 및 가까운 초록 공 판단, 가까운 초록 공을 향한 직진, 좌우 이동 및 servo motor 가동의 세 가지 파트로 나뉜다.

```
if(web1_green_number <2){//whe
  if(release_direction == 0){/
    turn_CCW(0.4);
  }
  else{//turn CW when green ba
    turn_CW(0.4);
  }
}
```

그림 39. 중심 조정 회전 1

바구니 중심 조정 과정은 초록 공 2개 사이의 중점이 webcam 1의 중앙에 위치하도록 회전하는 것이다. 먼저 webcam 1에 초록 공이 2개 미만일 경우 초록 공이 2개가 보이도록 회전한다. 이 회전에서 소요되는 시간이 상당히 크므로 회전방향을 clock wise와 counter clock wise 중 회전 각도가 최소가 되도록 지정해야 한다. 이는 main function에서 저장된 release_direction 변수(main function part 참고)를 이용해 구현 가능하다.

```
if(web1_green_X_average > 1.3+web1_center){//when average x position of 2 green balls is over 1.3
  //turn CW until the average x position of 2 green balls is lower than 0.6
  while(web1_green_X_average>0.6+web1_center && web1_green_number>=2 && web2_green_number ==0){
    turn_CW(0.35);
    ros::spinOnce();
    sleep_count(t);
  }
}
else if(web1_green_X_average < -1.3+web1_center){//when average x position of 2 green balls is lo
  //turn CCW until the average x position of 2 green balls is over -0.6
  while(web1_green_X_average <-0.6+web1_center && web1_green_number>=2 && web2_green_number ==0){
    turn_CCW(0.35);
    ros::spinOnce();
    sleep_count(t);
  }
}
else{
  a=false;//change 'a' to false - do the center targeting only once during the drive
}
```

그림 40 중심 조정 회전 2

Webcam 1에서 초록 공 2개가 존재할 경우 초록 공 2개의 x좌표 중점(web1_green_X_average)이 webcam 1의 중심 범위 안으로 들어오게끔 회전한다. x좌표 중점이 중심 범위 안으로 들어올 경우 중심조정이 완료된 것이므로 a 변수를 false로 변경하여 중심조정 while문에서 빠져나온다.

```
// decide closest ball (left? right?)
if(abs(web1_green_X_min-web1_green_X_closest)<0.05){
    leftright = 0;    // left = closest.
}
else{
    leftright = 1;    // right = closest.
}
```

그림 41. target 설정

가장 가까운 초록 공의 x좌표가 left 초록 공 x좌표와 같으면 왼쪽이, 다르면 오른쪽이 가까운 공이다. 조건문에서는 혹시 모를 오차를 감안하여 좌표차이가 0.05보다 작을 때 같은 좌표로 인식하게 만들었다. leftright가 0일 경우 callback function에서 target 초록 공(web1_green_X_target)을 왼쪽 초록 공으로 지정하고, 1일 경우 오른쪽 초록 공으로 지정한다. 따라서 가까운 초록 공을 향해 직진하기 위해선 target 초록 공을 향해 직진하면 된다.

가까운 공을 향한 직진 알고리즘은 다음과 같다. webcam 2에 초록 공이 없을 경우 앞서 지정한 target 초록 공 x좌표(web1_green_X_target)가 webcam 1 중심 범위 밖이면 중심 범위 안으로 오게끔 회전하고 중심 범위 안이면 직진한다.

```
if (leftright == 0){//when the green ball detedcted is left ball
  //move right until green ball disappears from webcam2
  while(web2_green_number != 0 && web2_green_X_max-web2_green_X_min<400){
    move_right(0.4);
    ros::spinOnce();
    sleep_count(t);
  }
```

그림 42 web2 초록 공 approach

초록 공에 가까워져 webcam 2에 초록 공이 들어올 경우 webcam 2를 통해 중심 조정 없이 직진만 하여 초록 공과 거의 딱 붙을 정도로 가까이 다가간다. 중심 조정과정이 없는 이유는 가까이에서 조정 회전을 할 경우 각도가 크게 변해 좌우 이동시에 문제될 수 있기 때문이다.

좌우 이동 알고리즘은 다음과 같다. 먼저 leftright 변수가 0이면 왼쪽 공에 도착했다는 의미이므로 오른쪽으로 이동해야 하고 leftright가 1이면 반대로 왼쪽으로 이동해야 한다. Pipe 출구가 바구니 안쪽으로 오게 해야 하므로 webcam 2 상에서 초록 공이 없어질 때까지 이동한다.

```
for(float k=0; k<2; k=k+t){
 data[20]=1;
 data[0]=0;
 data[1]=0;
 data[4]=0;
 data[5]=0;
 write(c_socket, data, sizeof(data));
 sleep_count(t);
}
dataInit();//do nothing
break; //end while looop
```

그림 43. Release and end

좌우 이동이 종료되면 servo motor를 가동하여 저장된 ball을 releasing하고 dataInit() 함수로 모든 command를 멈춘다. 이후 break를 통해 while문을 탈출, release() function을 탈출하게 되고 main function에서 break가 되어 node가 종료된다.

Ⅷ. Desired coding schedule

-4 월 20 일) Concept design 참여

4 월 20-24 일) Algorithm 확정

4 월 25 일-5 월 2 일) Moving part coding

5 월 3 일-11 일) Avoid part coding

5 월 11 일-17 일) Pick up part coding

5 월 18 일-23 일) Release part coding

5 월 24 일-Final Demo day) 세부 조정 및 optimization

예상치 못한 문제들이 상당수 존재하여 debugging 시간이 많이 소요되므로 coding 을 빠르게 시작하는 것이 좋다. 위의 coding schedule은 우리 팀의 경우였고 당연하게도 제품의 hardware나 주행 algorithm이 다르면 그에 맞게 schedule을 조정해야 한다.