자율 주행 및 Google\_Cartographer

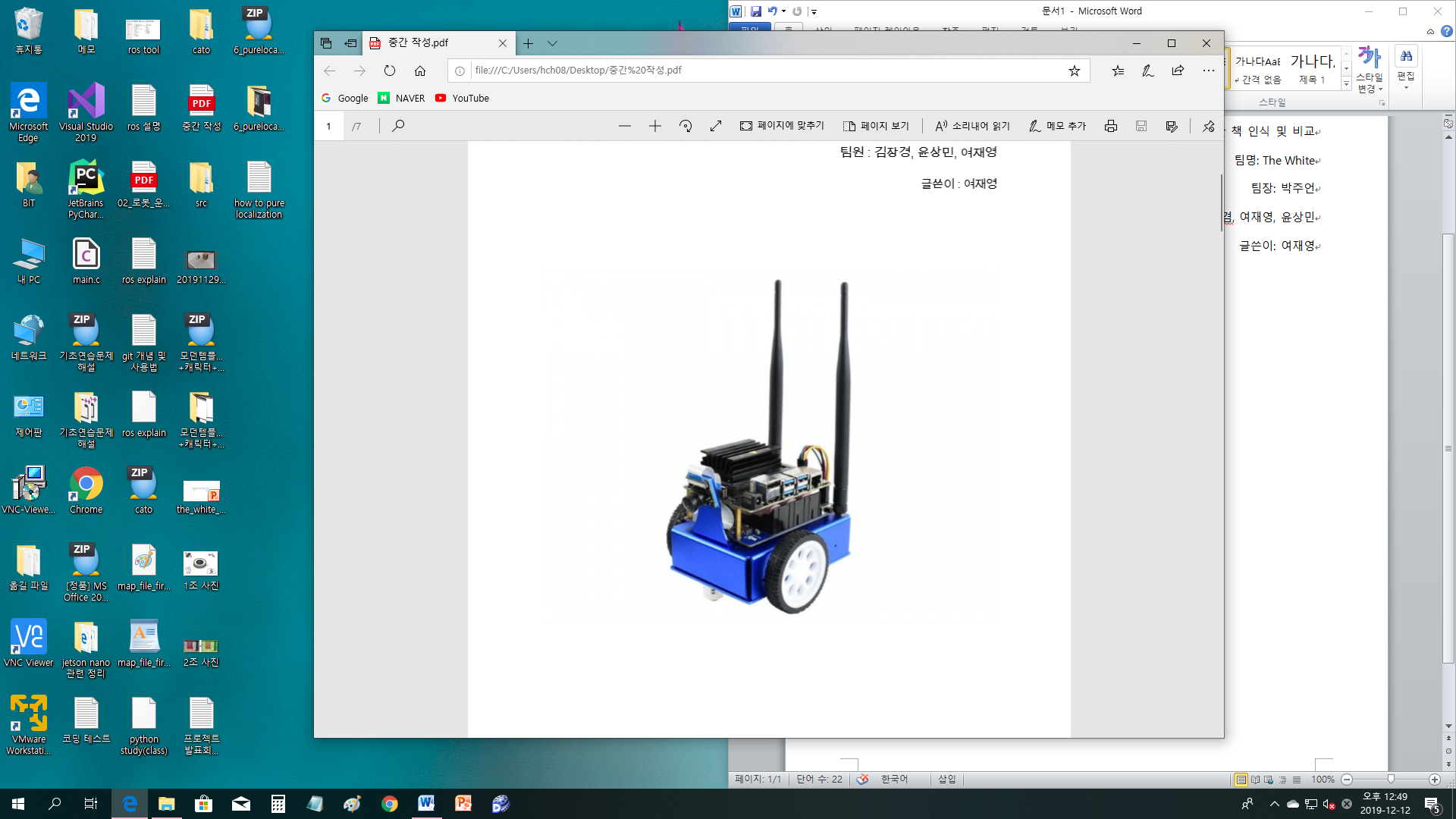
주제: 자율주행을 통한 책 인식 및 비교

팀명: The White

팀장: 박주언

팀원: 김장겸, 여재영, 윤상민

글쓴이: 여재영



목 차

(0) 개 요

(1) 자율주행

(2) 로봇(Jetbot)과 필요한 부품들

(3) ROS

(4) ROS & Google\_Cartographer Install

(5) Map Save & Load

(6) Map에서의 현재 위치(tf)

(7) BFS

(8) A\* Algorithm

(0) 개요

대규모 서점 및 도서관에서 책을 정리하는 일은 많은 인적자원과 시간을 소비하는 작업입니다.

본 프로젝트는 머신러닝 및 자율 주행 기술을 활용하여 이러한 작업을 도와줄 수 있는 로봇을 개발하면 어떨까 하는 아이디어에서 출발했습니다.

2D Lidar 및 IMU 센서 데이터를 활용하여 Cartographer로 Map을 제작 후 현재 위치를 파악하며 지정된 책장까지 최단거리를 계산합니다.

지정된 책장 위치로 자율 주행을 한 후에 책장을 인식하여 촬영을 시작합니다.

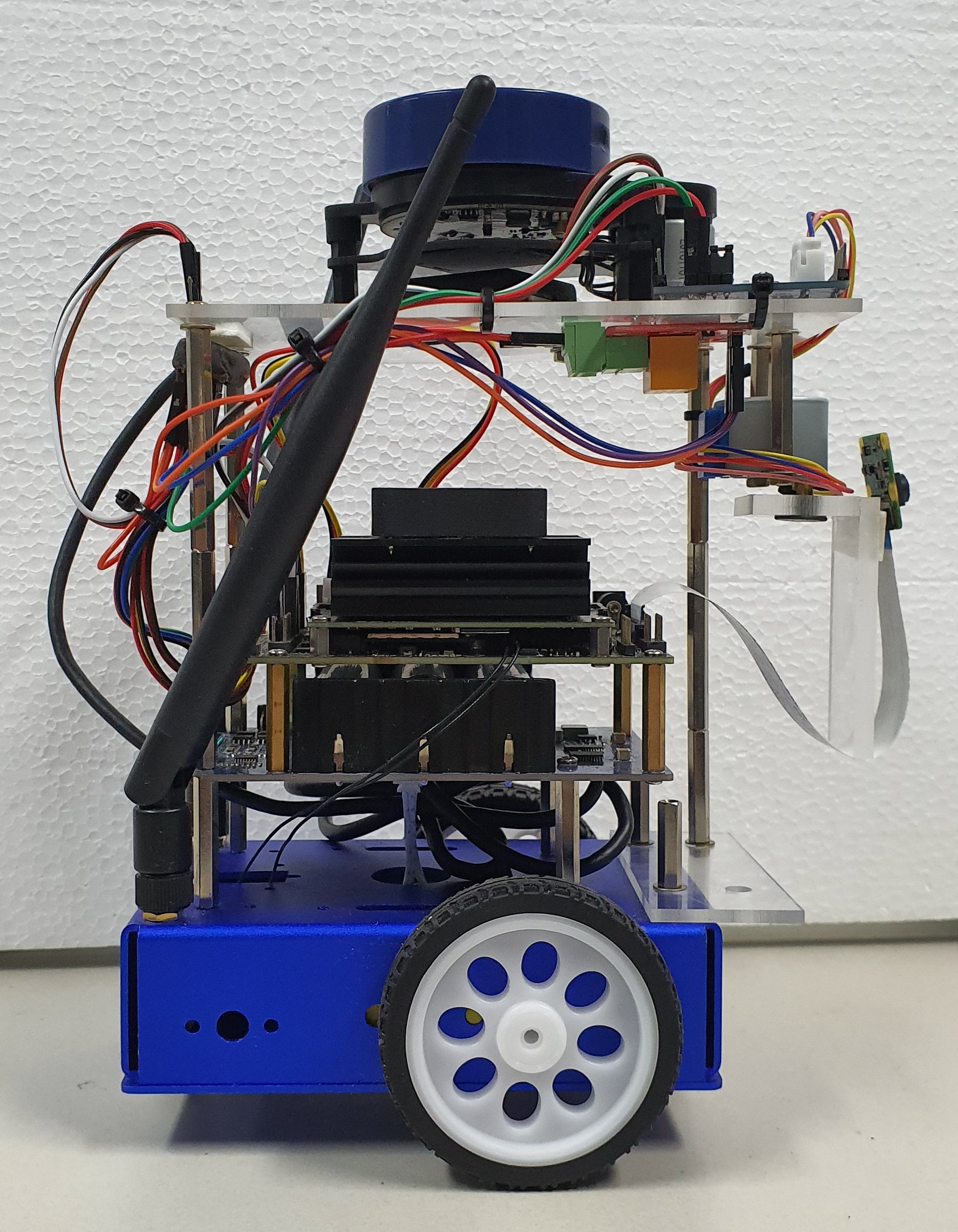
본 프로젝트는 많은 인적 자원, 시간 등을 절약할 수 있으며 책장이나 선반에 책을 인식해서 비교하는 것과 같이 비슷한 작업에서 연동시켜 응용해 사용이 가능합니다.

(1) 자율주행

요즘 나오는 자율주행은 카메라, 근접 센서 등을 사용하여 주행을 하지만 구글 등에 올라온 예를 보면 카메라를 사용하여 차선을 인식하여 주행하는 경우가 많습니다.

우리의 주제는 책장이나 선반에 사물을 분류하는 것으로 서점이나 도서관을 목표로 하기에 바닥에 라인을 그리는 것은 무리가 있어서 Lidar와 Google\_Cartographer를 사용해서 Map을 만들어서 원하는 지점까지 최단경로로 가서 사진을 찍고 사물을 인식하기로 정하였습니다.

(2) 로봇(Jetbot)과 필요 부품들



1) Ydlidar – X2: Map에 필요한 정보를 인식.

2) Raspberry\_Pi Camera: 사물 인식을 위한 카메라.

3) Step Motor : 카메라를 원하는 방향으로 회전시킴.

4) Relay **:** Step Motor을 끄고 킬 수 있게 함.(켜두면 온도가 올라감)

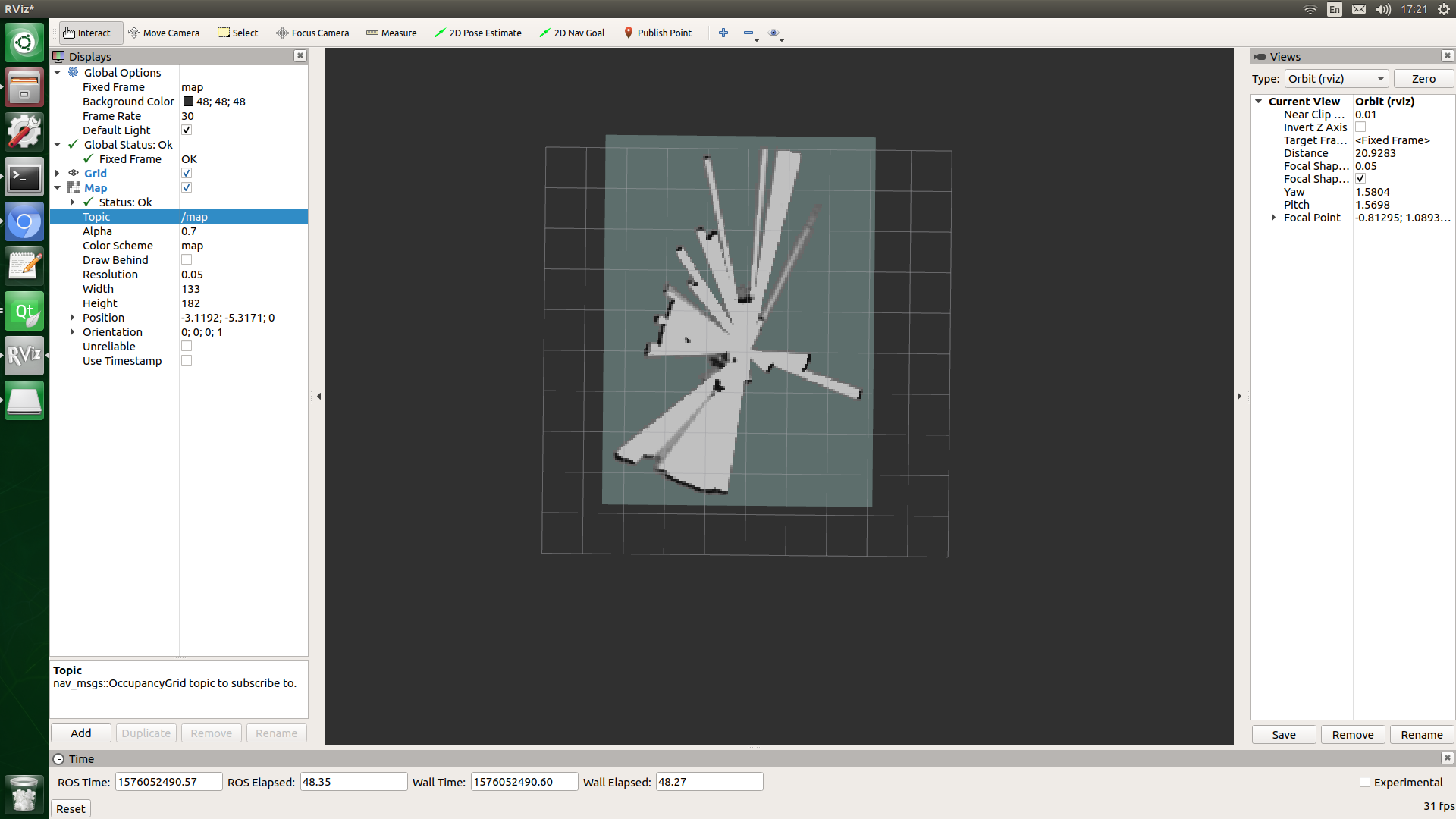
5) Lsm303(IMU) : 로봇의 회전 방향을 정확히 알기 위함.

6) Jetbot : Jetson Nano를 기반으로 한 자동차 모형.

(3) ROS

Map을 만드는 Google Cartographer와 IMU, 다른 부품들을 한번에 다루기 위해 Ros를 사용하기로 했습니다.

Ros란 Robot Operating System의 약자로 로봇 소프트웨어 플랫폼이며 센서나 통신 개발 환경 등을 제공합니다, 특히 다른 기계, 언어 간의 통신이 쉬운 장점이 있고 각종 여러가지의 Tool을 제공합니다.



↑(Ridar로 읽은 정보를 Rviz란 Tool을 사용해 Map을 생성 & Cartographer\_ex)

Ros 용어)

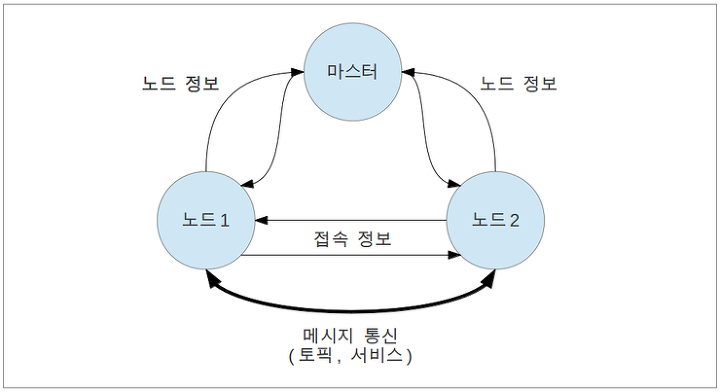
node : 실행되는 최소 단위의 프로세서, 하나의 프로그램 이기도 하며 topic이나 service를 이용하여 통신이 가능.

master : node와 node 사이를 연결하는 역할이며 master를 키지 않는 한 통신이 불가능. (roscore로 master를 키며 키지 않을 경우 IP오류를 띄움)

message : message를 통해 node와 node 사이에 데이터를 주고 받음.

topic : topic이란 노드 간에 통신을 할 수 있는 채널의 개념이며 message의 이름. (publish -> subscribe란 구조로 한쪽 방향으로 넘겨주며 여러 번 통신이 가능)

service : service는 topic과는 반대로 서로 통신이 가능하지만 한번 통신 후 연결이 끊긴다.



4) ROS & Google\_Cartographer Install

//////////////////////////////////Ros install/////////////////////////////////////////////

ROS는 Melodic 버전을 설치하였습니다.

공식 가이드: http://wiki.ros.org/melodic/Installation/Ubuntu

1) sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb\_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

2) sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654

3) sudo apt-get update

4) sudo apt-get install ros-melodic-desktop-full // rviz등의 시각화 툴이 포함되어있는 Desktop-Full 버전을 설치합니다.

5) sudo rosdep init // rodep를 초기화합니다.

6) rosdep update

7) echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc // ~/.bashrc에 설정을 추가합니다.

8) source ~/.bashrc

9) sudo apt install python-rosinstall python-rosinstall-generator python-wstool build-essential // 개발에 필수적인 dependencies들을 설치합니다.

/////////////////////////////Cartographer install//////////////////////////////////////

1) mkdir -p ~/carto/src // 작업을 할 파일을 생성합니다.

2) cd ~/carto/src // 그 파일로 이동 후

3) catkin\_init\_workspace // (작업공간 초기화인데 안해도 상관 없음)

4) cd ~/carto&&catkin\_make // carto파일로 이동후 compile을 해줍니다.

5) echo "source ~/carto/devel/setup.bash" >> ~/.bashrc // bashrc에 경로 추가

6) source ~/.bashrc // 경로 추가 후 터미널을 껐다 켜야하지만 이 코드를 입력 하면 해결됨.

7) cd src // src(script)파일로 이동.

8) git clone ​https://github.com/msadowski/x2\_cartographer.git​ // git에서 코드를 다운 cd x2\_cartographer // 받은 파일로 이동을 합니다.

9) rm -y ydlidar // 그리고 ydlidar 파일을 삭제(ydlidar 주소가 잘못 되어있음)

10) git clone git://github.com/YDLIDAR/ydlidar\_ros.git ydlidar // 그리고 ydlidar를 직접 받습니다.

11) cd ../.. // carto 파일로 다시 이동 합니다.

12) rosdep install --from-paths src --ignore-src --rosdistro melodic -r –y

13) source devel/setup.bash

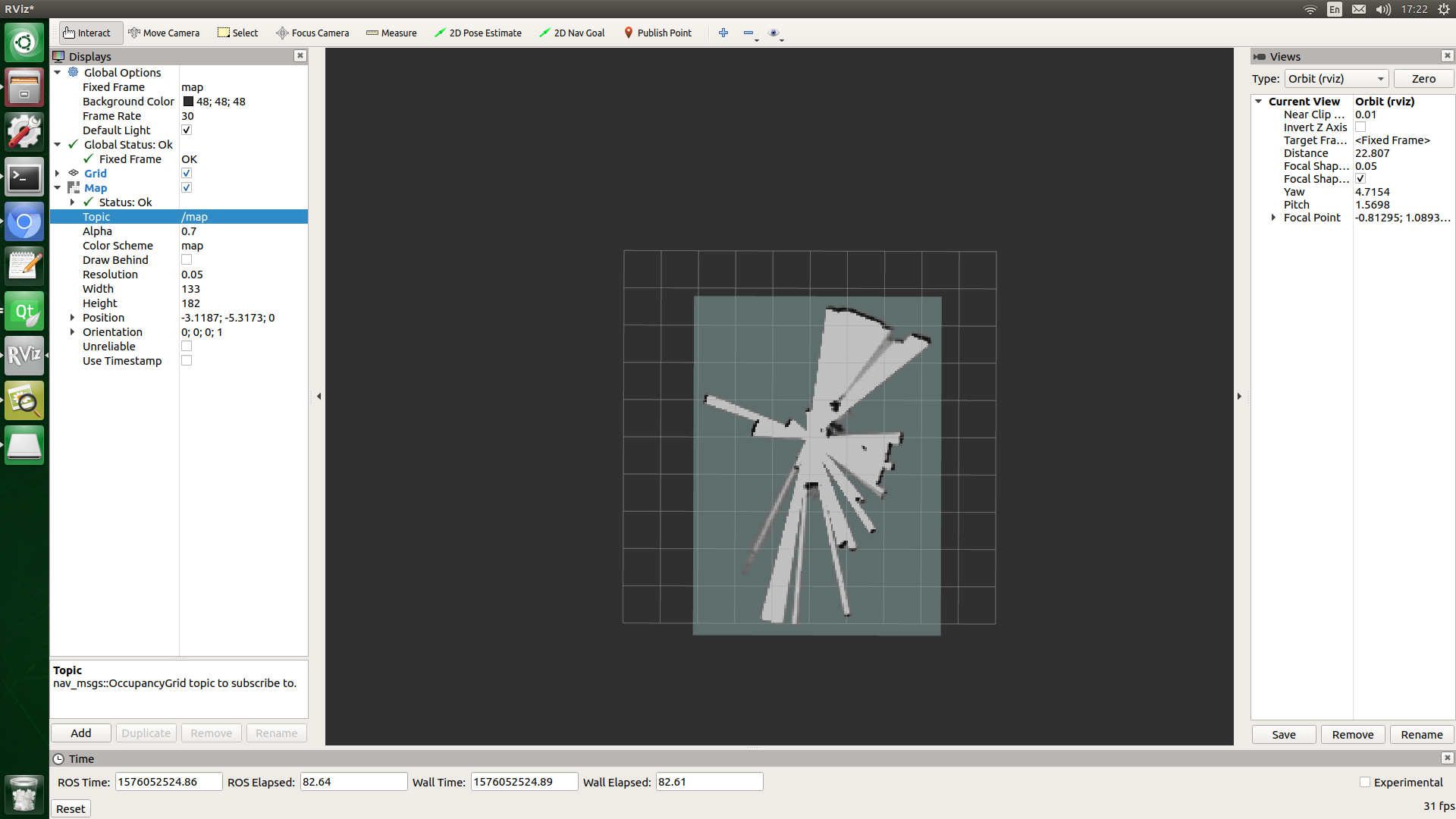
14) catkin\_make // compile(carto 파일에서만 해야함)

실행) roslaunch carto\_mapper mapper.launch // cartographer를 실행 해 줍니다.

5) Map\_Save & Load

roslaunch carto\_mapper mapper.launch을 실행하면 Rviz가 같이 실행되는데

왼쪽 아래에 Add버튼을 눌러서 목록에서 Map을 찾아주시고 추가하신 다음에 Map을 누르고 Topic을 찾아서 /map을 추가해 줍니다.



기계를 움직여 Map을 다 완성했다면 pbstream으로 저장을 합니다.

1) rosservice call /finish\_trajectory 0 //을 입력해서 프로그램을 중지시킵니다.

2) rosservice call /write\_state '/home/jetbot/map.bag.pbstream' // 그 후 원하는 위치와 이름을 정해줍니다.(rosservice call /write\_state '(위치 경로)/(이름.pbstream)')

3) 그 후에 x2\_cartographer -> carto\_mapper -> launch파일에 들어가서 mapper를 수정합니다.

-configuration\_basename x2.lua"

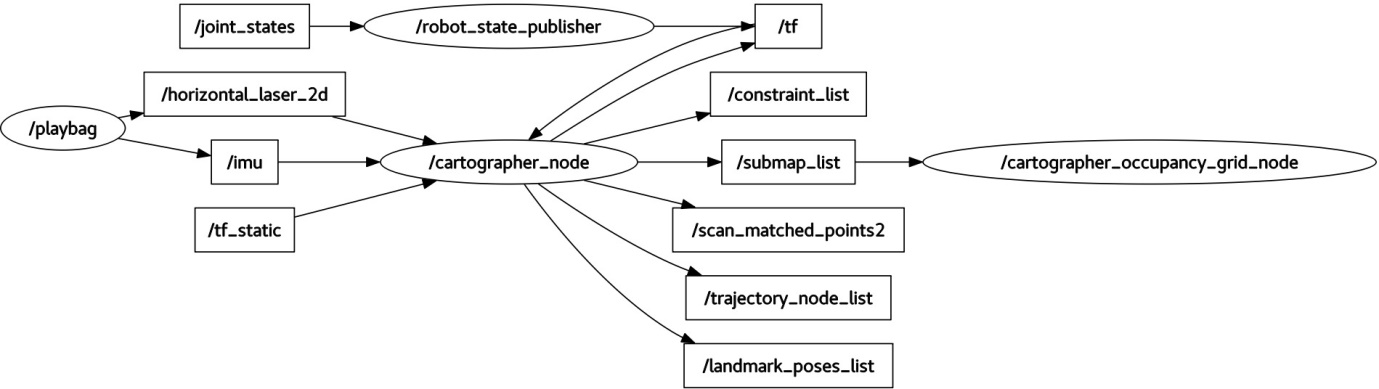
-load\_state\_filename /home/jetbot/map.bag.pbstream" x2.// lua 아래에 추가!

4) 마지막으로 아래 문장을 넣으면 저장된 Map과 지금 불러오는 정보를 조합해 기계와 Map의 위치 방향 등을 자동으로 잡아줍니다.

roslaunch carto\_mapper pure\_localization.launch load\_state\_filename:=${HOME}/map.bag.pbstream

load\_state\_filename:=파일 위치/ 파일 이름(${HOME}/이 안되면 /home/.. 이렇게 직접 설정)

6) Map에서의 현재 위치(tf)

위에 그림은 Cartographer가 실행 될 때 Node 들 간의 통신 상태를 나타낸 것인데 우리는 이것을 이용하여 Map에서의 현재 값 위치를 나타낼 수 있습니다.

<http://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20listener%20%28C%2B%2B%29>

위에 사이트에 들어가보면 tf로 현재 위치 값을 받을 수 있는 코드가 설명이 되어 있습니다.

#include <tf/transform\_listener.h>

tf::TransformListener listener;

ros::Rate rate(10.0);

while (node.ok()){

tf::StampedTransform transform;

try{

listener.lookupTransform("/base\_footprint", "/map",

ros::Time(0), transform);

}

catch (tf::TransformException &ex) {

ROS\_ERROR("%s",ex.what());

ros::Duration(1.0).sleep();

continue;

}

double theta = atan2(transform.getOrigin().y(),

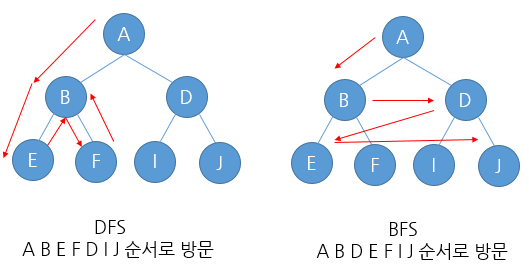
transform.getOrigin().x());

double x = transform.getOrigin().x();

double y = transform.getOrigin().y();

위에 코드에서 transform.getOrigin().x() or y()로 좌표값과 arctan를 이용하여 theta도 구할 수 있습니다.

7) BFS(너비우선 탐색)

이제 좌표를 알고 시작점과 도착점을 설정해야 하는데 큰 Map을 작성할 경우 한 Map에 한 Pixcel에 값이 갈 수 있는지 없는지 하나하나 알 수 없기에 근처 좌표를 지정하면 근처 가능한 점을 시작점으로 잡습니다.

그래서 BFS(너비우선 탐색)으로 시작점과 끝점을 잡습니다.

너비 탐색을 할 때 queue와 set을 사용하여 프로그램을 구현했습니다.

queue에 값을 넣어서 차례대로 값을 넣고 set에는 갈 수 없는 부분을 골라내어서 넣어 줍니다 다음으로 넘어가 같은 곳을 탐색해 넣는다 해도 set이 중복을 제거해 줍니다.

그리고 queue에 A를 넣고 A를 꺼내어 인접한 B, D를 넣습니다.

다시 맨 앞의 값을 빼내면 B가 되므로 B에 인접한 E,F를 넣고 다시 앞의 값을 꺼내면 D이므로 인접한 I와 J를 넣으면 모든 값을 가까운 순서로 구할 수 있습니다.8) A\* Algorithm

Map을 사용하여 현재의 로봇과 동기화, 현재 위치를 알 수 있으니 이것을 토대로 우리가 원하는 위치까지 최단경로로 갈려고 합니다.

이때 최단경로 알고리즘으로 많이 쓰이는 A\*를 사용해서 최단경로 좌표를 구합니다.

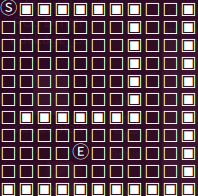
A\*는 지나온 목록(closelist), 갈 수 있는 목록(openlist) 2가지 목록을 만들고 현재 위치에서 어느쪽으로 갈 수 있는지 파악하여, 지나갈 수 있으면 openlist에 넣고 그 중에 목표 값으로까지의 최소값을 골라 지나온 목록에다가 넣습니다. (그 후 openlist 제외)

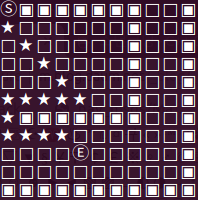
(목표 값까지의 계산)

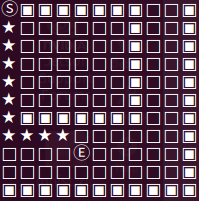
***F = G + H***

* G - 시작점 A로부터 현재 사각형까지의 경로를 따라 이동하는데 소요되는 비용입니다.
* H - 현재 사각형에서 목적지까지의 예상 이동 비용입니다.
* F - 현재까지 이동하는데 걸린 비용과 예상 비용을 합친 총 비용입니다.
* 그리고 openlist에 값을 넣을 때 마다 openlist에 있는 값을 비교해 좌표 값이 같고 G값이 최소인 것을 대신하여 넣어서 잘못된 길을 간다 해도 다시 최소로 가는 길을 구할 수 있습니다

S는 시작지점 E 도착지점.



 A\* 알고리즘을 쓰지 않는다면 시작점부터 최소 경로로 가다가 벽을 만나서 돌아가게 됩니다.

 A\* 알고리즘은 다른 벽을 지나침 없이 바로 정해둔 목표로 향합니다.