Project: Scan Matching Localization

(English Version)

#1. Project Goal:

Localization refers to the process by which an autonomous vehicle accurately identifies its current location and pose (position and direction). In other words, the vehicle must continuously update its location based on its surroundings (map data, map.pcd file).

Lidar: Collect point cloud data around the vehicle through sensors (provided by the carla simulator)

Map.pcd: Compare the static point cloud map (static map data) stored in this file with the collected Lidar data to calculate the vehicle's location (pose)

1. Maintain pose error of 1.2m or less: The error between the current location and the calculated location must not exceed 1.2m while the vehicle is driving.

2. Drive more than 170m: The vehicle must drive at least 170m at medium speed.

3. Implement localization algorithm: You must write a code to calculate the vehicle's location by filtering (voxel filter) and matching (ICP or NDT) Lidar data. Students just need to write the todo section in the c3-main.cpp file.

#2. C3-main.cpp :

##Receiving LiDAR data:

The following code receives the LiDAR data generated by the Carla simulator.

```
lidar->Listen([&new_scan, &lastScanTime, &scanCloud](auto data){
    auto scan = boost::static_pointer_cast<csd::LidarMeasurement>(data);
    for (auto detection : *scan){
        pclCloud.points.push_back(PointT(detection.x, detection.y, detection.z));
    }
```

```
    *scanCloud = pclCloud;
});
```

Lidar data is stored in a variable called pclCloud, copied to scanCloud and used later.

Transfer-1: Put pclcloud in scanCloud and return

##Improving the quality of lidar data:

Remove unnecessary noise using a voxel filter, and perform calculations using the filtered points.

```
// TODO: (Filter scan using voxel filter)
// TODO: (복셀 필터를 사용하여 스캔 데이터 필터링)
 pcl::VoxelGrid<pcl::PointXYZ> voxel_filter; // 복셀 필터 객체 생성
voxel_filter.setInputCloud(scanCloud); // 스캔 데이터 입력
voxel_filter.setLeafSize(0.5f, 0.5f, 0.5f);  // TODO: 각 복셀의 크기를 0.5m로 설정
voxel_filter.filter(*cloudFiltered);  // TODO: 필터링된 데이터를 cloudFiltered에 저장
```

Transfer-2: Process scanCloud and return it in cloudFiltered

##Accurate Scan Matching Technique:

The current location is estimated by comparing the lidar data with Map.pcd.

NDT and ICP are used to find the optimal transformation between the two scans (lidar scan and map data).

NDT (Normal Distributions Transform):

It models the map data as a probability density function to provide fast and stable matching.

Setting the appropriate resolution (ndt.setResolution) is important.

ICP (Iterative Closest Point):

Iteratively calculates the proximity between points to find the transformation.

If the data is not well aligned, it is likely to not converge.

This task used NDT

```
//TODO: NDT를 사용하여 변환 행렬 계산

 pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;

ndt.setResolution(1.0);           // TODO: NDT 해상도 설정

ndt.setInputSource(cloudFiltered);      // TODO: 필터링된 소스 점 구름 설정

ndt.setInputTarget(mapCloud);           // TODO: 대상 점 구름(mapCloud) 설정

pcl::PointCloud<pcl::PointXYZ>::Ptr output(new pcl::PointCloud<pcl::PointXYZ>());

// 계산 결과를 저장할 포인터

ndt.align(*output);      // TODO: NDT 정합 수행

transform = ndt.getFinalTransformation();      // TODO: 최종 변환 행렬 계산
```

Trasfer-3: Process what is passed to cloudFiltered and return it by putting it in transform on the last line.

##Scan Transformation and Rendering:

The lidar data needs to be transformed to fit the map using the matching transformation matrix.

The transformed data is then appropriately rendered to visually compare the estimated and actual positions of the vehicle.

```
// TODO: Transform scan so it aligns with ego's actual pose and render that scan

// TODO: (스캔 데이터를 차량의 실제 위치에 맞게 변환)

//pcl::transformPointCloud(*cloudFiltered, *scanCloud, transform);


PointCloudT::Ptr transformedScan(new PointCloudT); // 변환된 스캔 데이터를 저장할 변수
```

pcl::transformPointCloud(*cloudFiltered, *transformedScan, transform); // TODO: 스캔 데이터를 변환 행렬로 변환

viewer->removePointCloud("scan"); // 기존의 스캔 데이터 제거

// TODO: Change `scanCloud` below to your transformed scan

//renderPointCloud(viewer, scanCloud, "scan", Color(1,0,0)); //scanCloud는 필터링된 원본 데이터를 표현하는 데 사용

renderPointCloud(viewer, transformedScan, "scan", Color(1, 0, 0));

//transformedScan은 변환된 데이터를 렌더링하기 때문에, 스캔 데이터와 차량의 실제 위치를 맞추는 데 적합

// TODO: 변환된 스캔 데이터를 렌더링

Transfer-4: Transform cloudFiltered data using the calculated transformation matrix transform, and save the result in transforemdScan. The data saved in Transformedscan is finally rendered on the screen through renderPointCloud.

#3. Speed Control:

If the vehicle speed is too fast, the lidar data may not be collected enough, making accurate matching difficult. Use the arrow keys appropriately to maintain a medium speed!

4. Core Tasks:

The core of the project is to determine the vehicle's position based on the relative match between the static map stored in map.pcd and the lidar data, not based on the position on the Earth such as GPS.

Therefore, this project is:

1.Refining the lidar data.

2.Comparing with map.pcd (scan matching).

3.Calculating the optimal transformation (NDT or ICP).

4.Maintaining the stability of the system so that the vehicle can continuously estimate its position while driving.

Through this process, the goal is to minimize the error between the actual position and the estimated position.

5. Data Flow:

Transfer-1: Copy `pclCloud` to `scanCloud` and transfer.

Transfer-2: Filter `scanCloud` and store it in `cloudFiltered`.

Transfer-3: Use `cloudFiltered` for NDT matching to calculate the transformation matrix `transform`.

Transfer-4: Transform `cloudFiltered` using the calculated transformation matrix `transform` and store the result in `transformedScan`.

Final: The data stored in `transformedScan` is finally rendered on the screen through `renderPointCloud`.

###cloudFiltered** is used twice, as input data for matching and input data for transformation.

#6. Execution Environment:

This project uses Udacity's Ubuntu Cloud environment.

First console window:

First, you need to do git clone in Ubuntu environment.

Git clone https://github.com/udacity/nd0013_cd2693_Exercise_Starter_Code

Check the cloned directory and move to the path below.

After checking the cloned directory, go to the path below

cd nd0013_cd2693_Exercise_Starter_Code/Lesson_7_Project_Scan_Matching_Localization/c3-project

Check if the following files exist using the ls command

├── **CMakeLists.txt**

├── **README.md**

├── **c3-main.cpp**

├── **helper.cpp**

├── **helper.h**

├── **make-libcarla-install.sh**

├── **map.pcd**

├── **map_loop.pcd**

├── **rpclib**

└── **run_carla.sh**

Now run the following command

chmod +x make-libcarla-install.sh

Then run the following command

./make-libcarla-install.sh

Update the code in the C3-main.cpp file.

Now run the following command

‘cmake .

Once it’s done, run the following command

make

Second console window:

To use Scan Matching Localisation, the Carla simulator must be running in the background. Open a new terminal window and type the following:

This command will run the Carla simulator

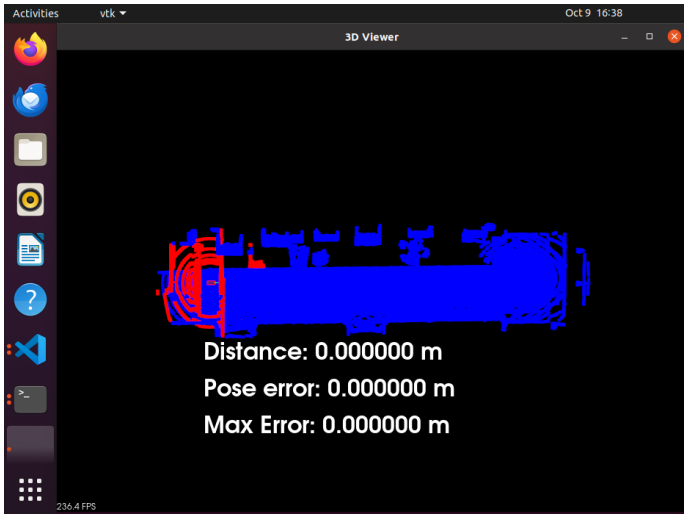
./run_carla.sh

Third console window:

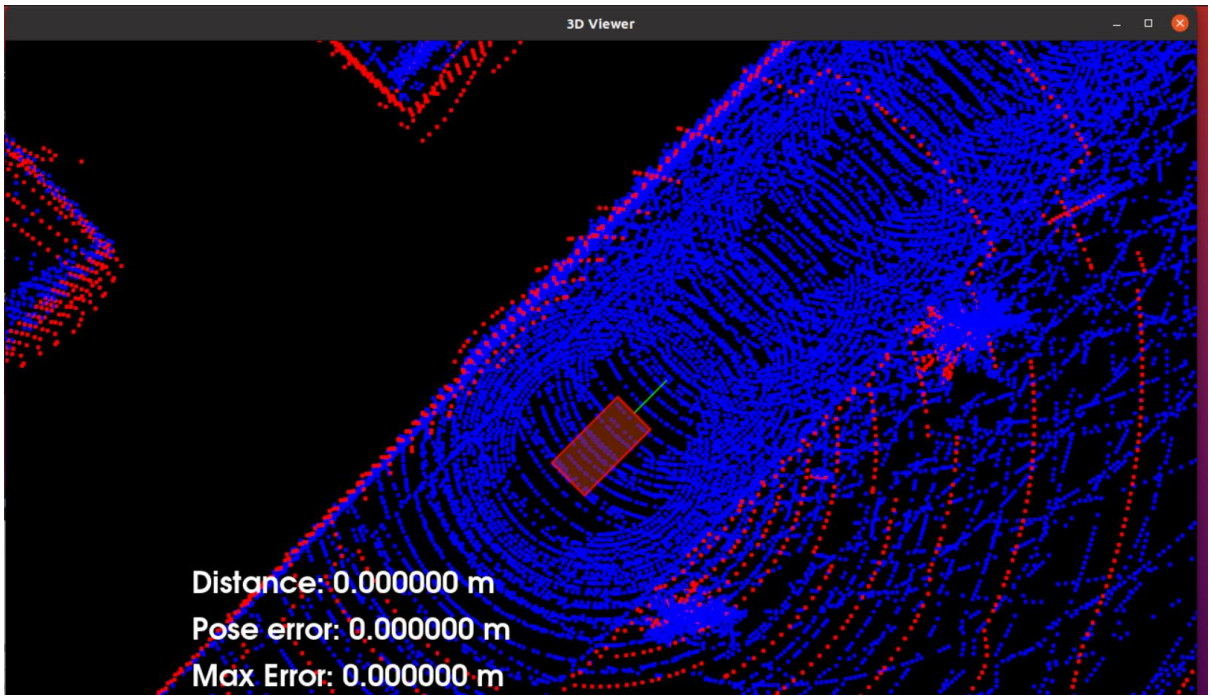
Open another terminal window. Then type the following command:

./cloud_loc

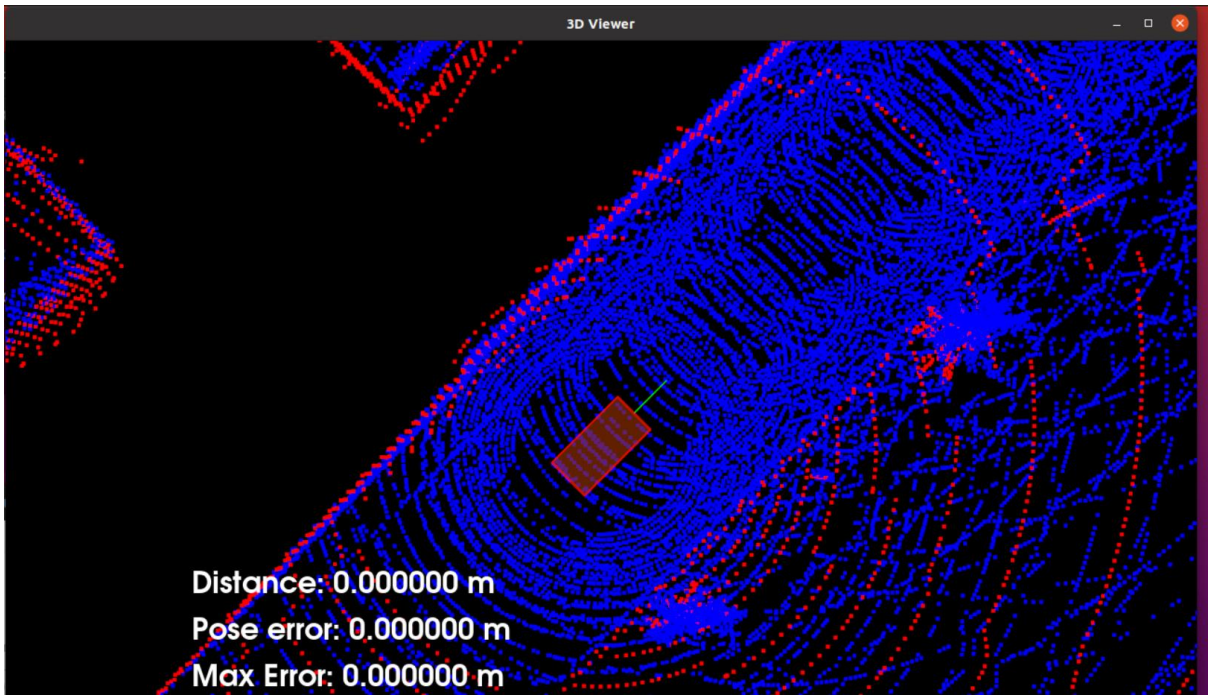
The Carla simulator will now start running.



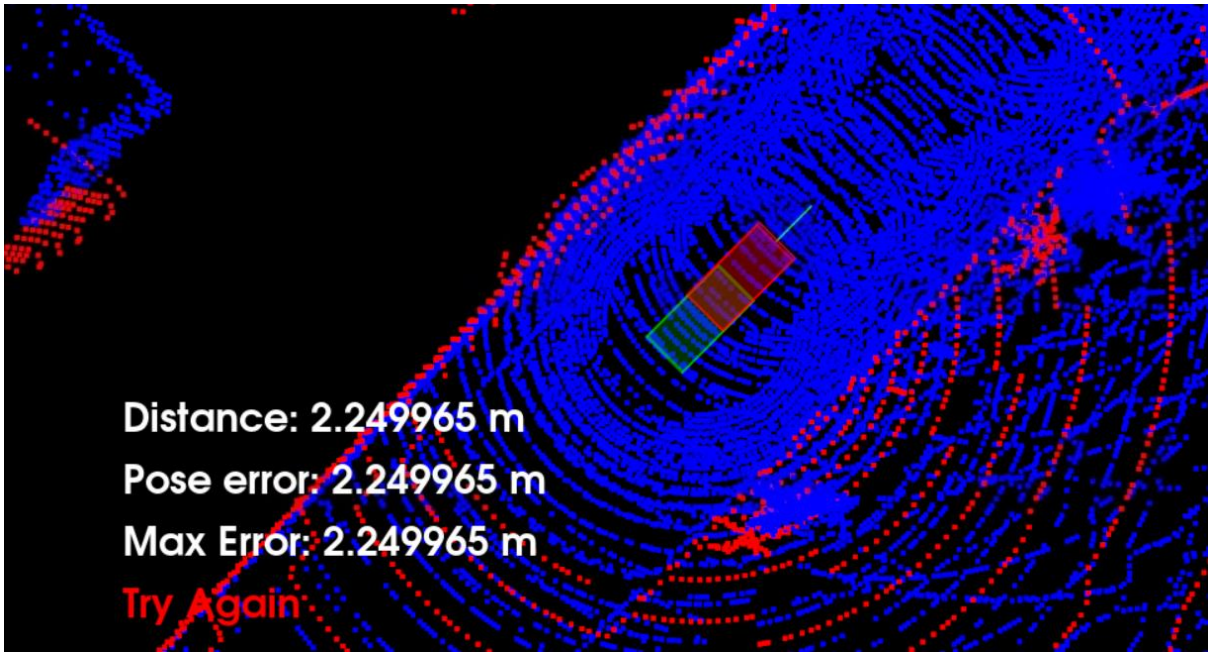
#7. Execution result:



Press the a key to set the top view



Press the arrow (up) key 3 times to start driving at medium speed



Currently, the simulator has been upgraded to a new version, and there is a problem that it does not match well with the blue map, so the result is not good as above

If the simulator problem is resolved in the future, I will update it again.

(Korean Version)

#1.프로젝트 목표 :

현지화(Localization)은 자율주행 차량이 자신의 현재위치와 자세(Pose:위치와 방향)을 정확히 파악하는 과정을 말합니다. 즉, 차량은 자신의 주변환경(맵데이터,map.pcd파일)을 기반으로 자신의 위치를 지속적으로 업데이트 해야 합니다.

Lidar : 센서를 통해 차량 주변의 포인트 클라우드 데이터를 수집 (carla시뮬레이터가 제공)

Map.pcd : 이 파일에 저장된 정적인 포인트 클라우드 맵(정적인 지도 데이터)과 수집한 라이다 데이터를 비교하여 차량의 위치 (포즈)를 계산

1.포즈 오차 1.2m 이하 유지: 차량이 주행하는 동안 현재 위치와 계산된 위치 간의 오차가 1.2m 를 넘지 않아야 합니다.

2.170m 이상 주행: 차량을 중속으로 최소 170m 주행해야 합니다.

3.현지화 알고리즘 구현: Lidar 데이터를 필터링(복셀 필터), 매칭(ICP 또는 NDT)하여 차량의 위치를 계산하는 코드를 작성해야 합니다.

학생은 c3-main.cpp파일에서 todo부분을 작성하면 됩니다

#2. C3-main.cpp :

##라이다 데이터 수신:

Carla시뮬레이터에서 생성되는 라이다 데이터를 하기의 코드에서 수신합니다

```
lidar->Listen([&new_scan, &lastScanTime, &scanCloud](auto data){
    auto scan = boost::static_pointer_cast<csd::LidarMeasurement>(data);
    for (auto detection : *scan){
        pclCloud.points.push_back(PointT(detection.x, detection.y, detection.z));
    }
    *scanCloud = pclCloud;
});
```

라이다 데이터는 pclCloud라는 변수에 저장되며, scanCloud로 복사되어 뒤에 사용됩니다.

전달-1 : pclcloud를 scanCloud에 담아서 return

##라이다 데이터의 품질향상:

복셀필터(voxel filter)를 사용해 불필요한 노이즈를 제거, 필터링된 포인트를 사용해 계산수행

```
// TODO: (Filter scan using voxel filter)
// TODO: (복셀 필터를 사용하여 스캔 데이터 필터링)
 pcl::VoxelGrid<pcl::PointXYZ> voxel_filter; // 복셀 필터 객체 생성
voxel_filter.setInputCloud(scanCloud); // 스캔 데이터 입력
```

voxel_filter.setLeafSize(0.5f, 0.5f, 0.5f); // TODO: 각 복셀의 크기를 0.5m로 설정

voxel_filter.filter(*cloudFiltered); // TODO: 필터링된 데이터를 cloudFiltered에 저장

```
new_scan = true;
// TODO: (Filter scan using voxel filter)
// TODO: (복셀 필터를 사용하여 스캔 데이터 필터링)
pcl::VoxelGrid<pcl::PointXYZ> voxel_filter; // 복셀 필터 객체 생성
voxel_filter.setInputCloud(scanCloud); // 스캔 데이터 입력
voxel_filter.setLeafSize(0.5f, 0.5f, 0.5f);  // TODO: 각 복셀의 크기를 0.5m로 설정
voxel_filter.filter(*cloudFiltered);  // TODO: 필터링된 데이터를 cloudFiltered에 저장

// TODO: Find pose transform by using ICP or NDT matching
Eigen::Matrix4f transform; // 변환 행렬을 저장할 변수
```

전달-2 : scanCloud를 처리하여 cloudFiltered에 담아서 return

##정확한 스캔매칭기법:

라이다 데이터와 Map.pcd간의 비교를 통해서 현재위치를 추정합니다.

NDT와 ICP는 두 스캔(라이다 스캔과 맵 데이터) 간의 최적 변환을 찾는 데 사용됩니다.

NDT (Normal Distributions Transform):

맵 데이터를 확률 밀도 함수로 모델링하여 빠르고 안정적인 매칭을 제공합니다.

적절한 해상도(ndt.setResolution) 설정이 중요합니다.

ICP (Iterative Closest Point):

포인트 간의 근접성을 반복적으로 계산하여 변환을 찾습니다.

데이터가 잘 정렬되지 않으면 수렴하지 않을 가능성이 있습니다.

이 과제에서는 NDT를 이용했습니다

//TODO: NDT를 사용하여 변환 행렬 계산

pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;

ndt.setResolution(1.0); // TODO: NDT 해상도 설정

ndt.setInputSource(cloudFiltered); // TODO: 필터링된 소스 점 구름 설정

ndt.setInputTarget(mapCloud); // TODO: 대상 점 구름(mapCloud) 설정

pcl::PointCloud<pcl::PointXYZ>::Ptr output(new pcl::PointCloud<pcl::PointXYZ>());

// 계산 결과를 저장할 포인터

ndt.align(*output); // TODO: NDT 정합 수행

transform = ndt.getFinalTransformation(); // TODO: 최종 변환 행렬 계산

```
// TODO: Find pose transform by using ICP or NDT matching
Eigen::Matrix4f transform; // 변환 행렬을 저장할 변수

//pose = ....
// TODO: (NDT 또는 ICP를 사용하여 변환 행렬 계산)
//TODO: NDT를 사용하여 변환 행렬 계산
pcl::NormalDistributionsTransform<pcl::PointXYZ, pcl::PointXYZ> ndt;
ndt.setResolution(1.0);                    // TODO: NDT 해상도 설정
ndt.setInputSource(cloudFiltered);         // TODO: 필터링된 소스 점 구름 설정
ndt.setInputTarget(mapCloud);              // TODO: 대상 점 구름(mapCloud) 설정
pcl::PointCloud<pcl::PointXYZ>::Ptr output(new pcl::PointCloud<pcl::PointXYZ>()); // 계산 결과를 저장할 포인터
ndt.align(*output);                        // TODO: NDT 정합 수행
transform = ndt.getFinalTransformation();  // TODO: 최종 변환 행렬 계산
```

전달-3 : cloudFiltered로 전달받은 것을 처리하여 마지막줄에 transform에 담아서 return

##스캔변환 및 렌더링:

매칭된 변환 행렬을 사용해 라이다 데이터를 맵에 맞춰 변환해야 합니다.

변환된 데이터를 적절히 렌더링하여 차량의 추정 위치와 실제 위치를 시각적으로 비교합니다.

// TODO: Transform scan so it aligns with ego's actual pose and render that scan

// TODO: (스캔 데이터를 차량의 실제 위치에 맞게 변환)

//pcl::transformPointCloud(*cloudFiltered, *scanCloud, transform);

PointCloudT::Ptr transformedScan(new PointCloudT); // 변환된 스캔 데이터를 저장할 변수

pcl::transformPointCloud(*cloudFiltered, *transformedScan, transform); // TODO: 스캔 데이터를 변환 행렬로 변환

viewer->removePointCloud("scan"); // 기존의 스캔 데이터 제거

// TODO: Change `scanCloud` below to your transformed scan

//renderPointCloud(viewer, scanCloud, "scan", Color(1,0,0)); //scanCloud는 필터링된 원본 데이터를 표현하는 데 사용

renderPointCloud(viewer, transformedScan, "scan", Color(1, 0, 0));

//transformedScan은 변환된 데이터를 렌더링하기 때문에, 스캔 데이터와 차량의 실제 위치를 맞추는 데 적합

// TODO: 변환된 스캔 데이터를 렌더링

```
// TODO: Transform scan so it aligns with ego's actual pose and render that scan
// TODO: (스캔 데이터를 차량의 실제 위치에 맞게 변환)
//pcl::transformPointCloud(*cloudFiltered, *scanCloud, transform);

PointCloudT::Ptr transformedScan(new PointCloudT); // 변환된 스캔 데이터를 저장할 변수
pcl::transformPointCloud(*cloudFiltered, *transformedScan, transform); // TODO: 스캔 데이터를 변환 행렬로 변환

viewer->removePointCloud("scan"); // 기존의 스캔 데이터 제거

// TODO: Change `scanCloud` below to your transformed scan
//renderPointCloud(viewer, scanCloud, "scan", Color(1,0,0) ); //scanCloud는 필터링된 원본 데이터를 표현하는 데 사용
renderPointCloud(viewer, transformedScan, "scan", Color(1, 0, 0));
//transformedScan은 변환된 데이터를 렌더링하기 때문에, 스캔 데이터와 차량의 실제 위치를 맞추는 데 적합
// TODO: 변환된 스캔 데이터를 렌더링
```

전달-4: 계산된 변환행렬 transform을 이용하여 cloudFiltered 데이터를 변환하고, 결과를 transforemdScan에 저장

Transformedscan에 저장된 데이터는 최종적으로 renderPointCloud를 통해 화면에 랜더링 됨.

#3. 속도 조절 :

차량 속도가 너무 빠르면 라이다 데이터가 충분히 수집되지 않아 정확한 매칭이 어려워질 수 있습니다. 화살표 키를 적절히 사용해 중속을 유지!

4. 핵심 과제:

GPS와 같은 지구에서의 위치를 기준으로 한 것이 아니고, map.pcd에 저장된 정적인 맵과 라이다 데이터 간의 상대적인 일치를 기반으로 차량의 위치를 파악하는 것이 프로젝트의 핵심입니다.

따라서 이 프로젝트는:

1.라이다 데이터의 정제.

2.map.pcd와의 비교(스캔 매칭).

3.최적의 변환 계산(NDT 또는 ICP).

4.차량이 주행하면서도 지속적으로 위치를 추정할 수 있도록 시스템 안정성 유지.

이 과정을 통해 실제 위치와 추정 위치 간의 오차를 최소화하는 것이 목표입니다.

5. 데이터의 흐름:

전달-1 : `pclCloud`를 `scanCloud`에 복사하여 전달.

전달-2 : `scanCloud`를 필터링하여 `cloudFiltered`에 저장.

전달-3 : `cloudFiltered`를 NDT 매칭에 사용하여 변환 행렬 `transform` 계산.

전달-4 : 계산된 변환 행렬 `transform`을 이용하여 `cloudFiltered`를 변환하고, 결과를 `transformedScan`에 저장.

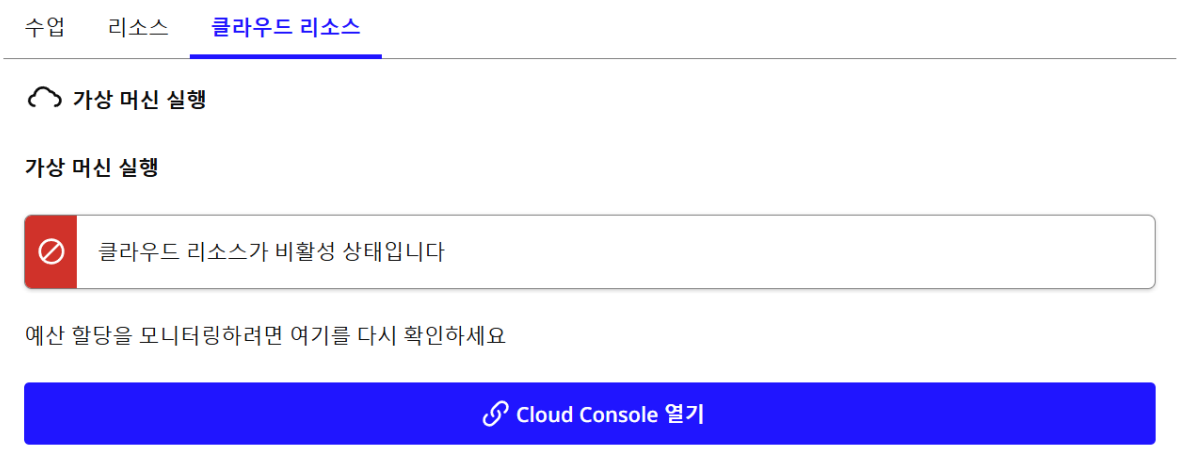
최종 : `transformedScan`에 저장된 데이터는 최종적으로 `renderPointCloud`를 통해 화면에 렌더링됨.

###cloudFiltered**는 **매칭의 입력 데이터**와 **변환의 입력 데이터**로 두 번 사용됨.

#6. 실행 환경 :

이 프로젝트는 Udacity의 우분투 클라우드 환경을 사용합니다.

프로젝트 워크스페이스



첫번째 콘솔 창 :

먼저 우분투 환경에서는 git clone을 진행해야 합니다.

Git clone https://github.com/udacity/nd0013_cd2693_Exercise_Starter_Code

Clone된 디렉토리를 확인 후 아래의 경로로 이동합니다

```
cd nd0013_cd2693_Exercise_Starter_Code/Lesson_7_Project_Scan_Matching_Localization/c3-project
```

ls명령어로 하기의 파일들이 있는지 확인합니다

```
.
├─── CMakeLists.txt
├─── README.md
├─── c3-main.cpp
├─── helper.cpp
├─── helper.h
├─── make-libcarla-install.sh
├─── map.pcd
├─── map_loop.pcd
├─── rpclib
└─── run_carla.sh
```

이제 아래의 명령어를 실행합니다

```
chmod +x make-libcarla-install.sh
```

그 다음 아래의 명령어를 실행합니다

```
./make-libcarla-install.sh
```

C3-main.cpp파일의 코드를 Update해줍니다.

프롬프트에서 nano c3-main.cpp를 입력하여 nano에디터에서 작업합니다.

Nano 에디터에서는 Ctrl key + shift Key + K key를 눌러서 한 행씩 삭제 후 paste 해줍니다.

Paste도 code전체는 한번에 안되기에 나누어서 하셔야 합니다.

완료시 crtl+o를 눌러서 저장하고 enter눌러서 나옵니다.

이제 아래의 명령어를 실행합니다

```
cmake .
```

실행이 완료되면 아래의 명령어도 실행해 줍니다

```
make
```

두번째 콘솔 창 :

Scan Matching Localisation을 사용하려면 Carla 시뮬레이터가 백그라운드에서 실행되어야 합니다.

새로운 터미널 창을 실행하고 아래와 같이 입력합니다

이 명령어는 Carla시뮬레이터를 실행하는 명령어입니다

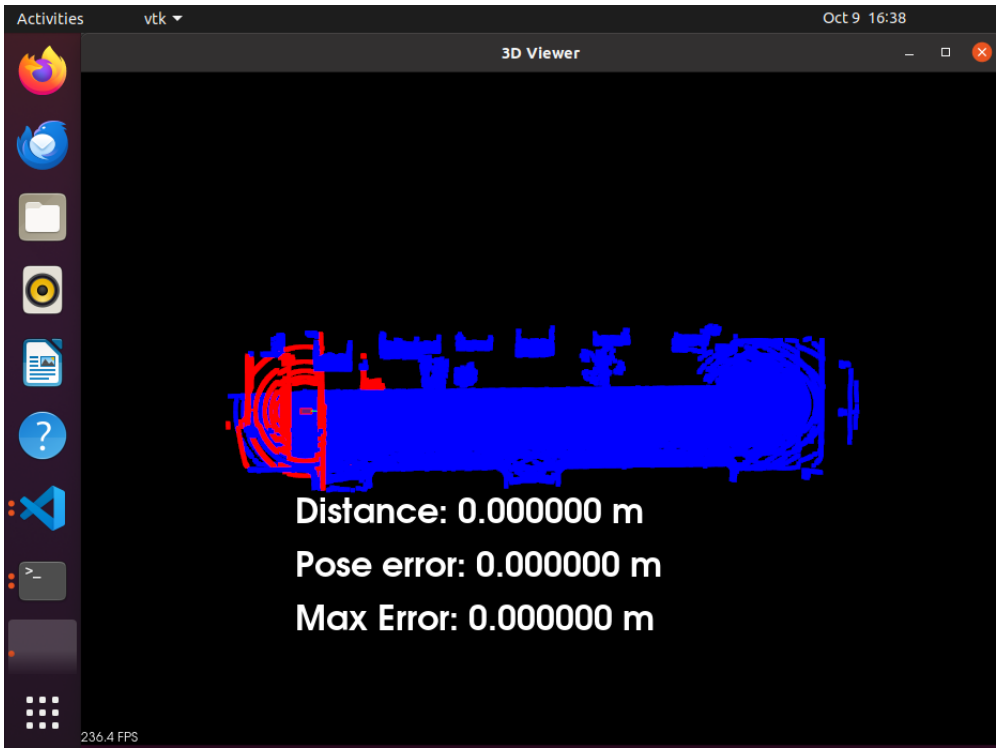
```
./run_carla.sh
```

세번째 콘솔 창 :

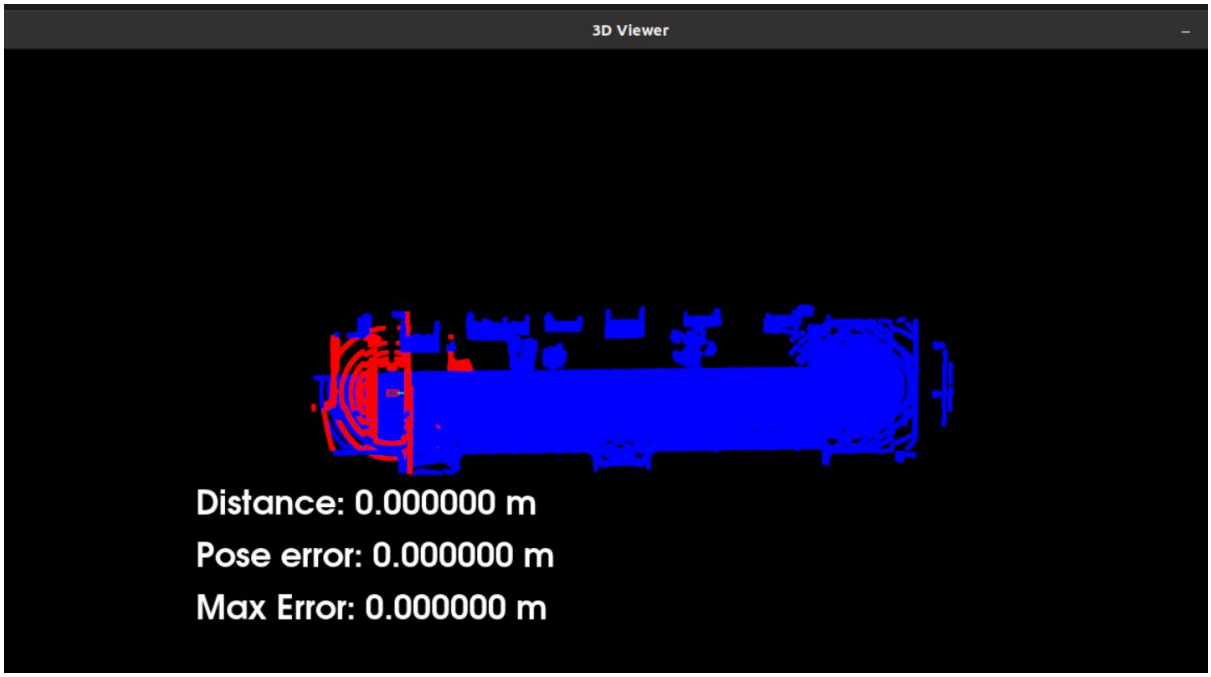
터미널창을 하나 더 열어줍니다. 그리고 아래와 같은 명령어를 입력합니다

```
./cloud_loc
```

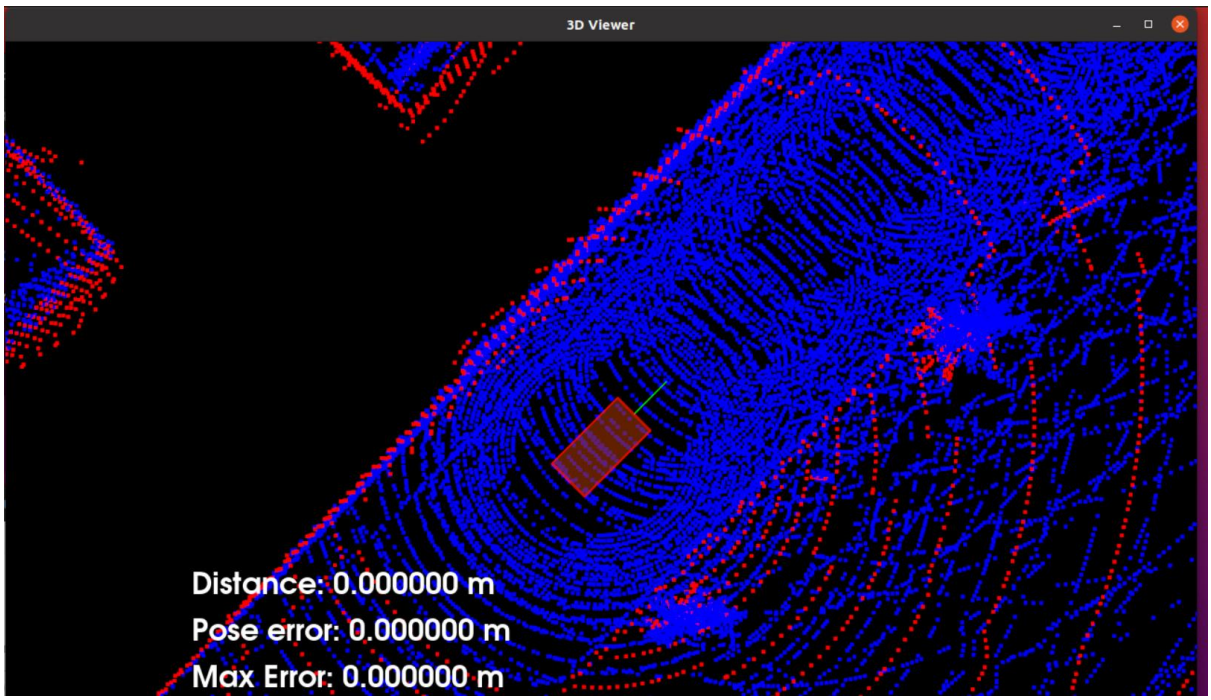
이제 Carla 시뮬레이터가 실행되기 시작합니다



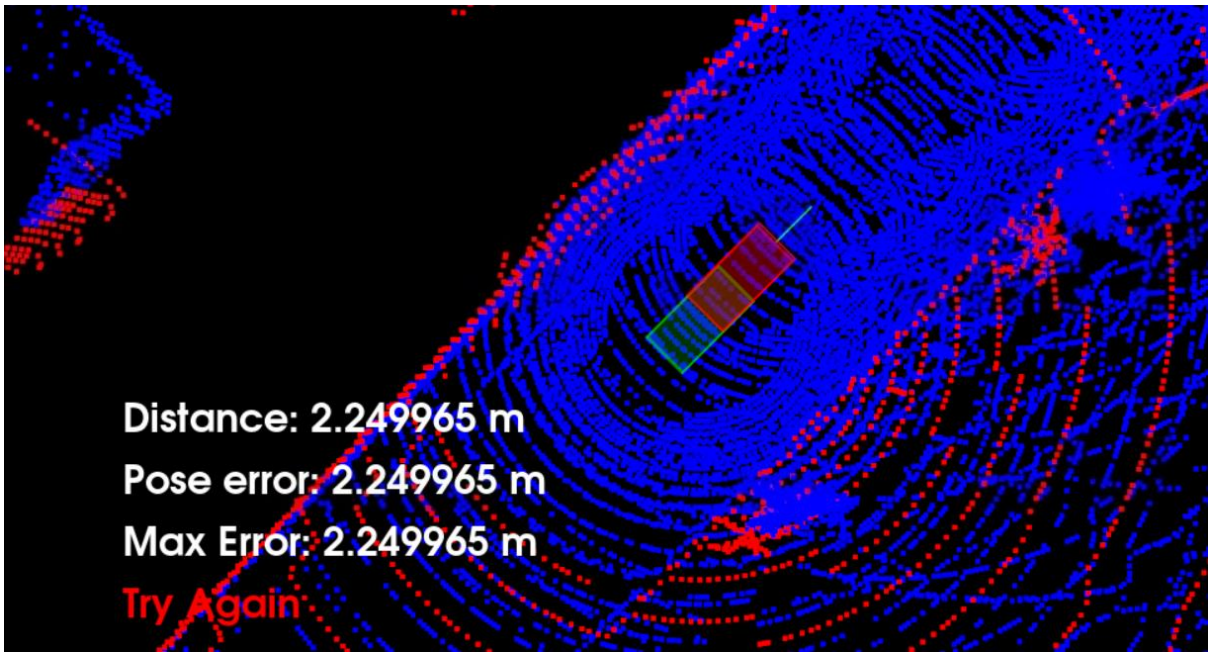
#7. 실행 결과 :



a키를 눌러서 top view 설정



화살표(up)키를 3번 눌러서 중간속도로 주행시작



현재 시뮬레이터가 새버전으로 업그레이드 되면서 파란색맵과는 잘 맞지 않는 문제가 발생하여 위와 같이 결과가 좋지 않습니다

추후 시뮬레이터 문제가 해결된다면, 다시 업데이트 하도록 하겠습니다