광운대 Lidar & ROS

Lidar & ROS

WeGo

목차

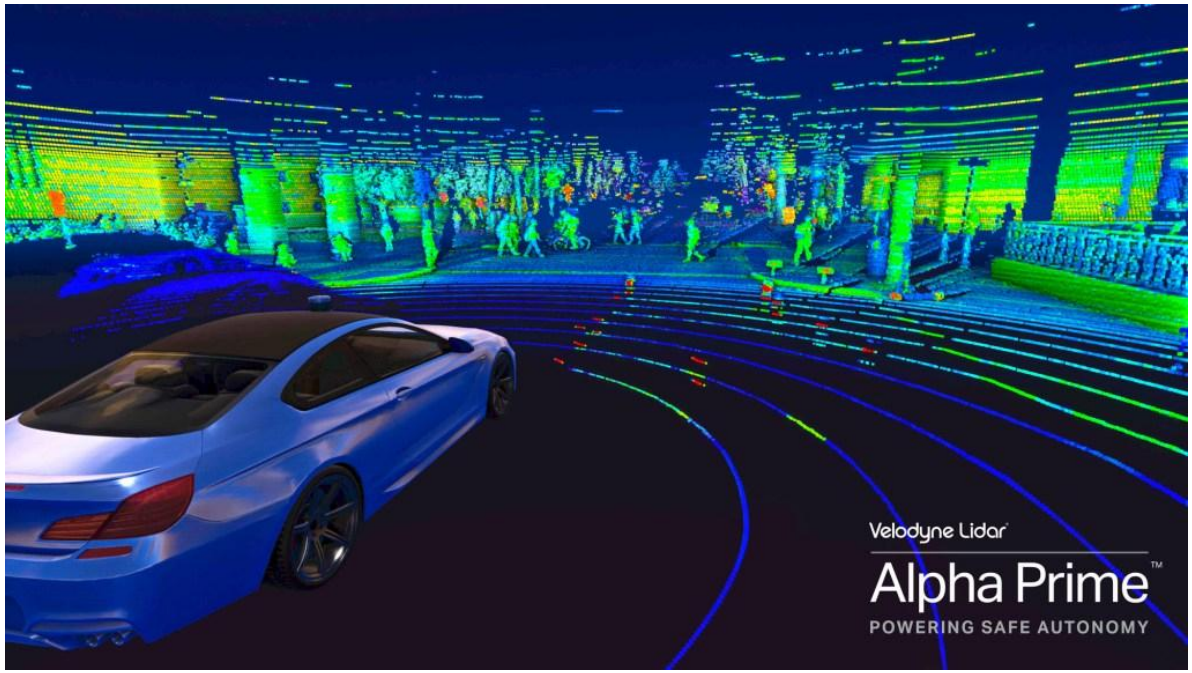
WeGo

01

자율주행 Lidar 소개

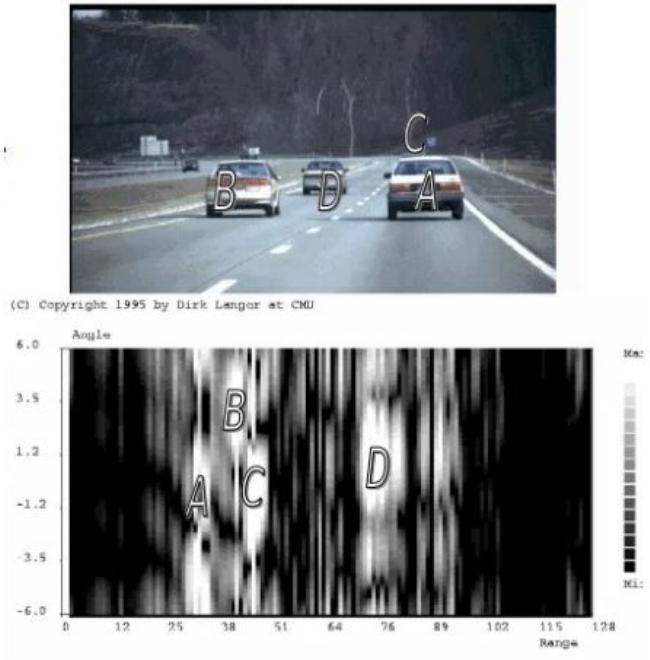
LiDAR

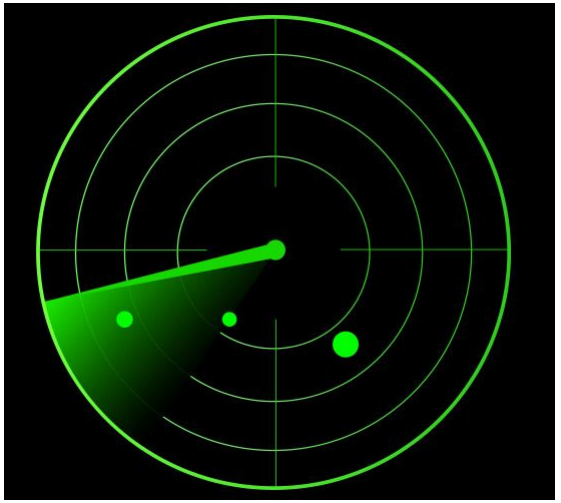
- Lidar는 레이저를 이용하여, 목표 물체까지의 거리, 방향, 반사율 등을 측정할 수 있는 센서를 의미한다.
- Radar나 Camera에 비해, 물체까지의 거리와 방향에 대한 정보가 매우 우수
- 반면, 날씨에 대한 의존도가 높으며 눈, 비가 오는 날씨에는 사용하기가 힘듦



LiDAR & Radar

- Radar는 전자파 기반의 센서
- 물체의 형상을 확실하게 인식 불가(여기쯤 있겠구나…)
- 기후에 상관없이 항상 제 성능을 발휘
- 사용 사례: 자동차긴급제동장치, 스마트 크루즈 컨트롤, 전방충돌 방지보조 등

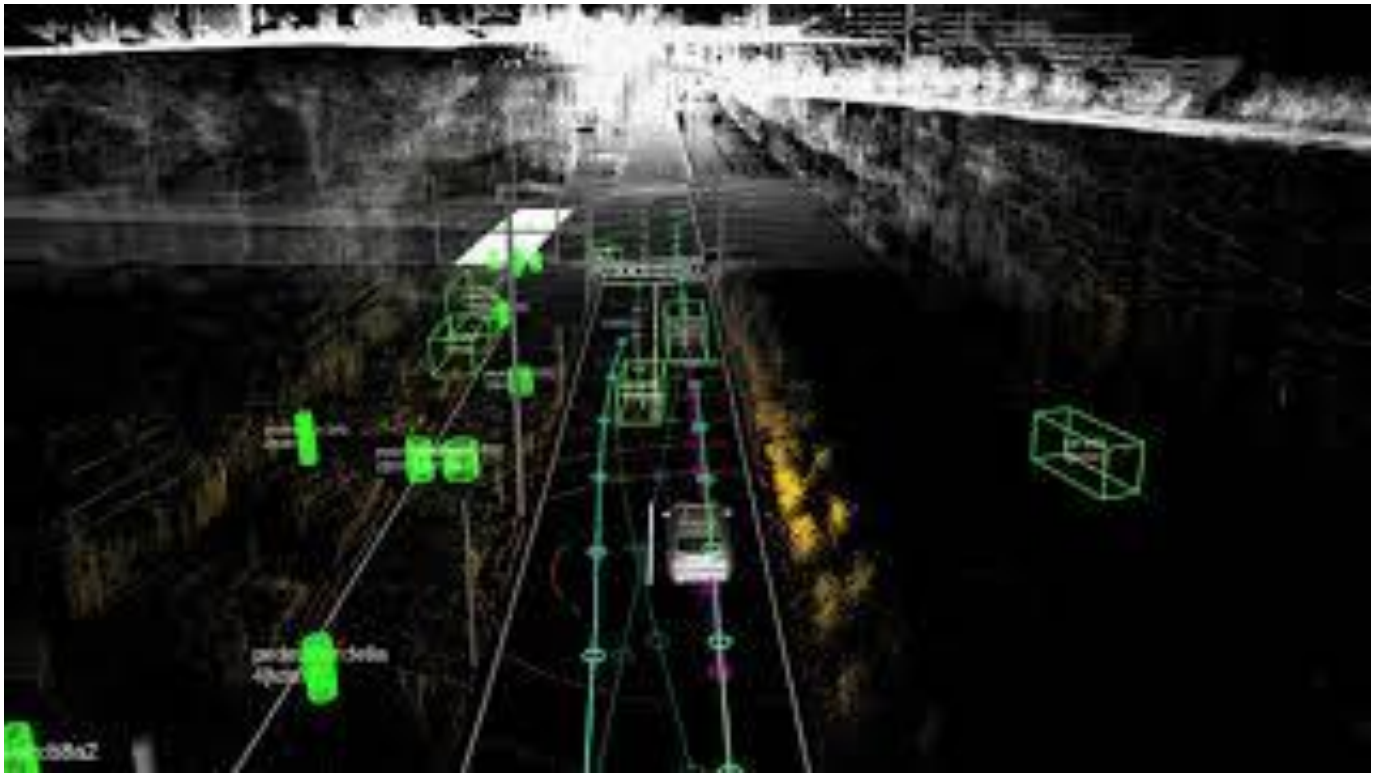




LiDAR & Radar

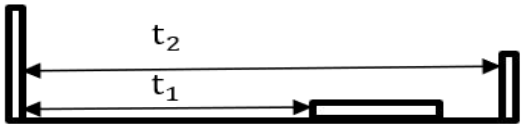
구분	발사체	물체 정확도	기후 영향
Radar	전자파	근사치	없음
Lidar	레이저(광선)	거의 정확	있음

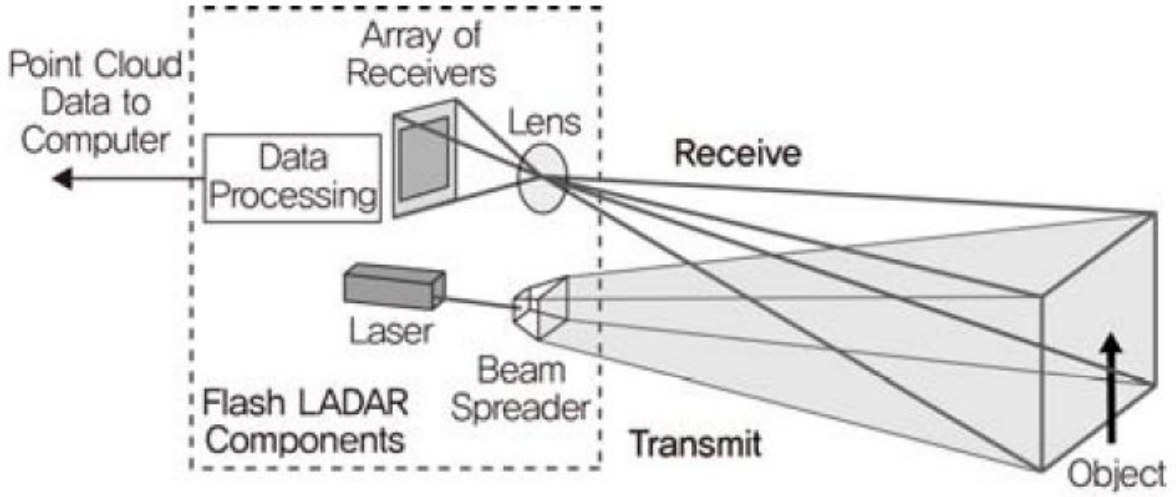
LiDAR 실차 적용 예시



• 작동원리

펄스폭이 매우 짧은 레이저광을 표적으로 보내고, 표적 표면에서 반사되어 되돌아 올 때 까지 시간을 측정하여 표적거리를 이 시간 값과 빛의 속도로부터 산출하는 방법이다. 일반적으로 출력되는 형태는 PointCloud 형태로 출력되며, 다수의 점들이 모여있는 형태이다.





WeGo

02

ROS Lidar Data Message

WeGo

ROS Lidar Message 종류

- sensor_msgs/LaserScan.msg

  일반적으로 2D Lidar에서 많이 사용하는 Message

- sensor_msgs/PointCloud.msg

  (Point = 좌표, Cloud = 구름) => Scan Data의 집합

  Lidar Data는 수많은 Point를 다뤄야 하는데 이를 위해 나온 라이브러리가

  PCL(Point Cloud Library)이며 ROS에서는 sensor_msgs에 정의되어 있습니다.

- sensor_msgs/PointCloud2.msg

  PointCloud.msg는 2D Lidar 까지는 무리없으나 3D Lidar와 같이 방대한

  데이터를 전송하는데 느린 단점이 있습니다. 그래서 이를 해결하고자 ROS에서

  새롭게 제공한 템플릿이 PointCloud2.msg 입니다.

ROS sensor_msgs/LaserScan

- Header □ LaserScan Data의 내부 정보를 포함

  uint32 seq □ 수신된 Data가 몇 번째 데이터인지 나타내는 정보

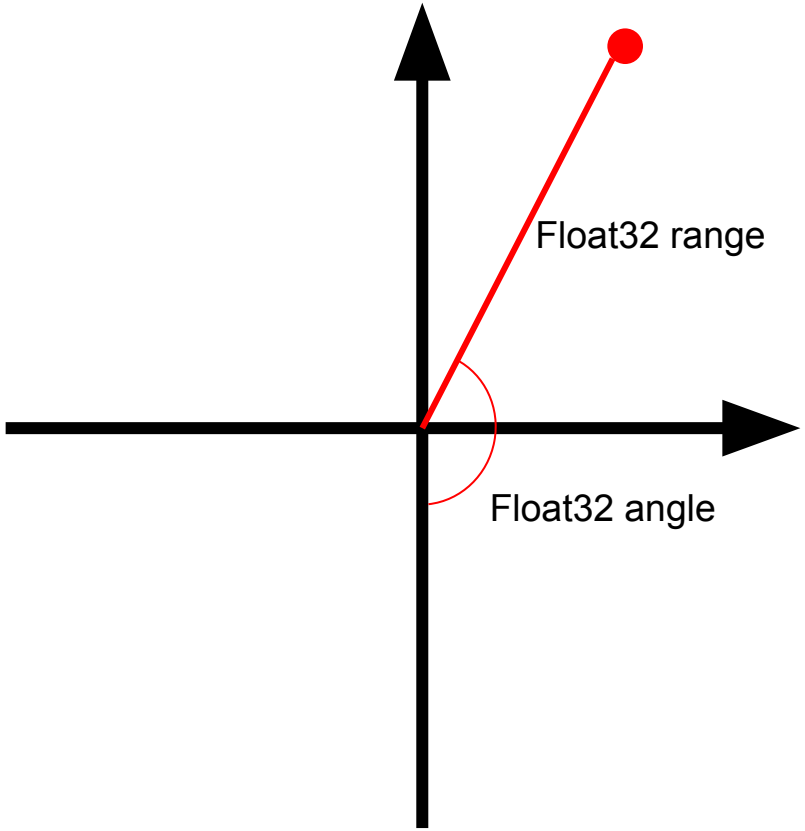
  time stamp □ 수신된 Data의 ROS Time

  string frame_id □ 수신된 Data의 Frame ID

- float32 angle_min □ Lidar 센서의 최소 측정 각도
- float32 angle_max □ Lidar 센서의 최대 측정 각도
- float32 angle_increment □ Lidar 센서의 측정 분해능
- float32 scan_time □ Scan 사이의 시간
- float32 range_min □ Lidar 센서의 최소 측정 거리
- float32 range_max □ Lidar 센서의 최대 측정 거리
- float32[ ] ranges □ 실제 측정된 거리 Data
- float32[ ] intensities □ 실제 측정된 Intensity Data

ROS sensor_msgs/LaserScan

- float32 angle_min ☐ Lidar 센서의 최소 측정 각도
- float32 angle_max ☐ Lidar 센서의 최대 측정 각도
- float32 angle_increment ☐ Lidar 센서의 측정 분해능
- float32 range_min ☐ Lidar 센서의 최소 측정 거리
- float32 range_max ☐ Lidar 센서의 최대 측정 거리
- float32[ ] ranges ☐ 실제 측정된 거리 Data
- float32[ ] intensities ☐ 실제 측정된 Intensity Data



실습 예제 2.1:

가상 LaserScan 데이터를 만들어 Publish 해보기(360개)

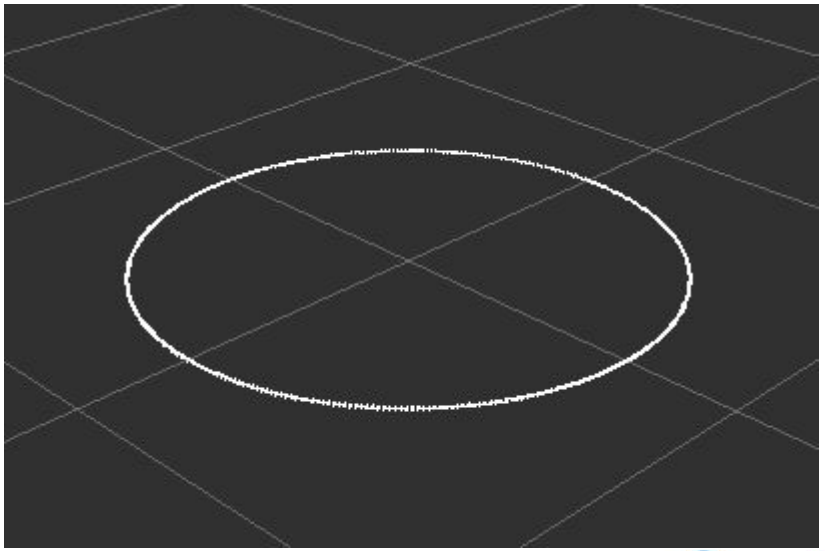
angle_min = −3.14, angle_max = 3.14

angle_increment = 6.28 / 360.0

time_increment = 0.1

ranges = 자유

```python
import rospy
import math
from sensor_msgs.msg import LaserScan

class pcl_processor:
    def __init__(self):…

    def make_samples(self):…


if __name__ == "__main__":…
```



WeGo

```
import rospy
import math
from sensor_msgs.msg import LaserScan

import numpy as np
import random

class pcl_processor:
    def __init__(self):
        self.pub_sample = rospy.Publisher("/sample_scan", LaserScan, queue_size=5)
    def make_samples(self):
        ls = LaserScan()
        ls.header.stamp = rospy.Time.now()
        ls.header.frame_id = "laser"
if __name__ == "__main__":
    rospy.init_node("sample_laserscan", anonymous=False)
    pp = pcl_processor()
    rate = rospy.Rate(15)
    while not rospy.is_shutdown():
        pp.make_samples()
        rate.sleep()
```

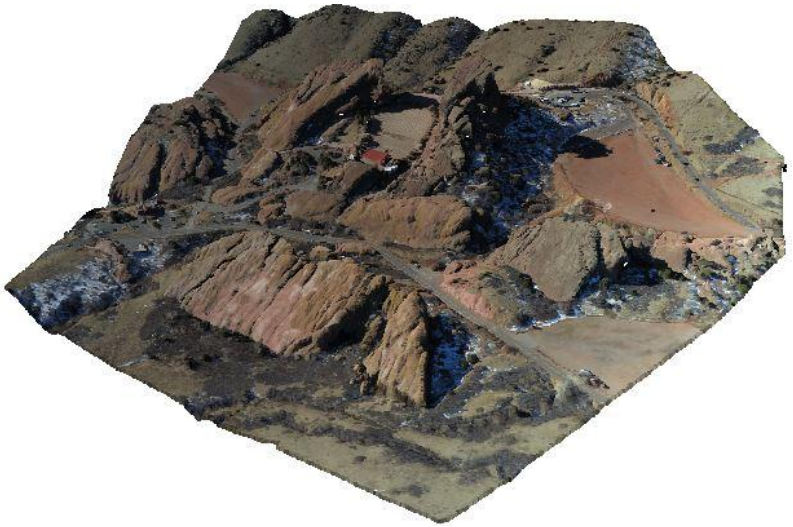
03

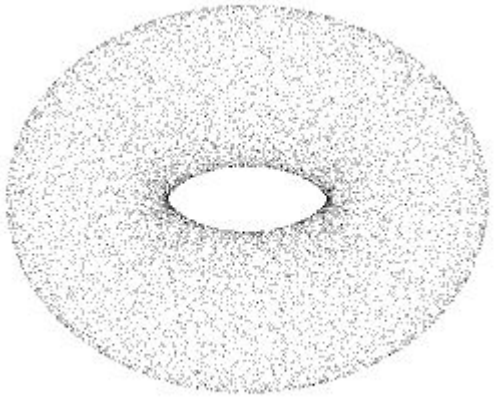
PointCloud & Marker 실습

WeGo

- PointCloud：

  Lidar나 3D 스캐너를 통해 얻을 수 있는 데이터로 하나의 데이터인 점(Point)는 3차원 좌표계의 X,Y,Z로 정의됩니다. 이러한 점들의 집합(Cloud)을 Point Cloud라고 합니다.





ROS sensor_msgs/PointCloud.msg

- Header □ LaserScan.msg와 동일

  uint32 seq □ 수신된 Data가 몇 번째 데이터인지 나타내는 정보

  time stamp □ 수신된 Data의 ROS Time

  string frame_id □ 수신된 Data의 Frame ID

- geometry_msgs/Point32[ ] points -> Point Data

  float32 x

  float32 y

  float32 z

- sensor_msgs/ChannelFloat32.msg -> 다 채널 라이다 사용 시 channel정보 (name, value 등)

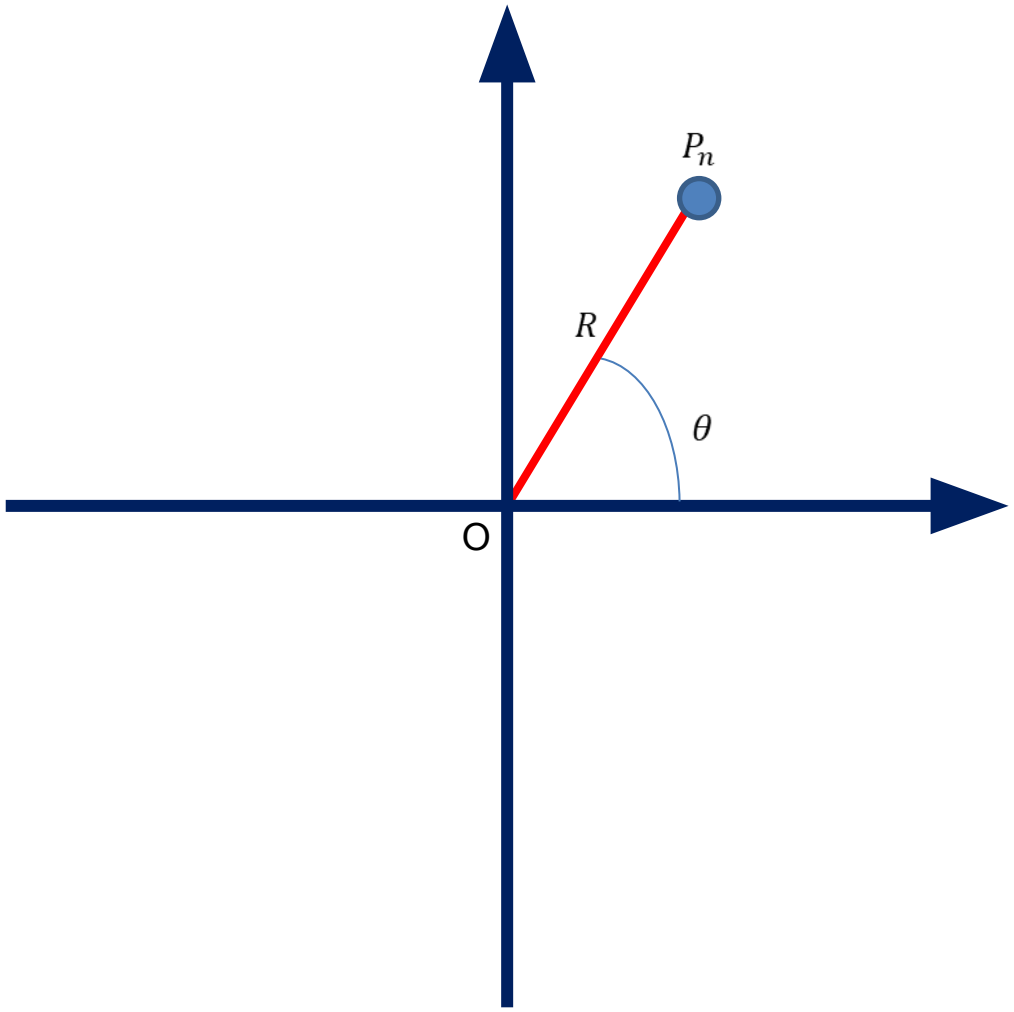
• LaserScan -> PointCloud

LaserScan에서는 앞서 살펴봤듯 각도와 거리 데이터가 주어집니다. 그럼 삼각함수를 이용하면 저희는 2D 좌표계에서 x, y를 구할 수 있습니다.

$$x = R * \cos\theta$$

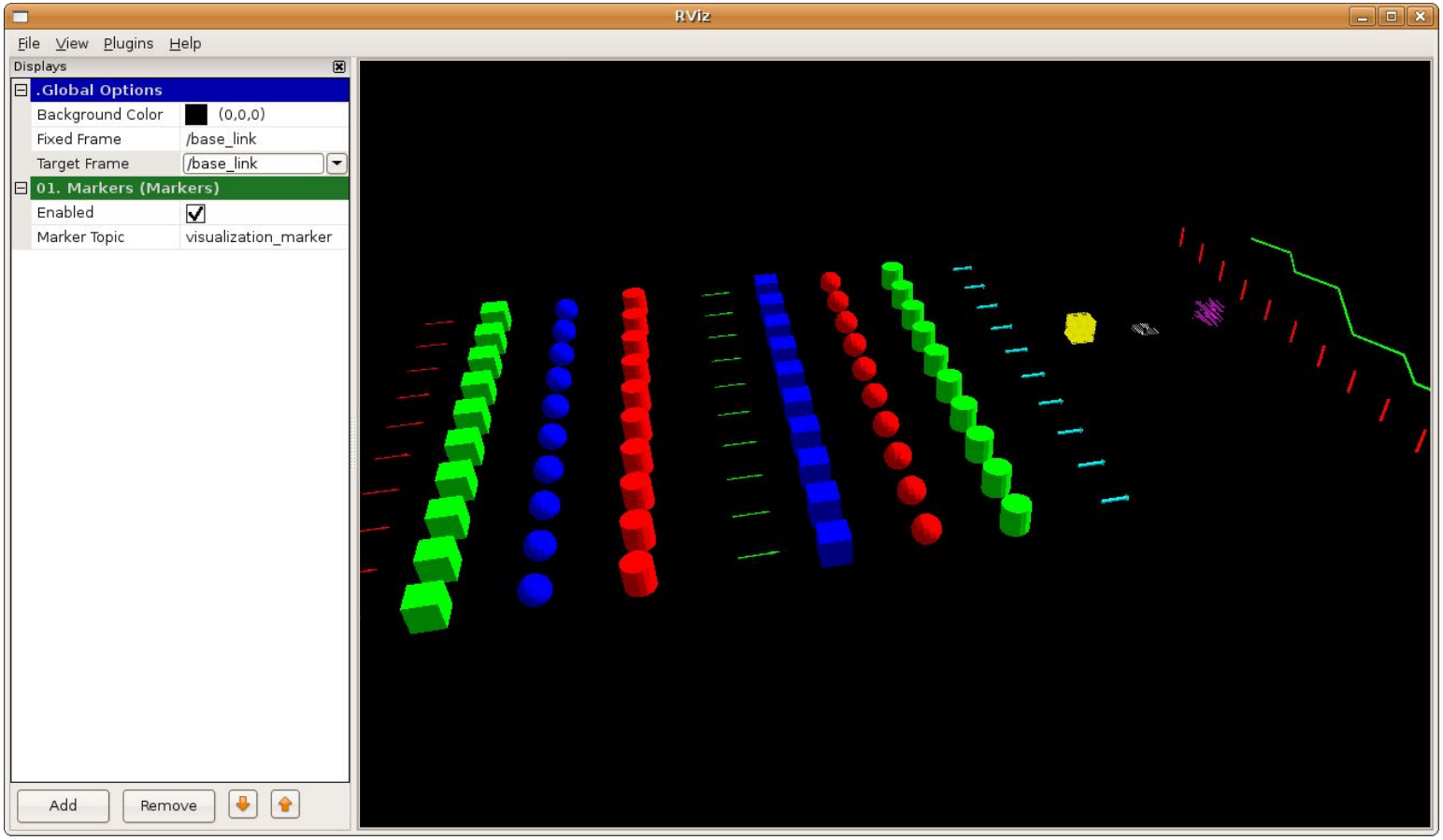
$$y = R * \sin\theta$$

PointCloud는 x, y, z로 이뤄져 있는 메시지입니다. 그렇다면 위에서 구한 좌표를 PointCloud에 대입할 수 있게됩니다.



- visualization_msgs/Marker

rviz에서 어떤 데이터(Lidar와 같은)를 시각화하기 위해 존재하는 메시지



- visualization_msgs/Marker

string ns : marker의 네임스페이스

int32 id : marker의 id

int32 type : marker의 모양

int32 action : ADD, MODIFY, DELETE, DELETEALL 등 marke가 취할 행동

duration lifetime : marker가 좌표계상에 찍혀있을 시간

geometry_msgs/Pose pose :

    pose/Point position : 좌표계상의 위치

    pose/Quaternion orientation : marker가 향하는 방향

geometry_msgs/Vector3 scale : marker의 크기

- visualization_msgs/MarkerArray

Marker 하나는 한 개의 데이터밖에 표시할 수 없습니다. 그래서 여러 데이터(Marker)를 표현하기 위해서는 MarkerArray를 이용해야 합니다. MarkerArray는 이름처럼 다음과 같은 형태를 띄는 메시지 가집니다.

visualization_msgs/Marker[ ] markers

03 Marker

- **Marker**를 사용하기 위해선 첫번째로 **MarkerArray**를 생성하여 줍니다. 그 다음 **Marker**를 생성하여 **MarkerArray.markers**에 하나씩 추가해줍니다. **(PointCloud**와 유사한 방식**)**
- **Marker**는 수많은 메시지가 정의되어 있습니다. 이번 실습에서는 **ADD**만 보도록 하겠습니다.

- 본 코드는 **LaserScan**의 원 데이터를 **Marker**로 바꾸는 코드의 일부입니다.
- **Marker**를 사용하기 위해 **MarkerArray()**를 만들었고 다음 결과 값에 따라 **MarkerArray**의 **markers**에 삽입하는 코드입니다. 다음 페이지에서 **setMarker** 함수를 알아보겠습니다.

```
currentRadian = _ls.angle_min
_rInc = _ls.angle_increment

_mkArray = MarkerArray()

for i in range(0, len(_ls.ranges)):
    x = _dis * math.cos(angle)
    y = _dis * math.sin(angle)
    _mkArray.markers.append(
        self.setMarker(
            (x, y),
            i,
            2
        )
    )
    currentRadian += _rInc

self.pub_marker.publish(_mkArray)
```

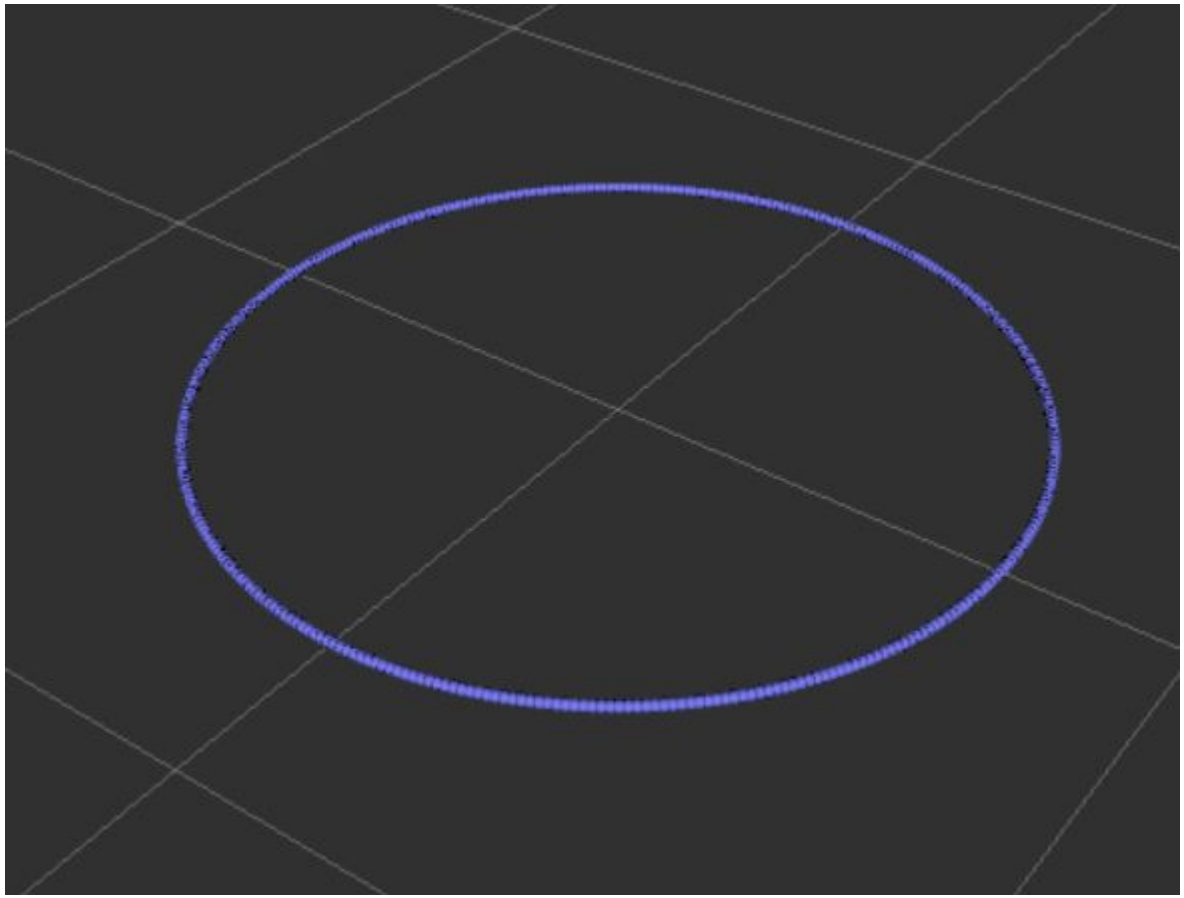
03 Marker

```
marker = Marker()
marker.header.frame_id = "laser"
marker.ns = "position"
marker.id = _id
marker.lifetime = rospy.Duration.from_sec(0.1)

marker.type = Marker.SPHERE
marker.action = Marker.ADD

marker.pose.position.x = _p[0]
marker.pose.position.y = _p[1]
marker.pose.position.z = 0.01

marker.pose.orientation.x = 0.0
marker.pose.orientation.y = 0.0
marker.pose.orientation.z = 0.0
marker.pose.orientation.w = 1.0

marker.scale.x = 0.02
marker.scale.y = 0.02
marker.scale.z = 0.02

marker.color.r = 1.0
marker.color.g = 0.0
marker.color.b = 0.0
marker.color.a = 1.0
```

- Marker를 생성하는 setMarker 함수입니다. 반환으로는 marke를 반환합니다.
- marker.ns(네임스페이스)는 모든 Marker가 동일한 값을 가집니다.
- 하지만 marker.id는 고유의 유니크한 값이기에 절대 같아서는 안됩니다.
- marker.lifetim은 marke가 rviz상에 살아있는 시간입니다. 적당히 0.1초로 하시고 넘어가면 됩니다.
- 이번 실습에서는 구 모양을 추가하는 실습을 진행하겠습니다.
- position은 의미 그대로 위치입니다(x,y,z)
- orientatio은 회전각으로 이해하시면 됩니다. 하지만 꼭 w는 1.0으로 놓아주시길 바랍니다.
- scale은 크기입니다
- rgba는 색상입니다.

- 그래서 최종적으로 소스코드를 실행하면 다음과 같은 결과를 확인할 수 있습니다.

- **Marker(MarkerArray)**를 이용한 원 그리기입니다.

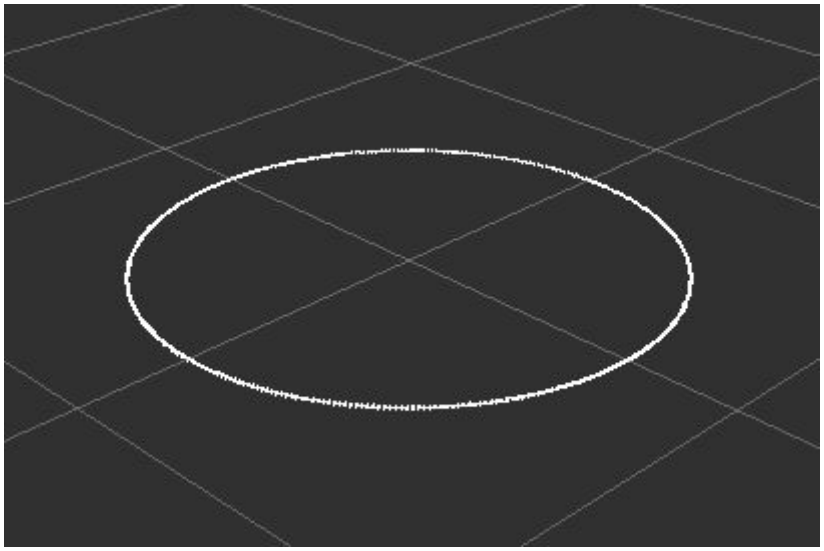


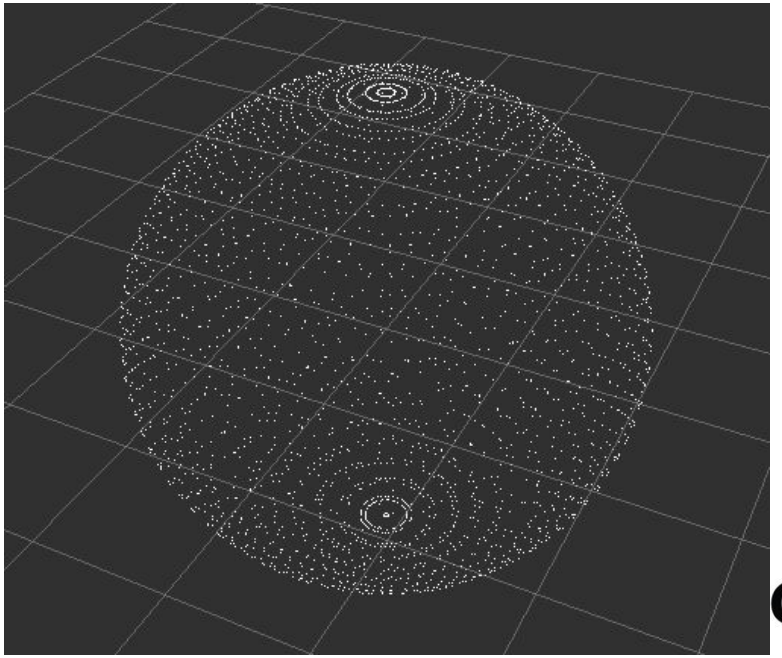
- 실습 예제 4.1~4.2:

이전 장에서 Laserscan을 직접 Publish 해봤습니다.

이번에는 똑같이 2차원좌표에 원을 만드는데 LaserScan이 아닌 PointCloud로 만들어 보세요.

다 만드신 분은 오른쪽처럼 “구”를 만들어보세요.





Go

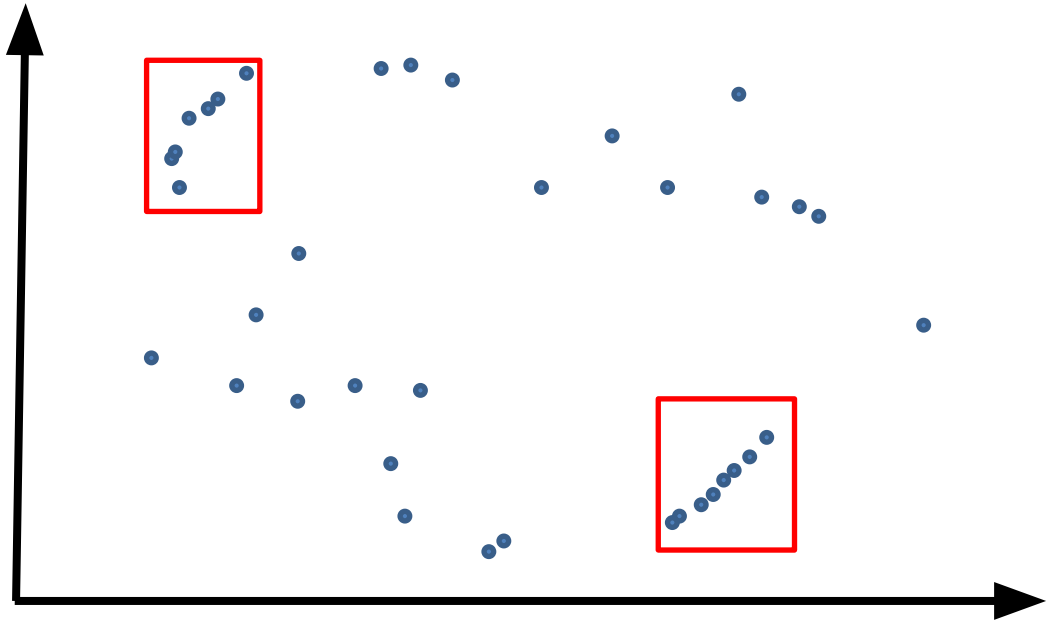
04

Lidar Clustering

WeGo

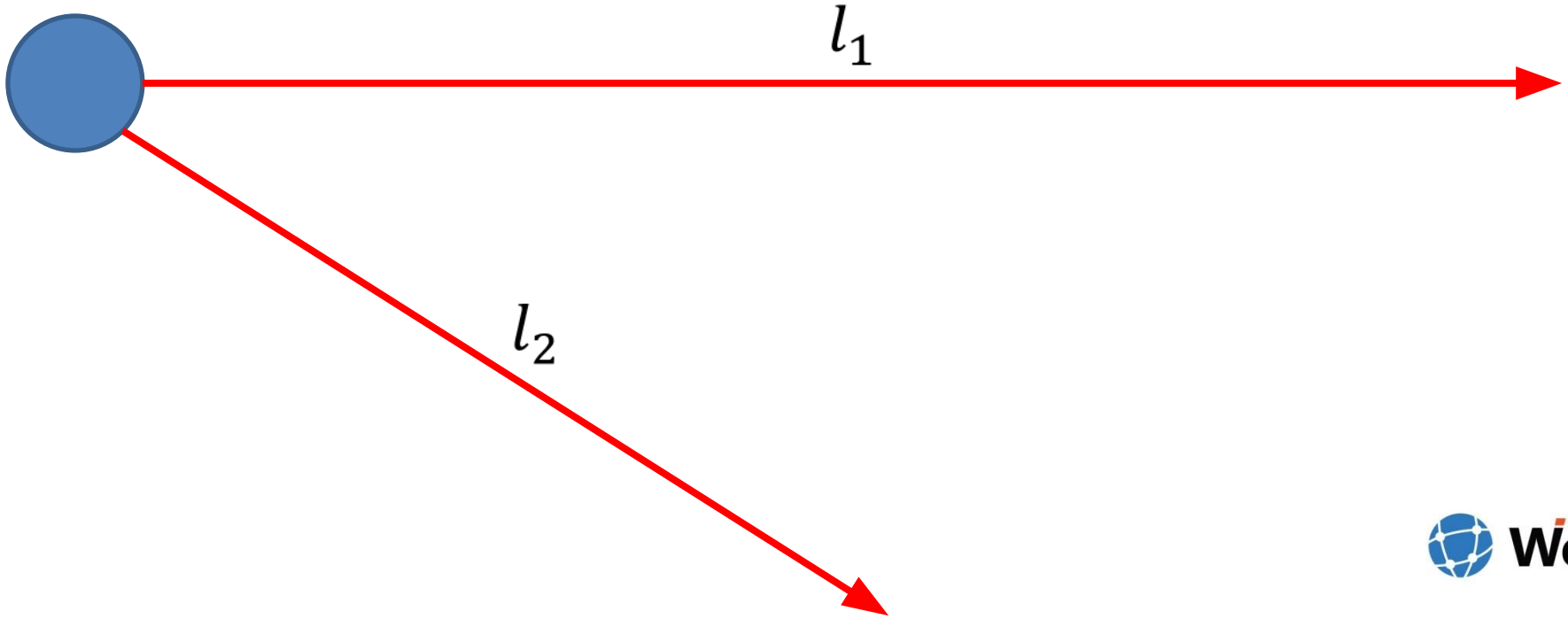
• Point Cloud 군집화:

각각의 Point들을 일정 기준을 통해 하나의 묶음으로 만드는 방법으로 벽이나 물체 등을 구분하기 위해 사용합니다. 3D에서는 이 군집화(Clustering)를 AI에 활용하여 Point Cloud만으로 Object Detection을 수행하는 알고리즘 개발이 활발하게 이뤄지고 있습니다.



• Lidar의 특성:

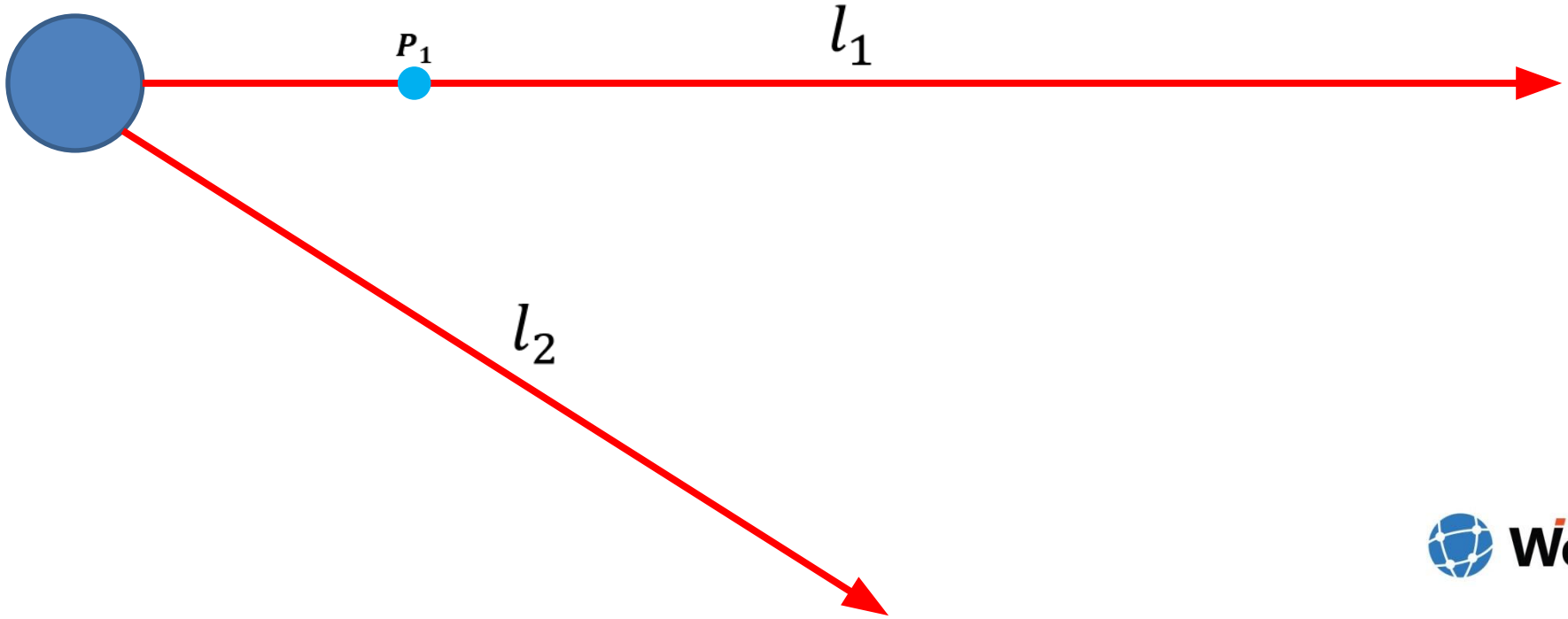
Lidar는 레이저를 이용한 센서입니다. 레이저는 일직선으로 발사됩니다. 아래 그림과 같이 각 위치에서 발사된 두 레이저는 직선(l_1, l_2)으로 표현할 수 있습니다.



• Lidar의 특성:

Laserscan 메시지에서는 거리와 각이 나옵니다. 이를 이용하여, 다음과 같은 정보를 알 수 있습니다.

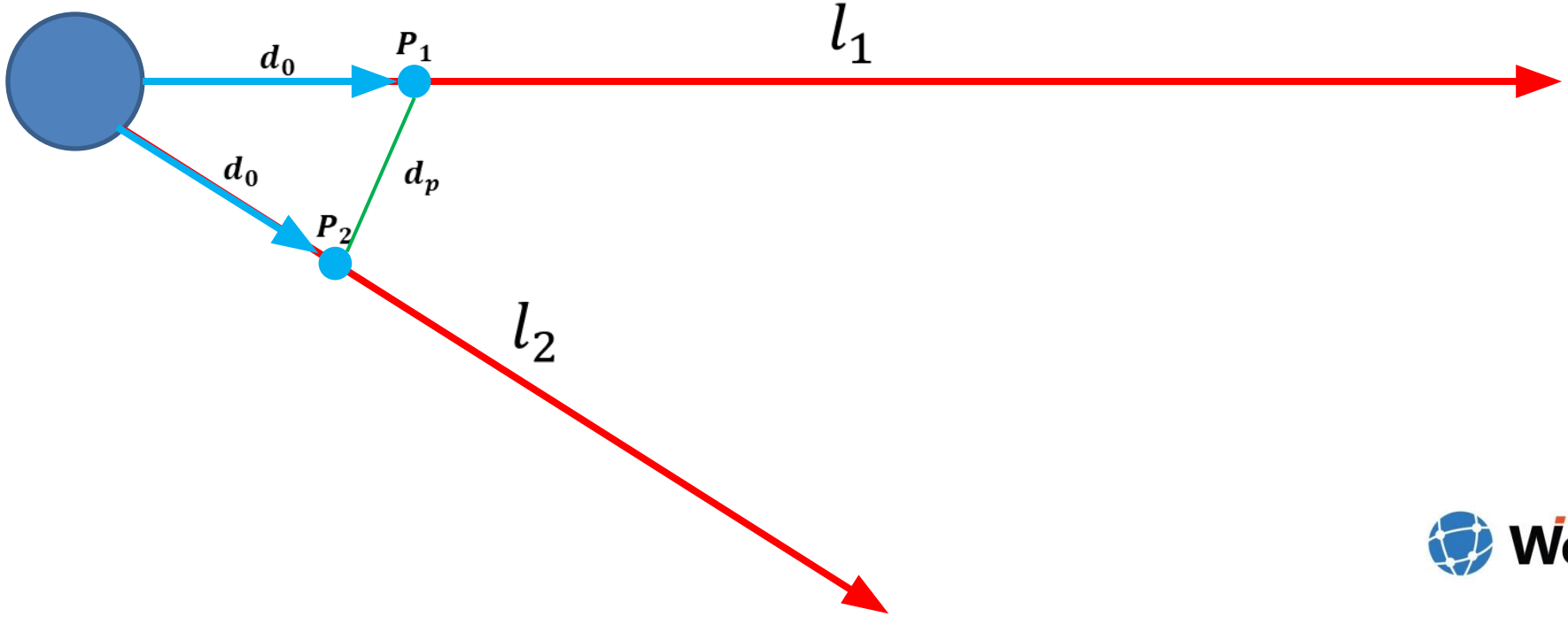
$$\boldsymbol{P_n} = (x_n, y_n) \quad | \quad x_n = r\ \cos\theta \quad | \quad y_n = r\ \sin\theta$$



• **Lidar의 특성:**

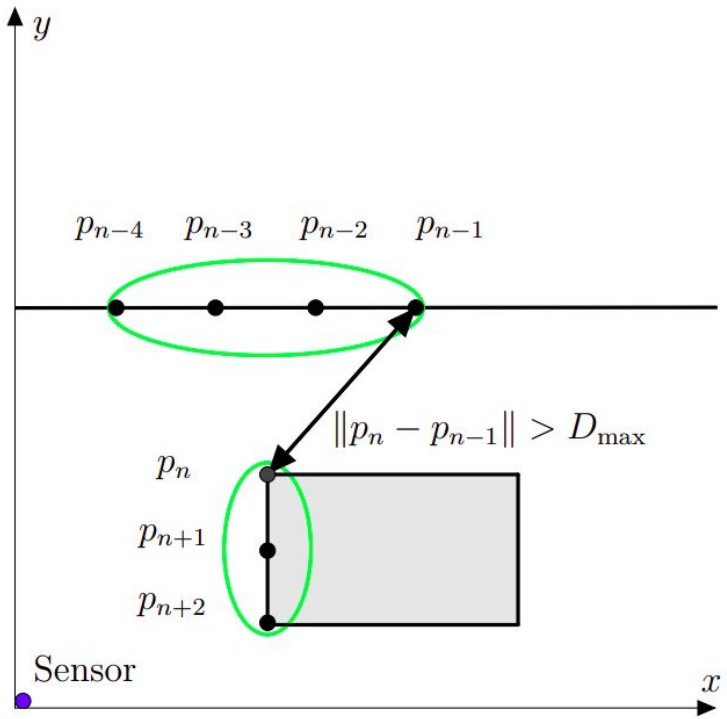
그리고 두 직선에서 동일한 거리(d_0)에 있는 두 점 사이의 거리(d_p)에서 다음과 같이 센서에서 멀어지는 물체는 점과 점 사이의 거리가 멀어지고 가까이 있는 물체는 점과 점 사이의 거리가 가까워짐을 알 수 있습니다.

$$d_0 \propto d_p$$



• Lidar Clustering_Breakpoint:

Breakpoint는 Point간 벌어진 부분을 의미합니다. Clustering을 진행할 때 이 Breakpoint의 기준을 잘 정하는 것이 중요합니다. 가장 간단한 방법으로는 두 Point간의 거리만을 이용하여 나누는 방법이 있습니다.



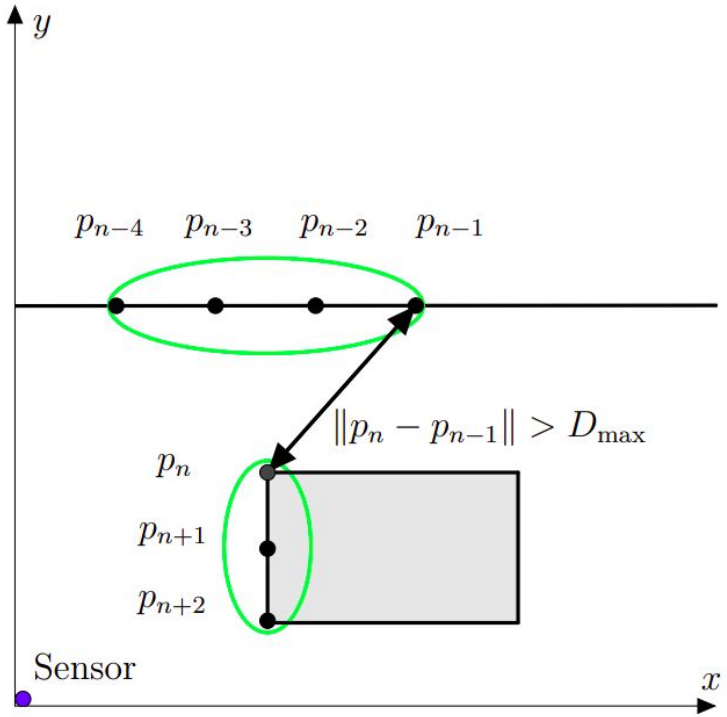
$$||P_n - P_{n-1}|| > D_{max}$$

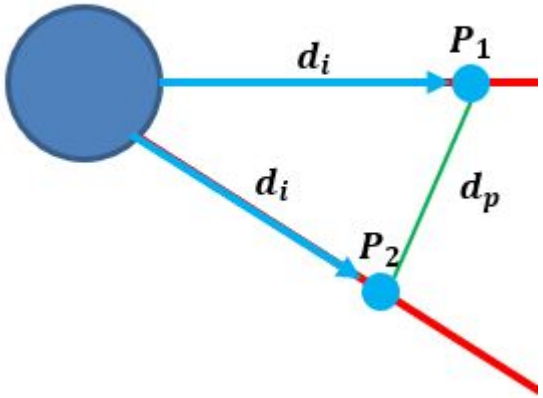
• Lidar Clustering_Breakpoint:

이 방법은 가장 간단하지만, 앞서 본 Lidar의 특성을 생각하면 단점이 많습니다.

같은 크기 조건으로 멀리있는 물체는 두 점 사이의 거리가 멀어지고,

가까이 있는 물체는 거리가 가깝기 때문입니다.

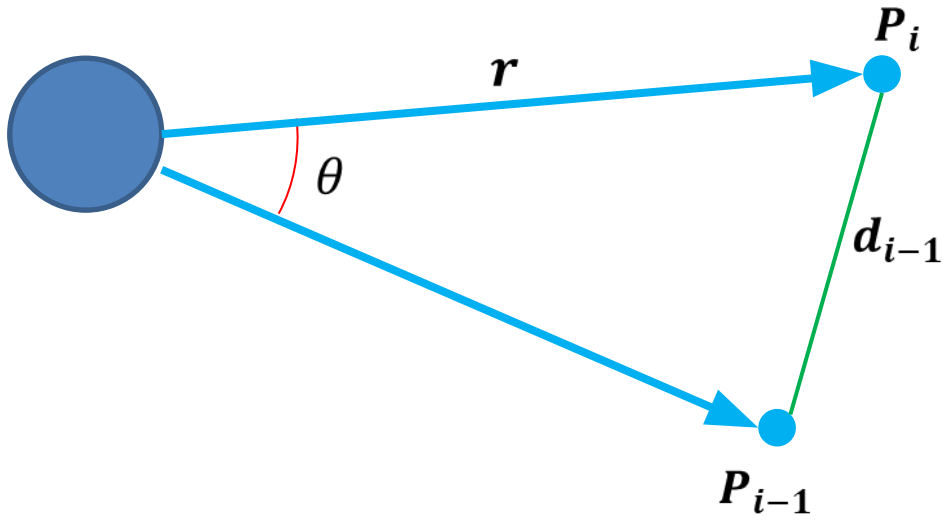




- **Lidar Clustering_Breakpoint:**

이 단점을 개선한 방식이 Adaptive Breakpoint Detector(ABD) 알고리즘입니다.

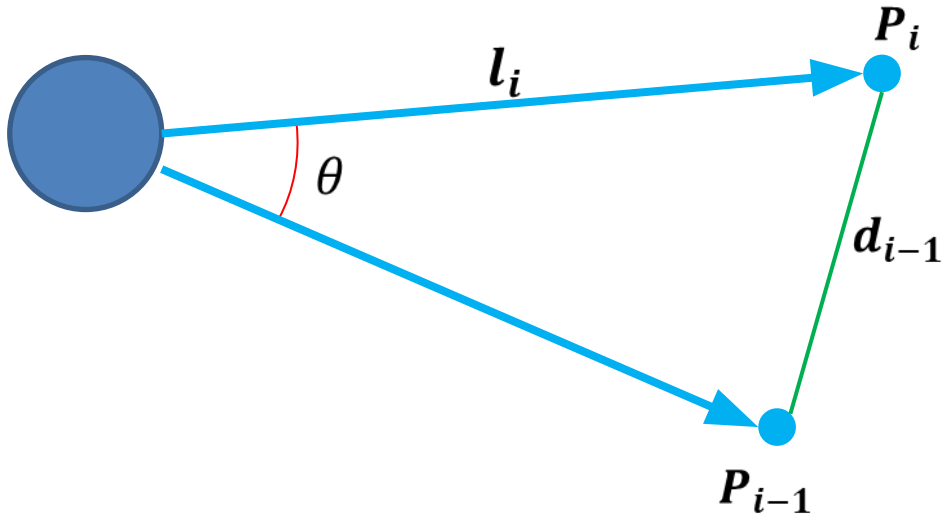
ABD는 Point 거리 비례하여 Breakpoint 기준이 다르게 적용됩니다. 그리고 반지름이 r,



- **Lidar Clustering_Breakpoint:**

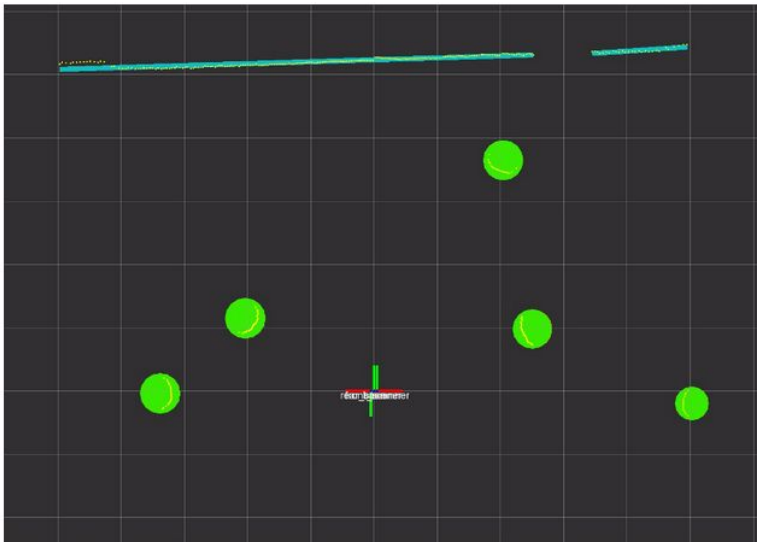
그럼 다음과 같은 가설 하나를 만들 수 있습니다. 여기서 d_p는 radian 값입니다.

$$\text{if } d_{i-1} < d_{group} + l_i d_p : P_{i-1} \text{ is Cluster}$$



- Lidar Clustering_Breakpoint:

이를 바탕으로 슈도코드를 짜보면 다음과 같습니다.



```
#user-defined value
_dgroup = 0.001
_dp = 0.001

for point in Points:
    _distance = p1 to p2 distance
    _ri = p2 to (0,0) distance

    if _distance < hypothesis(_dgroup, _dp, _ri)
        cluster.append(p1)
```

그리고 이 슈도코드를 바탕으로 프로그램을 구성하여 돌려보면 다음과 같은 결과를 얻을 수 있습니다.

https://https://github.com/tysik/obstacle_detector