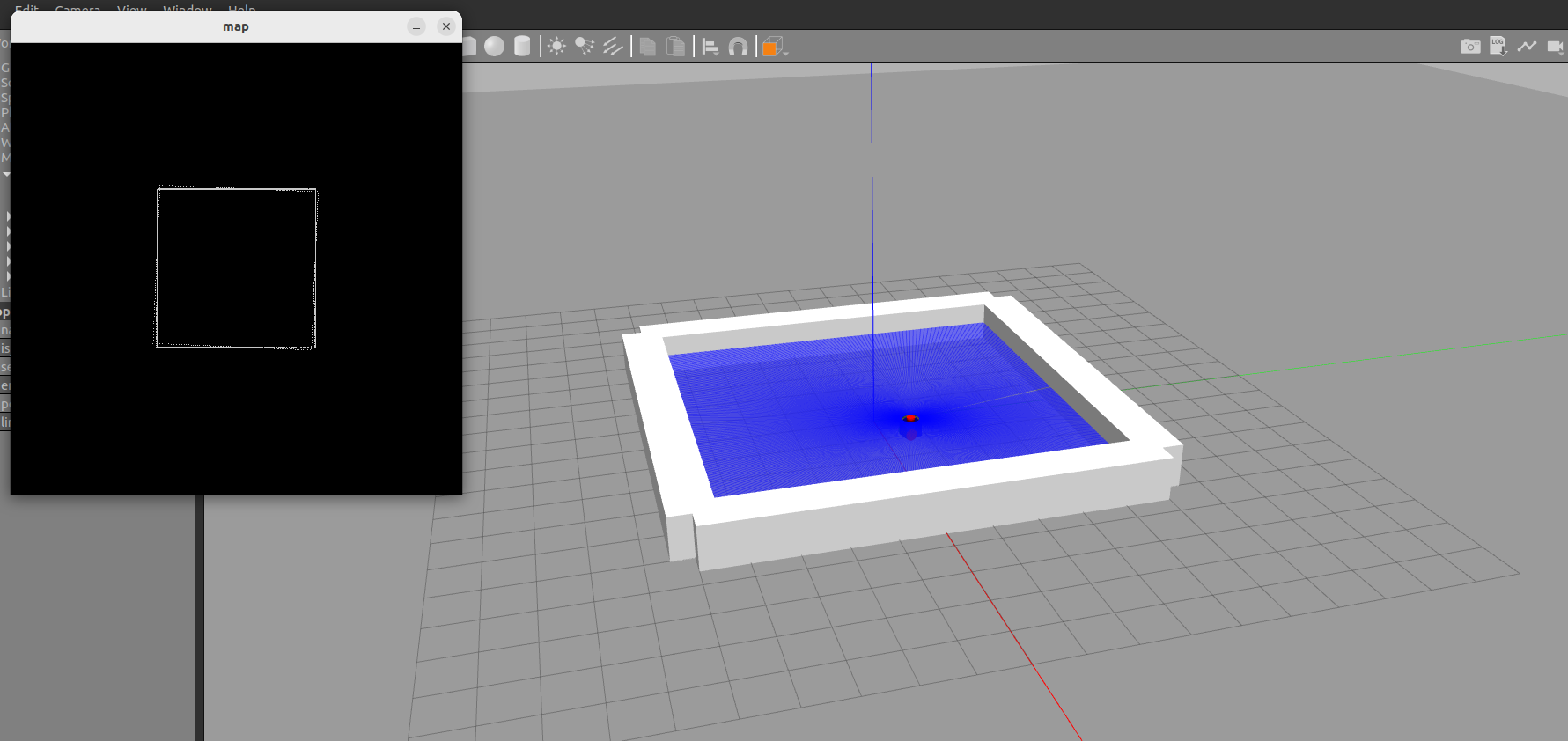
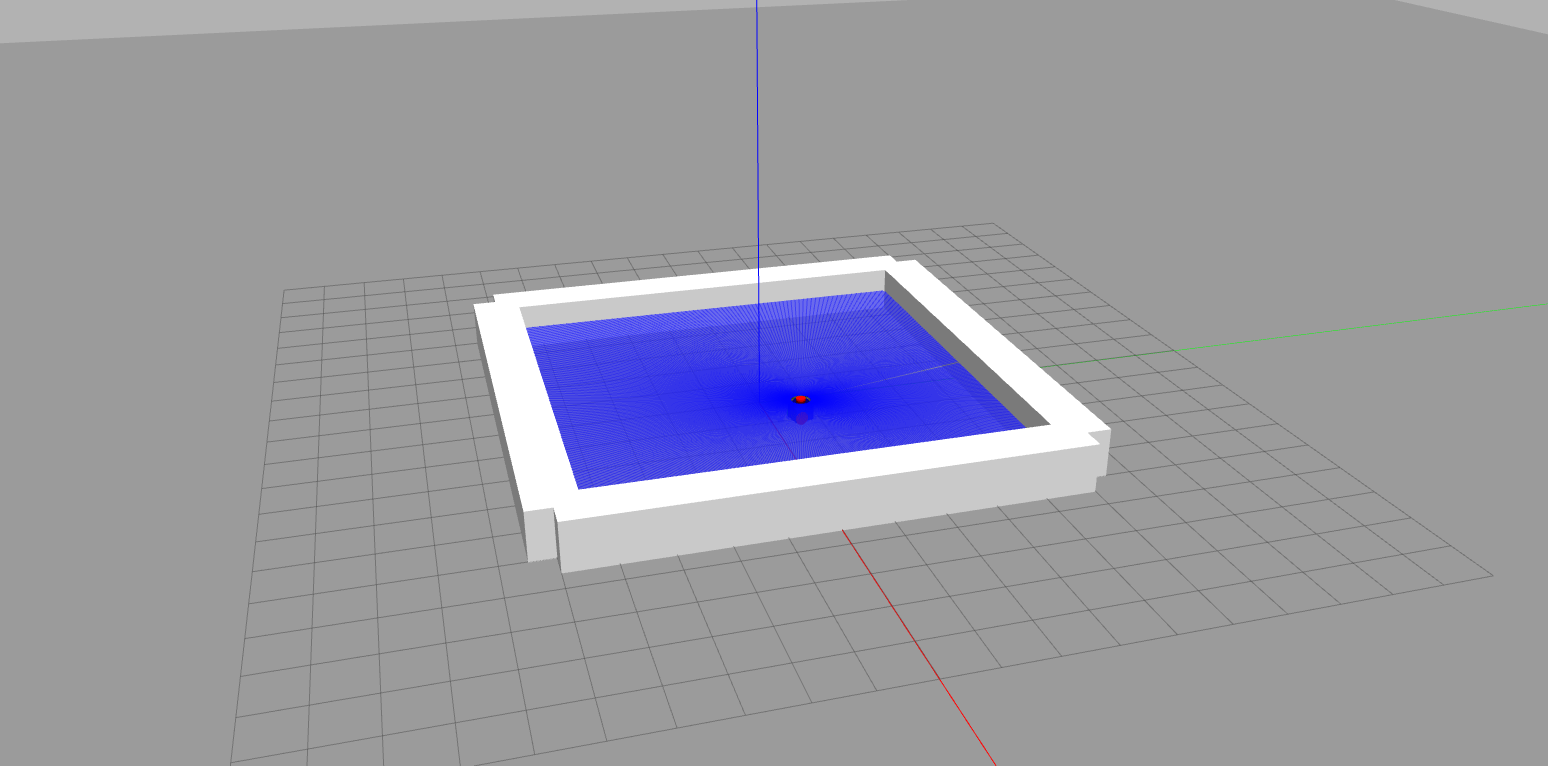
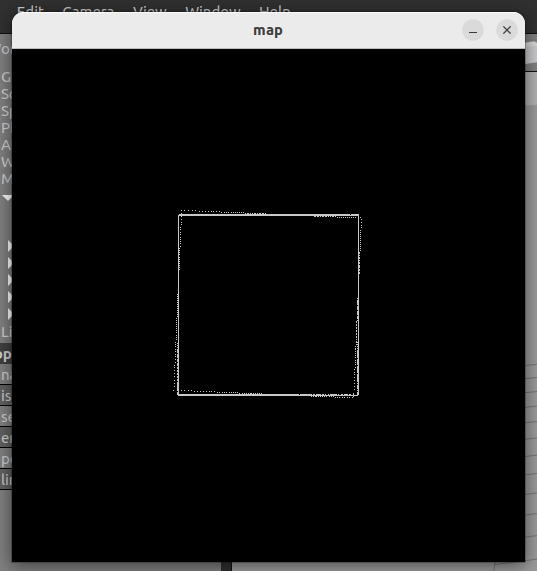
Christopher Budd

218919068



# Adjustments to add\_obstacle

# A very simple ros node to populate the world

#

import rclpy

import os

from gazebo\_msgs.srv import SpawnEntity, DeleteEntity

def make\_obstacle(node, id, x0, y0, h, w, l):

   CYLINDER\_MODEL = """

       <sdf version="1.6">        \

         <world name="default">                         \

           <model name="obstacle">      \

             <static>true</static>      \

             <link name="all">                          \

               <collision name="one">     \

                 <pose>0 0 {o} 0 0 0</pose>       \

                 <geometry>       \

                   <cylinder>                         \

                     <radius>{r}</radius>             \

                     <length>{h}</length>             \

                   </cylinder>        \

                  </geometry>       \

               </collision>       \

               <visual name="two">      \

                 <pose>0 0 {o} 0 0 0</pose>       \

                 <geometry>       \

                   <cylinder>                           \

                     <radius>{r}</radius>               \

                     <length>{h}</length>               \

                   </cylinder>                          \

                 </geometry>        \

               </visual>        \

             </link>                                    \

           </model>         \

         </world>                                       \

       </sdf>"""

   BOX\_MODEL = """

        <sdf version="1.6">                 \

          <world name="default">             \

            <model name="obstacle">          \

              <static>true</static>          \

              <link name="all">               \

                <collision name="one">        \

                  <pose>0 0 {o} 0 0 0</pose>  \

                  <geometry>                  \

                    <box>                     \

                      <size>{l} {w} {h}</size> \

                    </box>                    \

                  </geometry>                 \

                </collision>                  \

                <visual name="two">           \

                  <pose>0 0 {o} 0 0 0</pose>  \

                  <geometry>                  \

                    <box>                     \

                      <size>{l} {w} {h}</size> \

                    </box>                    \

                  </geometry>                 \

                </visual>                     \

              </link>                          \

            </model>                           \

          </world>                             \

         </sdf>"""

   client = node.create\_client(SpawnEntity, "/spawn\_entity")

   node.get\_logger().info("Connecting to /spawn\_entity service...")

   client.wait\_for\_service()

   node.get\_logger().info("...connected")

   request = SpawnEntity.Request()

   request.name = id

   request.initial\_pose.position.x = float(x0)

   request.initial\_pose.position.y = float(y0)

   request.initial\_pose.position.z = float(0)

   dict = {'h' : h, 'l':l, 'w':w, 'o': h/2}

   request.xml = BOX\_MODEL.format(\*\*dict)

   node.get\_logger().info(f"Making request...")

   future = client.call\_async(request)

   while not future.done():

       rclpy.spin\_once(node)

   node.get\_logger().info("...done")

   if not future.result().success:

       node.get\_logger().info(f"Failure {future.result()}")

def remove\_obstacle(node, id):

   client = node.create\_client(DeleteEntity, "/delete\_entity")

   node.get\_logger().info("Connecting to /delete\_entity service...")

   client.wait\_for\_service()

   node.get\_logger().info("...connected")

   request = DeleteEntity.Request()

   request.name = id

   node.get\_logger().info("Making request...")

   future = client.call\_async(request)

   rclpy.spin\_until\_future\_complete(node, future)

   node.get\_logger().info("...done")

   if not future.result().success:

       node.get\_logger().info(f"Failure {future.result()}")

def main(args=None):

  rclpy.init(args=args)

  node = rclpy.create\_node('demo')

  make\_obstacle(node, 'blob', 2, 3, 2, 1)

  node.destroy\_node()

  rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

build\_map to create the map  
# Populate the world with a json map file

#

import sys

import json

import rclpy

from ament\_index\_python.packages import get\_package\_share\_directory

from .add\_obstacle import make\_obstacle

def main(args=None):

    rclpy.init(args=args)

    node = rclpy.create\_node('build\_map')

    node.declare\_parameter('map', 'default.yaml')

    map\_name = node.get\_parameter('map').get\_parameter\_value().string\_value

    package\_path = get\_package\_share\_directory('cpmr\_ch2')

    try:

        with open(f"{package\_path}/{map\_name}") as fd:

            map = json.load(fd)

    except Exception as e:

        node.get\_logger().error(f"Unable to find/parse map in {package\_path}/{map\_name}")

        sys.exit(1)

    for o in map.keys():

        node.get\_logger().info(f"Populating map with {map[o]}")

        make\_obstacle(node, o, map[o]['x'], map[o]['y'], 1.0, map[o]['w'], map[o]['l'])

    node.destroy\_node()

    rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

# json file

{

  "o1" : {"x" : -5, "y" : 0, "l" : 1, "w" : 10},

  "o2" : {"x" : 5, "y" : 0, "l" : 1, "w" : 10},

  "o3" : {"x" : 0, "y" : -5, "l" : 10, "w" : 1},

  "o4" : {"x" : 0, "y" : 5, "l" : 10, "w" : 1}

}

# Lidar map generation

class CollectLidar(Node):

    \_WIDTH = 513

    \_HEIGHT = 513

    \_M\_PER\_PIXEL = 0.05

    cur\_x = 0

    cur\_y = 0

    cur\_t = 0

    def \_\_init\_\_(self):

        super().\_\_init\_\_('collect\_lidar')

        self.get\_logger().info(f'{self.get\_name()} created')

        self.\_map = np.zeros((CollectLidar.\_HEIGHT, CollectLidar.\_WIDTH), dtype=np.uint8)

        self.get\_logger().info(f"map init {self.\_map}")

        self.create\_subscription(Odometry, "/odom", self.\_odom\_callback, 1)

        self.create\_subscription(LaserScan, "/scan", self.\_scan\_callback, 1)

    def \_scan\_callback(self, msg):

        angle\_min = msg.angle\_min

        angle\_max = msg.angle\_max

        angle\_increment = msg.angle\_increment

        ranges = msg.ranges

        # self.get\_logger().info(f"Ranges ({ranges})\n\n")

        self.get\_logger().info(f"lidar ({angle\_min},{angle\_max},{angle\_increment},{len(ranges)})")

        for r in enumerate(ranges):

            # self.get\_logger().info(f"Range {r}")

            scan\_t = self.cur\_t + angle\_min + r[0] \* angle\_increment

            if r[1] < 10:

                self.get\_logger().info(f"Range {r}")

                row = int(int(self.\_HEIGHT/2) - (self.cur\_y + r[1] \* math.sin(scan\_t)) / self.\_M\_PER\_PIXEL)

                col = int(int(self.\_WIDTH/2) + (self.cur\_x + r[1] \* math.cos(scan\_t)) / self.\_M\_PER\_PIXEL)

                self.get\_logger().info(f"Map {row} {col}")

                if (row >= 0) and (col >= 0) and (row < self.\_HEIGHT) and (col < self.\_WIDTH):

                    self.\_map[row, col] = 250

        cv2.imshow('map',self.\_map)

        cv2.waitKey(10)

    def \_odom\_callback(self, msg):

        pose = msg.pose.pose

        self.cur\_x = pose.position.x

        self.cur\_y = pose.position.y

        o = pose.orientation

        roll, pitchc, yaw = euler\_from\_quaternion(o)

        self.cur\_t = yaw

        self.get\_logger().info(f"at ({self.cur\_x},{self.cur\_y},{self.cur\_t})")

def main(args=None):

    rclpy.init(args=args)

    node = CollectLidar()

    try:

        rclpy.spin(node)

    except KeyboardInterrupt:

        pass

    rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

    main()

# Questions

1. How did you decide to represent the map and update it when sensor measurements are obtained?

I represent the map in a 2D array that was located in lidar collector/ collect\_lidar  
I updated based on how close the ping is to prevent distanced reading, and the position of the robot assuming the odom is 100% accurate. Once the x and y of the point is calculated, I fill in that spot in the array. I then display the map.

1. How did you decide where to drive the robot?

I did it based on the object and testing the map. After setting up the box, I added in a few other objects to the map. One key movement was driving all the way around an object to get the full object in the map.

1. Was the model consistent over all measurement obtained?

No, there were some points that were added that were not from an object. This was due to the robot design where it sometimes wobbles and the lidar gets a ping from the floor. There were some other issues with going around objects but likely an error with my math.