Diagram

Description automatically generated with medium confidence

Text

Description automatically generated

(file)

#include <AFMotor.h> // Motor Controller Library

AF\_DCMotor motor1(1);

AF\_DCMotor motor2(2);

AF\_DCMotor motor3(3);

AF\_DCMotor motor4(4);

#define echoPin 16 // attach pin D2 Arduino to pin Echo of HC-SR04

#define trigPin 17 //attach pin D3 Arduino to pin Trig of HC-SR04

int distance;

int min\_distance;

int x= 0;

long t1;

long t2;

long duration; // variable for the duration of sound wave travel

int distanceU; // variable for the distance measurement

int ultrasonic()

{

  // Clears the trigPin condition

  digitalWrite(trigPin, LOW);

  delayMicroseconds(2);

  // Sets the trigPin HIGH (ACTIVE) for 10 microseconds

  digitalWrite(trigPin, HIGH);

  delayMicroseconds(10);

  digitalWrite(trigPin, LOW);

  // Reads the echoPin, returns the sound wave travel time in microseconds

  duration = pulseIn(echoPin, HIGH);

  // Calculating the distance

  distanceU = duration \* 0.034 / 2; // Speed of sound wave divided by 2 (go and back)

  // Displays the distance on the Serial Monitor

  //Serial.print("Distance: ");

  //Serial.print(distanceU);

  //Serial.println(" cm");

  return distanceU;

}

void moveForward() {

  motor1.setSpeed(200);

  motor2.setSpeed(200);

  motor3.setSpeed(200);

  motor4.setSpeed(200);

  motor1.run(FORWARD);

  motor2.run(FORWARD);

  motor3.run(FORWARD);

  motor4.run(FORWARD);

}

void moveBackward() {

  motor1.setSpeed(200);

  motor2.setSpeed(200);

  motor3.setSpeed(200);

  motor4.setSpeed(200);

  motor1.run(BACKWARD);

  motor2.run(BACKWARD);

  motor3.run(BACKWARD);

  motor4.run(BACKWARD);

}

void Stop() {

  motor1.run(RELEASE);

  motor2.run(RELEASE);

  motor3.run(RELEASE);

  motor4.run(RELEASE);

}

void setup() {

  Serial.begin(9600);

  min\_distance = 5;

  // Ultrasonic setup

  pinMode(trigPin, OUTPUT); // Sets the trigPin as an OUTPUT

  pinMode(echoPin, INPUT); // Sets the echoPin as an INPUT

  Serial.begin(9600); // // Serial Communication is starting with 9600 of baudrate speed

  //Serial.println("Ultrasonic Sensor HC-SR04 Test"); // print some text in Serial Monitor

  //Serial.println("with Arduino UNO R3");

}

void loop() {

//t1 = millis();

distance = ultrasonic();

  if (distance <= min\_distance)

  {

    Stop();

    Serial.println(distance);

    //Serial.write('\n');

    //Serial.write("stop");

    //Serial.write('\n');

    //delay ();

  }

  else

  {

    moveForward();

    Serial.println(distance);

    //Serial.write('\n');

    //Serial.write("moving");

    //Serial.write('\n');

    //delay (3000);

  }

delay(1500);

//t2 = millis();

}

Diagram

Description automatically generated

(file)

#!/usr/bin/env python3

import serial

import json

import time

import AWSIoTPythonSDK.MQTTLib as AWSIoTPyMQTT

# Define ENDPOINT, CLIENT\_ID, PATH\_TO\_CERTIFICATE, PATH\_TO\_PRIVATE\_KEY, PATH\_TO\_AMAZON\_ROOT\_CA\_1, MESSAGE, TOPIC, and RANGE

ENDPOINT = "avsnsxi1w2nv8-ats.iot.me-central-1.amazonaws.com"

CLIENT\_ID = "iot\_thing"

PATH\_TO\_CERTIFICATE = "d42194e6a78b821823451cc7b1d29896291e00dffd00576f6197250175f33b2b-certificate.pem.crt"

PATH\_TO\_PRIVATE\_KEY = "d42194e6a78b821823451cc7b1d29896291e00dffd00576f6197250175f33b2b-private.pem.key"

PATH\_TO\_AMAZON\_ROOT\_CA\_1 = "AmazonRootCA1.pem"

TOPIC = "test/testing"

myAWSIoTMQTTClient = AWSIoTPyMQTT.AWSIoTMQTTClient(CLIENT\_ID)

myAWSIoTMQTTClient.configureEndpoint(ENDPOINT, 8883)

myAWSIoTMQTTClient.configureCredentials(PATH\_TO\_AMAZON\_ROOT\_CA\_1, PATH\_TO\_PRIVATE\_KEY, PATH\_TO\_CERTIFICATE)

myAWSIoTMQTTClient.configureMQTTOperationTimeout(1000)

myAWSIoTMQTTClient.connect()

ser = serial.Serial('/dev/ttyACM0', 9600, timeout=2)

ser.reset\_input\_buffer()

def sr():

    reading = ser.readline().decode()

    reading = reading.strip()

    distance = int(reading) if len(reading) > 0 else 0

    return distance

def publish\_to\_cloud(msg):

    myAWSIoTMQTTClient.publish(TOPIC, json.dumps(msg), 1)

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

    counter = 0

    while True:

        distance = sr()

        publish\_to\_cloud(distance)

        #time.sleep(1)

        print(distance)

Diagram

Description automatically generated

(file – part1)

#!/usr/bin/env python3.7

import time

import sys

sys.path.append("/home/m/catkin\_ws/src/pro/scripts/PythonAPI/iot")

import rospy

from std\_msgs.msg import Float32

from subscribe import mqttc

class Echo(object):

    def \_\_init\_\_(self):

        rospy.init\_node('echoer')

        self.pub = rospy.Publisher('/out\_value', Float32, latch=True, queue\_size=10)

        rospy.Subscriber('/out\_value', Float32)

        self.distance = 0.2

    def on\_message(self, client, userdata, msg):  # Func for receiving msgs

        # print("topic: " + msg.topic)

        # print("payload: " + str(msg.payload))

        self.distance = int(msg.payload.decode("utf-8"))

        print(msg.payload, self.distance)

    def run(self):

        mqttc.loop\_start()

        while not rospy.is\_shutdown():

            self.pub.publish(self.distance)

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

    echo\_obj = Echo()

    mqttc.on\_message = echo\_obj.on\_message  # assign on\_message func

    time.sleep(0.1)

    echo\_obj.run()

(file – part2)

#!/usr/bin/env python3.7

import ssl

import paho.mqtt.client as paho

awshost = "avsnsxi1w2nv8-ats.iot.me-central-1.amazonaws.com"  # Endpoint

awsport = 8883  # Port no.

clientId = "iot\_thing"  # Thing\_Name

thingName = "iot\_thing"  # Thing\_Name

caPath = "AmazonRootCA1.pem"  # Root\_CA\_Certificate\_Name

certPath = "d42194e6a78b821823451cc7b1d29896291e00dffd00576f6197250175f33b2b-certificate.pem.crt"  # <Thing\_Name>.cert.pem

keyPath = "d42194e6a78b821823451cc7b1d29896291e00dffd00576f6197250175f33b2b-private.pem.key"  # <Thing\_Name>.private.key

TOPIC = "test/testing"

def on\_connect(client, userdata, flags, rc):  # func for making connection

    print("Connection returned result: " + str(rc))

    # Subscribing in on\_connect() means that if we lose the connection and

    # reconnect then subscriptions will be renewed.

    client.subscribe(TOPIC, 1)  # Subscribe to all topics

# def on\_log(client, userdata, level, msg):

#    print(msg.topic+" "+str(msg.payload))

mqttc = paho.Client()  # mqttc object

mqttc.on\_connect = on\_connect  # assign on\_connect func

# mqttc.on\_log = on\_log

mqttc.tls\_set(caPath, certfile=certPath, keyfile=keyPath, cert\_reqs=ssl.CERT\_REQUIRED, tls\_version=ssl.PROTOCOL\_TLSv1\_2, ciphers=None)  # pass parameters

mqttc.connect(awshost, awsport, keepalive=60)  # connect to aws server

#mqttc.loop\_start()

Diagram

Description automatically generated

(file)

#!/usr/bin/env python3.7

# Copyright (c) 2018 Intel Labs.

# authors: German Ros (german.ros@intel.com)

#

# This work is licensed under the terms of the MIT license.

# For a copy, see <https://opensource.org/licenses/MIT>.

"""Example of automatic vehicle control from client side."""

from \_\_future\_\_ import print\_function

import argparse

import collections

import datetime

import glob

import logging

import math

import os

import numpy.random as random

import re

import sys

import weakref

import rospy

from std\_msgs.msg import Float32

try:

    import pygame

    from pygame.locals import KMOD\_CTRL

    from pygame.locals import K\_ESCAPE

    from pygame.locals import K\_q

except ImportError:

    raise RuntimeError('cannot import pygame, make sure pygame package is installed')

try:

    import numpy as np

except ImportError:

    raise RuntimeError(

        'cannot import numpy, make sure numpy package is installed')

# ==============================================================================

# -- Find CARLA module ---------------------------------------------------------

# ==============================================================================

try:

    sys.path.append(glob.glob('../carla/dist/carla-\*%d.%d-%s.egg' % (

        sys.version\_info.major,

        sys.version\_info.minor,

        'win-amd64' if os.name == 'nt' else 'linux-x86\_64'))[0])

except IndexError:

    pass

# ==============================================================================

# -- Add PythonAPI for release mode --------------------------------------------

# ==============================================================================

try:

    sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))) + '/carla')

except IndexError:

    pass

import carla

from carla import ColorConverter as cc

from agents.navigation.behavior\_agent import BehaviorAgent  # pylint: disable=import-error

from agents.navigation.basic\_agent import BasicAgent  # pylint: disable=import-error

# ==============================================================================

# -- Global functions ----------------------------------------------------------

# ==============================================================================

def find\_weather\_presets():

    """Method to find weather presets"""

    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')

    def name(x): return ' '.join(m.group(0) for m in rgx.finditer(x))

    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]

    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]

def get\_actor\_display\_name(actor, truncate=250):

    """Method to get actor display name"""

    name = ' '.join(actor.type\_id.replace('\_', '.').title().split('.')[1:])

    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name

# ==============================================================================

# -- World ---------------------------------------------------------------

# ==============================================================================

class World(object):

    """ Class representing the surrounding environment """

    def \_\_init\_\_(self, client, carla\_world, hud, args):

        """Constructor method"""

        self.\_args = args

        self.world = carla\_world

        try:

            #self.map = self.world.get\_map()

            self.map = client.load\_world('Town01\_Opt')

        except RuntimeError as error:

            print('RuntimeError: {}'.format(error))

            print('  The server could not send the OpenDRIVE (.xodr) file:')

            print('  Make sure it exists, has the same name of your town, and is correct.')

            sys.exit(1)

        self.hud = hud

        self.player = None

        self.collision\_sensor = None

        self.lane\_invasion\_sensor = None

        self.gnss\_sensor = None

        self.camera\_manager = None

        self.\_weather\_presets = find\_weather\_presets()

        self.\_weather\_index = 0

        self.\_actor\_filter = args.filter

        self.restart(args)

        self.world.on\_tick(hud.on\_world\_tick)

        self.recording\_enabled = False

        self.recording\_start = 0

    def restart(self, args):

        """Restart the world"""

        # Keep same camera config if the camera manager exists.

        cam\_index = self.camera\_manager.index if self.camera\_manager is not None else 0

        cam\_pos\_id = self.camera\_manager.transform\_index if self.camera\_manager is not None else 0

        # Get a random blueprint.

        blueprint = self.world.get\_blueprint\_library().filter('charger\_2020')[0]

        blueprint.set\_attribute('role\_name', 'hero')

        if blueprint.has\_attribute('color'):

            color = blueprint.get\_attribute('color').recommended\_values[0]

            blueprint.set\_attribute('color', color)

        # Spawn the player.

        if self.player is not None:

            spawn\_point = self.player.get\_transform()

            spawn\_point.location.z += 2.0

            spawn\_point.rotation.roll = 0.0

            spawn\_point.rotation.pitch = 0.0

            self.destroy()

            self.player = self.world.try\_spawn\_actor(blueprint, spawn\_point)

            self.modify\_vehicle\_physics(self.player)

        while self.player is None:

            spawn\_point = carla.Transform()

            spawn\_point.location.x = 10.868797

            spawn\_point.location.y = 2.461965

            spawn\_point.location.z = 0.3

            spawn\_point.rotation.pitch = 0

            spawn\_point.rotation.yaw = 0

            spawn\_point.rotation.roll = 0

            self.player = self.world.try\_spawn\_actor(blueprint, spawn\_point)

            self.modify\_vehicle\_physics(self.player)

        if self.\_args.sync:

            self.world.tick()

        else:

            self.world.wait\_for\_tick()

        # Set up the sensors.

        self.collision\_sensor = CollisionSensor(self.player, self.hud)

        self.lane\_invasion\_sensor = LaneInvasionSensor(self.player, self.hud)

        self.gnss\_sensor = GnssSensor(self.player)

        self.camera\_manager = CameraManager(self.player, self.hud)

        self.camera\_manager.transform\_index = cam\_pos\_id

        self.camera\_manager.set\_sensor(cam\_index, notify=False)

        actor\_type = get\_actor\_display\_name(self.player)

        self.hud.notification(actor\_type)

    def next\_weather(self, reverse=False):

        """Get next weather setting"""

        self.\_weather\_index += -1 if reverse else 1

        self.\_weather\_index %= len(self.\_weather\_presets)

        preset = self.\_weather\_presets[self.\_weather\_index]

        self.hud.notification('Weather: %s' % preset[1])

        self.player.get\_world().set\_weather(preset[0])

    def modify\_vehicle\_physics(self, actor):

        #If actor is not a vehicle, we cannot use the physics control

        try:

            physics\_control = actor.get\_physics\_control()

            physics\_control.use\_sweep\_wheel\_collision = True

            actor.apply\_physics\_control(physics\_control)

        except Exception:

            pass

    def tick(self, clock):

        """Method for every tick"""

        self.hud.tick(self, clock)

    def render(self, display):

        """Render world"""

        self.camera\_manager.render(display)

        self.hud.render(display)

    def destroy\_sensors(self):

        """Destroy sensors"""

        self.camera\_manager.sensor.destroy()

        self.camera\_manager.sensor = None

        self.camera\_manager.index = None

    def destroy(self):

        """Destroys all actors"""

        actors = [

            self.camera\_manager.sensor,

            self.collision\_sensor.sensor,

            self.lane\_invasion\_sensor.sensor,

            self.gnss\_sensor.sensor,

            self.player]

        for actor in actors:

            if actor is not None:

                actor.destroy()

# ==============================================================================

# -- KeyboardControl -----------------------------------------------------------

# ==============================================================================

class KeyboardControl(object):

    def \_\_init\_\_(self, world):

        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

    def parse\_events(self):

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                return True

            if event.type == pygame.KEYUP:

                if self.\_is\_quit\_shortcut(event.key):

                    return True

    @staticmethod

    def \_is\_quit\_shortcut(key):

        """Shortcut for quitting"""

        return (key == K\_ESCAPE) or (key == K\_q and pygame.key.get\_mods() & KMOD\_CTRL)

# ==============================================================================

# -- HUD -----------------------------------------------------------------------

# ==============================================================================

class HUD(object):

    """Class for HUD text"""

    def \_\_init\_\_(self, width, height):

        """Constructor method"""

        self.dim = (width, height)

        font = pygame.font.Font(pygame.font.get\_default\_font(), 20)

        font\_name = 'courier' if os.name == 'nt' else 'mono'

        fonts = [x for x in pygame.font.get\_fonts() if font\_name in x]

        default\_font = 'ubuntumono'

        mono = default\_font if default\_font in fonts else fonts[0]

        mono = pygame.font.match\_font(mono)

        self.\_font\_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)

        self.\_notifications = FadingText(font, (width, 40), (0, height - 40))

        self.help = HelpText(pygame.font.Font(mono, 24), width, height)

        self.server\_fps = 0

        self.frame = 0

        self.simulation\_time = 0

        self.\_show\_info = True

        self.\_info\_text = []

        self.\_server\_clock = pygame.time.Clock()

    def on\_world\_tick(self, timestamp):

        """Gets informations from the world at every tick"""

        self.\_server\_clock.tick()

        self.server\_fps = self.\_server\_clock.get\_fps()

        self.frame = timestamp.frame\_count

        self.simulation\_time = timestamp.elapsed\_seconds

    def tick(self, world, clock):

        """HUD method for every tick"""

        self.\_notifications.tick(world, clock)

        if not self.\_show\_info:

            return

        transform = world.player.get\_transform()

        vel = world.player.get\_velocity()

        control = world.player.get\_control()

        heading = 'N' if abs(transform.rotation.yaw) < 89.5 else ''

        heading += 'S' if abs(transform.rotation.yaw) > 90.5 else ''

        heading += 'E' if 179.5 > transform.rotation.yaw > 0.5 else ''

        heading += 'W' if -0.5 > transform.rotation.yaw > -179.5 else ''

        colhist = world.collision\_sensor.get\_collision\_history()

        collision = [colhist[x + self.frame - 200] for x in range(0, 200)]

        max\_col = max(1.0, max(collision))

        collision = [x / max\_col for x in collision]

        vehicles = world.world.get\_actors().filter('vehicle.\*')

        self.\_info\_text = [

            'Server:  % 16.0f FPS' % self.server\_fps,

            'Client:  % 16.0f FPS' % clock.get\_fps(),

            '',

            'Vehicle: % 20s' % get\_actor\_display\_name(world.player, truncate=20),

            #'Map:     % 20s' % world.map.name.split('/')[-1],

            'Simulation time: % 12s' % datetime.timedelta(seconds=int(self.simulation\_time)),

            '',

            'Speed:   % 15.0f km/h' % (3.6 \* math.sqrt(vel.x\*\*2 + vel.y\*\*2 + vel.z\*\*2)),

            u'Heading:% 16.0f\N{DEGREE SIGN} % 2s' % (transform.rotation.yaw, heading),

            'Location:% 20s' % ('(% 5.1f, % 5.1f)' % (transform.location.x, transform.location.y)),

            'GNSS:% 24s' % ('(% 2.6f, % 3.6f)' % (world.gnss\_sensor.lat, world.gnss\_sensor.lon)),

            'Height:  % 18.0f m' % transform.location.z,

            '']

        if isinstance(control, carla.VehicleControl):

            self.\_info\_text += [

                ('Throttle:', control.throttle, 0.0, 1.0),

                ('Steer:', control.steer, -1.0, 1.0),

                ('Brake:', control.brake, 0.0, 1.0),

                ('Reverse:', control.reverse),

                ('Hand brake:', control.hand\_brake),

                ('Manual:', control.manual\_gear\_shift),

                'Gear:        %s' % {-1: 'R', 0: 'N'}.get(control.gear, control.gear)]

        elif isinstance(control, carla.WalkerControl):

            self.\_info\_text += [

                ('Speed:', control.speed, 0.0, 5.556),

                ('Jump:', control.jump)]

        self.\_info\_text += [

            '',

            'Collision:',

            collision,

            '',

            'Number of vehicles: % 8d' % len(vehicles)]

        if len(vehicles) > 1:

            self.\_info\_text += ['Nearby vehicles:']

        def dist(l):

            return math.sqrt((l.x - transform.location.x)\*\*2 + (l.y - transform.location.y)

                             \*\* 2 + (l.z - transform.location.z)\*\*2)

        vehicles = [(dist(x.get\_location()), x) for x in vehicles if x.id != world.player.id]

        for dist, vehicle in sorted(vehicles):

            if dist > 200.0:

                break

            vehicle\_type = get\_actor\_display\_name(vehicle, truncate=22)

            self.\_info\_text.append('% 4dm %s' % (dist, vehicle\_type))

    def toggle\_info(self):

        """Toggle info on or off"""

        self.\_show\_info = not self.\_show\_info

    def notification(self, text, seconds=2.0):

        """Notification text"""

        self.\_notifications.set\_text(text, seconds=seconds)

    def error(self, text):

        """Error text"""

        self.\_notifications.set\_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):

        """Render for HUD class"""

        if self.\_show\_info:

            info\_surface = pygame.Surface((220, self.dim[1]))

            info\_surface.set\_alpha(100)

            display.blit(info\_surface, (0, 0))

            v\_offset = 4

            bar\_h\_offset = 100

            bar\_width = 106

            for item in self.\_info\_text:

                if v\_offset + 18 > self.dim[1]:

                    break

                if isinstance(item, list):

                    if len(item) > 1:

                        points = [(x + 8, v\_offset + 8 + (1 - y) \* 30) for x, y in enumerate(item)]

                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)

                    item = None

                    v\_offset += 18

                elif isinstance(item, tuple):

                    if isinstance(item[1], bool):

                        rect = pygame.Rect((bar\_h\_offset, v\_offset + 8), (6, 6))

                        pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1] else 1)

                    else:

                        rect\_border = pygame.Rect((bar\_h\_offset, v\_offset + 8), (bar\_width, 6))

                        pygame.draw.rect(display, (255, 255, 255), rect\_border, 1)

                        fig = (item[1] - item[2]) / (item[3] - item[2])

                        if item[2] < 0.0:

                            rect = pygame.Rect(

                                (bar\_h\_offset + fig \* (bar\_width - 6), v\_offset + 8), (6, 6))

                        else:

                            rect = pygame.Rect((bar\_h\_offset, v\_offset + 8), (fig \* bar\_width, 6))

                        pygame.draw.rect(display, (255, 255, 255), rect)

                    item = item[0]

                if item:  # At this point has to be a str.

                    surface = self.\_font\_mono.render(item, True, (255, 255, 255))

                    display.blit(surface, (8, v\_offset))

                v\_offset += 18

        self.\_notifications.render(display)

        self.help.render(display)

# ==============================================================================

# -- FadingText ----------------------------------------------------------------

# ==============================================================================

class FadingText(object):

    """ Class for fading text """

    def \_\_init\_\_(self, font, dim, pos):

        """Constructor method"""

        self.font = font

        self.dim = dim

        self.pos = pos

        self.seconds\_left = 0

        self.surface = pygame.Surface(self.dim)

    def set\_text(self, text, color=(255, 255, 255), seconds=2.0):

        """Set fading text"""

        text\_texture = self.font.render(text, True, color)

        self.surface = pygame.Surface(self.dim)

        self.seconds\_left = seconds

        self.surface.fill((0, 0, 0, 0))

        self.surface.blit(text\_texture, (10, 11))

    def tick(self, \_, clock):

        """Fading text method for every tick"""

        delta\_seconds = 1e-3 \* clock.get\_time()

        self.seconds\_left = max(0.0, self.seconds\_left - delta\_seconds)

        self.surface.set\_alpha(500.0 \* self.seconds\_left)

    def render(self, display):

        """Render fading text method"""

        display.blit(self.surface, self.pos)

# ==============================================================================

# -- HelpText ------------------------------------------------------------------

# ==============================================================================

class HelpText(object):

    """ Helper class for text render"""

    def \_\_init\_\_(self, font, width, height):

        """Constructor method"""

        lines = \_\_doc\_\_.split('\n')

        self.font = font

        self.dim = (680, len(lines) \* 22 + 12)

        self.pos = (0.5 \* width - 0.5 \* self.dim[0], 0.5 \* height - 0.5 \* self.dim[1])

        self.seconds\_left = 0

        self.surface = pygame.Surface(self.dim)

        self.surface.fill((0, 0, 0, 0))

        for i, line in enumerate(lines):

            text\_texture = self.font.render(line, True, (255, 255, 255))

            self.surface.blit(text\_texture, (22, i \* 22))

            self.\_render = False

        self.surface.set\_alpha(220)

    def toggle(self):

        """Toggle on or off the render help"""

        self.\_render = not self.\_render

    def render(self, display):

        """Render help text method"""

        if self.\_render:

            display.blit(self.surface, self.pos)

# ==============================================================================

# -- CollisionSensor -----------------------------------------------------------

# ==============================================================================

class CollisionSensor(object):

    """ Class for collision sensors"""

    def \_\_init\_\_(self, parent\_actor, hud):

        """Constructor method"""

        self.sensor = None

        self.history = []

        self.\_parent = parent\_actor

        self.hud = hud

        world = self.\_parent.get\_world()

        blueprint = world.get\_blueprint\_library().find('sensor.other.collision')

        self.sensor = world.spawn\_actor(blueprint, carla.Transform(), attach\_to=self.\_parent)

        # We need to pass the lambda a weak reference to

        # self to avoid circular reference.

        weak\_self = weakref.ref(self)

        self.sensor.listen(lambda event: CollisionSensor.\_on\_collision(weak\_self, event))

    def get\_collision\_history(self):

        """Gets the history of collisions"""

        history = collections.defaultdict(int)

        for frame, intensity in self.history:

            history[frame] += intensity

        return history

    @staticmethod

    def \_on\_collision(weak\_self, event):

        """On collision method"""

        self = weak\_self()

        if not self:

            return

        actor\_type = get\_actor\_display\_name(event.other\_actor)

        self.hud.notification('Collision with %r' % actor\_type)

        impulse = event.normal\_impulse

        intensity = math.sqrt(impulse.x \*\* 2 + impulse.y \*\* 2 + impulse.z \*\* 2)

        self.history.append((event.frame, intensity))

        if len(self.history) > 4000:

            self.history.pop(0)

# ==============================================================================

# -- LaneInvasionSensor --------------------------------------------------------

# ==============================================================================

class LaneInvasionSensor(object):

    """Class for lane invasion sensors"""

    def \_\_init\_\_(self, parent\_actor, hud):

        """Constructor method"""

        self.sensor = None

        self.\_parent = parent\_actor

        self.hud = hud

        world = self.\_parent.get\_world()

        bp = world.get\_blueprint\_library().find('sensor.other.lane\_invasion')

        self.sensor = world.spawn\_actor(bp, carla.Transform(), attach\_to=self.\_parent)

        # We need to pass the lambda a weak reference to self to avoid circular

        # reference.

        weak\_self = weakref.ref(self)

        self.sensor.listen(lambda event: LaneInvasionSensor.\_on\_invasion(weak\_self, event))

    @staticmethod

    def \_on\_invasion(weak\_self, event):

        """On invasion method"""

        self = weak\_self()

        if not self:

            return

        lane\_types = set(x.type for x in event.crossed\_lane\_markings)

        text = ['%r' % str(x).split()[-1] for x in lane\_types]

        self.hud.notification('Crossed line %s' % ' and '.join(text))

# ==============================================================================

# -- GnssSensor --------------------------------------------------------

# ==============================================================================

class GnssSensor(object):

    """ Class for GNSS sensors"""

    def \_\_init\_\_(self, parent\_actor):

        """Constructor method"""

        self.sensor = None

        self.\_parent = parent\_actor

        self.lat = 0.0

        self.lon = 0.0

        world = self.\_parent.get\_world()

        blueprint = world.get\_blueprint\_library().find('sensor.other.gnss')

        self.sensor = world.spawn\_actor(blueprint, carla.Transform(carla.Location(x=1.0, z=2.8)),

                                        attach\_to=self.\_parent)

        # We need to pass the lambda a weak reference to

        # self to avoid circular reference.

        weak\_self = weakref.ref(self)

        self.sensor.listen(lambda event: GnssSensor.\_on\_gnss\_event(weak\_self, event))

    @staticmethod

    def \_on\_gnss\_event(weak\_self, event):

        """GNSS method"""

        self = weak\_self()

        if not self:

            return

        self.lat = event.latitude

        self.lon = event.longitude

# ==============================================================================

# -- CameraManager -------------------------------------------------------------

# ==============================================================================

class CameraManager(object):

    """ Class for camera management"""

    def \_\_init\_\_(self, parent\_actor, hud):

        """Constructor method"""

        self.sensor = None

        self.surface = None

        self.\_parent = parent\_actor

        self.hud = hud

        self.recording = False

        bound\_y = 0.5 + self.\_parent.bounding\_box.extent.y

        attachment = carla.AttachmentType

        self.\_camera\_transforms = [

            (carla.Transform(

                carla.Location(x=-5.5, z=2.5), carla.Rotation(pitch=8.0)), attachment.SpringArm),

            (carla.Transform(

                carla.Location(x=1.6, z=1.7)), attachment.Rigid),

            (carla.Transform(

                carla.Location(x=5.5, y=1.5, z=1.5)), attachment.SpringArm),

            (carla.Transform(

                carla.Location(x=-8.0, z=6.0), carla.Rotation(pitch=6.0)), attachment.SpringArm),

            (carla.Transform(

                carla.Location(x=-1, y=-bound\_y, z=0.5)), attachment.Rigid)]

        self.transform\_index = 1

        self.sensors = [

            ['sensor.camera.rgb', cc.Raw, 'Camera RGB'],

            ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)'],

            ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)'],

            ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic Gray Scale)'],

            ['sensor.camera.semantic\_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)'],

            ['sensor.camera.semantic\_segmentation', cc.CityScapesPalette,

             'Camera Semantic Segmentation (CityScapes Palette)'],

            ['sensor.lidar.ray\_cast', None, 'Lidar (Ray-Cast)']]

        world = self.\_parent.get\_world()

        bp\_library = world.get\_blueprint\_library()

        for item in self.sensors:

            blp = bp\_library.find(item[0])

            if item[0].startswith('sensor.camera'):

                blp.set\_attribute('image\_size\_x', str(hud.dim[0]))

                blp.set\_attribute('image\_size\_y', str(hud.dim[1]))

            elif item[0].startswith('sensor.lidar'):

                blp.set\_attribute('range', '50')

            item.append(blp)

        self.index = None

    def toggle\_camera(self):

        """Activate a camera"""

        self.transform\_index = (self.transform\_index + 1) % len(self.\_camera\_transforms)

        self.set\_sensor(self.index, notify=False, force\_respawn=True)

    def set\_sensor(self, index, notify=True, force\_respawn=False):

        """Set a sensor"""

        index = index % len(self.sensors)

        needs\_respawn = True if self.index is None else (

            force\_respawn or (self.sensors[index][0] != self.sensors[self.index][0]))

        if needs\_respawn:

            if self.sensor is not None:

                self.sensor.destroy()

                self.surface = None

            self.sensor = self.\_parent.get\_world().spawn\_actor(

                self.sensors[index][-1],

                self.\_camera\_transforms[self.transform\_index][0],

                attach\_to=self.\_parent,

                attachment\_type=self.\_camera\_transforms[self.transform\_index][1])

            # We need to pass the lambda a weak reference to

            # self to avoid circular reference.

            weak\_self = weakref.ref(self)

            self.sensor.listen(lambda image: CameraManager.\_parse\_image(weak\_self, image))

        if notify:

            self.hud.notification(self.sensors[index][2])

        self.index = index

    def next\_sensor(self):

        """Get the next sensor"""

        self.set\_sensor(self.index + 1)

    def toggle\_recording(self):

        """Toggle recording on or off"""

        self.recording = not self.recording

        self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

    def render(self, display):

        """Render method"""

        if self.surface is not None:

            display.blit(self.surface, (0, 0))

    @staticmethod

    def \_parse\_image(weak\_self, image):

        self = weak\_self()

        if not self:

            return

        if self.sensors[self.index][0].startswith('sensor.lidar'):

            points = np.frombuffer(image.raw\_data, dtype=np.dtype('f4'))

            points = np.reshape(points, (int(points.shape[0] / 4), 4))

            lidar\_data = np.array(points[:, :2])

            lidar\_data \*= min(self.hud.dim) / 100.0

            lidar\_data += (0.5 \* self.hud.dim[0], 0.5 \* self.hud.dim[1])

            lidar\_data = np.fabs(lidar\_data)  # pylint: disable=assignment-from-no-return

            lidar\_data = lidar\_data.astype(np.int32)

            lidar\_data = np.reshape(lidar\_data, (-1, 2))

            lidar\_img\_size = (self.hud.dim[0], self.hud.dim[1], 3)

            lidar\_img = np.zeros(lidar\_img\_size)

            lidar\_img[tuple(lidar\_data.T)] = (255, 255, 255)

            self.surface = pygame.surfarray.make\_surface(lidar\_img)

        else:

            image.convert(self.sensors[self.index][1])

            array = np.frombuffer(image.raw\_data, dtype=np.dtype("uint8"))

            array = np.reshape(array, (image.height, image.width, 4))

            array = array[:, :, :3]

            array = array[:, :, ::-1]

            self.surface = pygame.surfarray.make\_surface(array.swapaxes(0, 1))

        if self.recording:

            image.save\_to\_disk('\_out/%08d' % image.frame)

# ==============================================================================

# -- Game Loop ---------------------------------------------------------

# ==============================================================================

global xscr

xscr = 0

def sub\_callback(msg):

    global xscr

    xscr = msg.data

    #print(xscr)

def run\_step():

    global xscr

    control = carla.VehicleControl()

    control.steer = 0.0

    print(xscr)

    control.throttle = xscr

    control.brake = 0.0

    control.hand\_brake = False

    control.manual\_gear\_shift = False

    control.reverse = False

    return control

def game\_loop(args):

    """

    Main loop of the simulation. It handles updating all the HUD information,

    ticking the agent and, if needed, the world.

    """

    pygame.init()

    pygame.font.init()

    world = None

    try:

        if args.seed:

            random.seed(args.seed)

        client = carla.Client(args.host, args.port)

        client.set\_timeout(4.0)

        traffic\_manager = client.get\_trafficmanager()

        sim\_world = client.get\_world()

        if args.sync:

            settings = sim\_world.get\_settings()

            settings.synchronous\_mode = True

            settings.fixed\_delta\_seconds = 0.05

            sim\_world.apply\_settings(settings)

            traffic\_manager.set\_synchronous\_mode(True)

        display = pygame.display.set\_mode(

            (args.width, args.height),

            pygame.HWSURFACE | pygame.DOUBLEBUF)

        hud = HUD(args.width, args.height)

        world = World(client, client.get\_world(), hud, args)

        controller = KeyboardControl(world)

        if args.agent == "Basic":

            agent = BasicAgent(world.player)

        else:

            agent = BehaviorAgent(world.player, behavior=args.behavior)

        clock = pygame.time.Clock()

        while True:

            clock.tick()

            if args.sync:

                world.world.tick()

            else:

                world.world.wait\_for\_tick()

            if controller.parse\_events():

                return

            world.tick(clock)

            world.render(display)

            pygame.display.flip()

            # if agent.done():

            #     if args.loop:

            #         agent.set\_destination(random.choice(spawn\_points).location)

            #         world.hud.notification("The target has been reached, searching for another target", seconds=4.0)

            #         print("The target has been reached, searching for another target")

            #     else:

            #         print("The target has been reached, stopping the simulation")

            #         break

            control = run\_step()  #agent.run\_step()

            world.player.apply\_control(control)

    finally:

        if world is not None:

            settings = world.world.get\_settings()

            settings.synchronous\_mode = False

            settings.fixed\_delta\_seconds = None

            world.world.apply\_settings(settings)

            traffic\_manager.set\_synchronous\_mode(True)

            world.destroy()

        pygame.quit()

# ==============================================================================

# -- main() --------------------------------------------------------------

# ==============================================================================

def main():

    """Main method"""

    argparser = argparse.ArgumentParser(

        description='CARLA Automatic Control Client')

    argparser.add\_argument(

        '-v', '--verbose',

        action='store\_true',

        dest='debug',

        help='Print debug information')

    argparser.add\_argument(

        '--host',

        metavar='H',

        default='127.0.0.1',

        help='IP of the host server (default: 127.0.0.1)')

    argparser.add\_argument(

        '-p', '--port',

        metavar='P',

        default=2000,

        type=int,

        help='TCP port to listen to (default: 2000)')

    argparser.add\_argument(

        '--res',

        metavar='WIDTHxHEIGHT',

        default='1280x720',

        help='Window resolution (default: 1280x720)')

    argparser.add\_argument(

        '--sync',

        action='store\_true',

        help='Synchronous mode execution')

    argparser.add\_argument(

        '--filter',

        metavar='PATTERN',

        default='vehicle.\*',

        help='Actor filter (default: "vehicle.\*")')

    argparser.add\_argument(

        '-l', '--loop',

        action='store\_true',

        dest='loop',

        help='Sets a new random destination upon reaching the previous one (default: False)')

    argparser.add\_argument(

        "-a", "--agent", type=str,

        choices=["Behavior", "Basic"],

        help="select which agent to run",

        default="Behavior")

    argparser.add\_argument(

        '-b', '--behavior', type=str,

        choices=["cautious", "normal", "aggressive"],

        help='Choose one of the possible agent behaviors (default: normal) ',

        default='normal')

    argparser.add\_argument(

        '-s', '--seed',

        help='Set seed for repeating executions (default: None)',

        default=None,

        type=int)

    args = argparser.parse\_args()

    args.width, args.height = [int(x) for x in args.res.split('x')]

    log\_level = logging.DEBUG if args.debug else logging.INFO

    logging.basicConfig(format='%(levelname)s: %(message)s', level=log\_level)

    logging.info('listening to server %s:%s', args.host, args.port)

    print(\_\_doc\_\_)

    try:

        game\_loop(args)

    except KeyboardInterrupt:

        print('\nCancelled by user. Bye!')

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

    rospy.init\_node('listener', anonymous=True)

    rospy.Subscriber('/out\_value', Float32, sub\_callback)

    main()

Diagram

Description automatically generated

#!/bin/sh

sudo docker run -d -p 2000-2002:2000-2002 --gpus all -e NVIDIA\_VISIBLE\_DEVICES=0 carlasim/carla:0.9.13 /bin/bash CarlaUE4.sh -RenderOffScreen -quality-level=Low