# Carla学习记录

## 基础知识

### Client

Carla是一个bs架构，服务端负责渲染，客户端负责获取服务端的渲染结果；因此显而易见，各类API的调用都会出现同步/异步问题；

host = '10.10.32.21'  
port=2000  
client = carla.Client(host,port)  
client.set\_timeout(2.0)

### World

可以理解为所有渲染场景的容器，所有的actor的添加，销毁，获取入口

world = client.get\_world()

### Settings

settings = world.get\_settings()

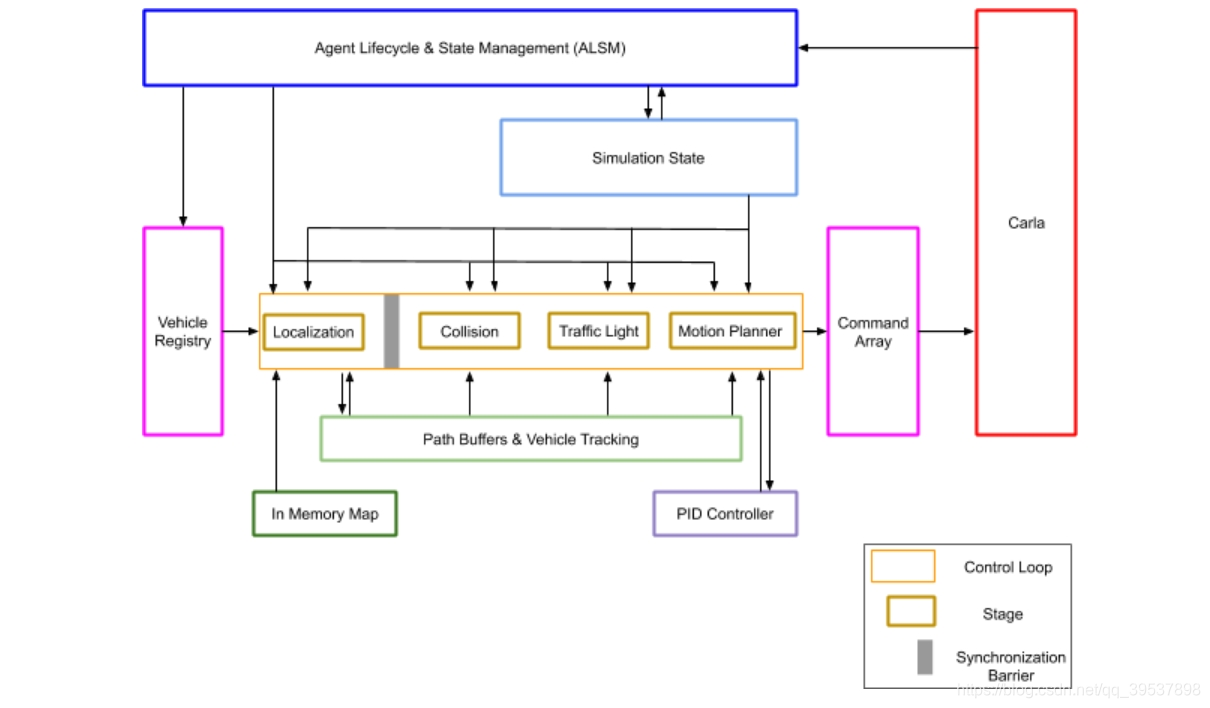
获取整个世界的设置；

settings.synchronous\_mode = True  
settings.fixed\_delta\_seconds = 0.05

world.apply\_settings(settings)

### traffic\_manager（可以声明多个）

<https://blog.csdn.net/qq_39537898/article/details/117562578>



所有交通对象的管理类，车辆，行人都由此获取

traffic\_manager = client.get\_trafficmanager(8000)

traffic\_manager.set\_synchronous\_mode(True) 设置为同步模式，并且world的设置也要设置为同步，否则碰撞计算就会全部错误；

traffic\_manager.set\_global\_distance\_to\_leading\_vehicle(2.5) 设置与前车的距离

注册：****Vehicle Registry****

计算：****Localization Stage，**** ****Collision Stage.Traffic Light Stage.Motion Planner.****

### 获取相应的蓝图类

bps = world.get\_blueprint\_library().filter(filter)

获取类之后，即可用已存在的类进行实例的创建

已知的filter：

VEHICLE='vehicle.\*'  
WALKER="walker.pedestrian.\*"

### 获取可放置车辆的点位（带transform）

spawn\_points = world.get\_map().get\_spawn\_points()

### SpawnActor

第一个参数为对象类型，第二个参数为放置的位置

初始化一个蓝图类SpawnActor(blueprint, transform)

### 初始化设置车辆

设置车辆的颜色，id，是否为hero，添加自动模式

blueprint = random.choice(blueprints)  
if blueprint.has\_attribute('color'):  
 color = random.choice(blueprint.get\_attribute('color').recommended\_values)  
 blueprint.set\_attribute('color', color)  
if blueprint.has\_attribute('driver\_id'):  
 driver\_id = random.choice(blueprint.get\_attribute('driver\_id').recommended\_values)  
 blueprint.set\_attribute('driver\_id', driver\_id)  
if hero:  
 #如果是hero代表为主控，否则为自动驾驶的车辆  
 blueprint.set\_attribute('role\_name', 'hero')  
 hero = False  
else:  
 blueprint.set\_attribute('role\_name', 'autopilot')  
 # spawn the cars and set their autopilot and light state all together  
item = SpawnActor(blueprint, transform)  
 .then(SetAutopilot(FutureActor, True, traffic\_manager.get\_port()))  
batch.append(item )

SetAutopilot

设置自动驾驶模式；以spawnActor的模式实例化，其中第一个参数为蓝图类，第二个参数为transform；

### 开启/关闭车辆的自动驾驶

SetAutopilot(FutureActor, True, traffic\_manager.get\_port())

### 批量添加车辆至场景中（同步）

for response in client.apply\_batch\_sync(batch, synchronous\_master):  
 if response.error:  
 logging.error(response.error)  
 else:  
 vehicles\_list.append(response.actor\_id)

### 添加自行走行人（必须和aiworker关联使用）

walker绑定了一个父级；

SpawnActor(walker\_controller\_bp, carla.Transform(), walkers\_list[i]["id"]

因此实现的逻辑为：

首先获取出walker中的行人类：

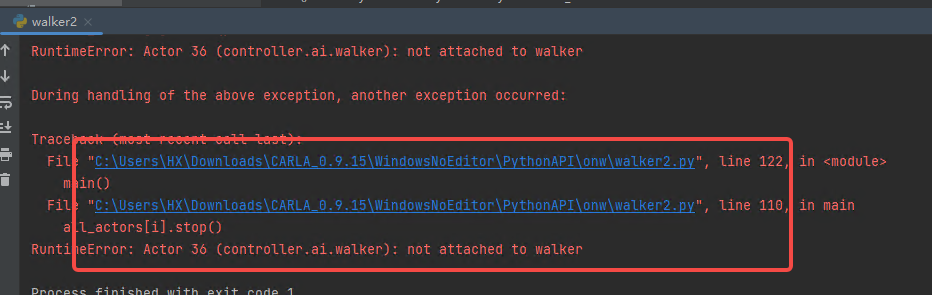
然后循环实例化，使用声明式transform设置其几何变换，地图随机选取点；

批量添加；

然后创建同样数量的aiwalker类，父级设定为人

再批量添加；

最后启动ai walker的行走即可；



### Weather

### Camera

获取相机蓝图类；

设置相机成像的宽高，设置fov，设置transform

camera\_bp = world.get\_blueprint\_library().find('sensor.camera.rgb')  
camera\_bp.set\_attribute("image\_size\_x", str(args.width))  
camera\_bp.set\_attribute("image\_size\_y", str(args.height))  
camera\_bp.set\_attribute("fov", str(args.fov))  
camera = world.spawn\_actor(camera\_bp, carla.Transform())

其后，如添加自走行人的逻辑一直；添加进入场景

world.set\_pedestrians\_seed(1235)  
ped\_bp = random.choice(world.get\_blueprint\_library().filter("walker.pedestrian.\*"))  
trans = carla.Transform()  
trans.location = world.get\_random\_location\_from\_navigation()  
ped = world.spawn\_actor(ped\_bp, trans)  
walker\_controller\_bp = world.get\_blueprint\_library().find('controller.ai.walker')  
controller = world.spawn\_actor(walker\_controller\_bp, carla.Transform(), ped)  
controller.start()  
controller.go\_to\_location(world.get\_random\_location\_from\_navigation())  
controller.set\_max\_speed(1.7)  
  
# keep tracking of actors to remove  
actor\_list.append(camera)  
actor\_list.append(ped)  
actor\_list.append(controller)

### 传感器观察类（用于场景返回数据）

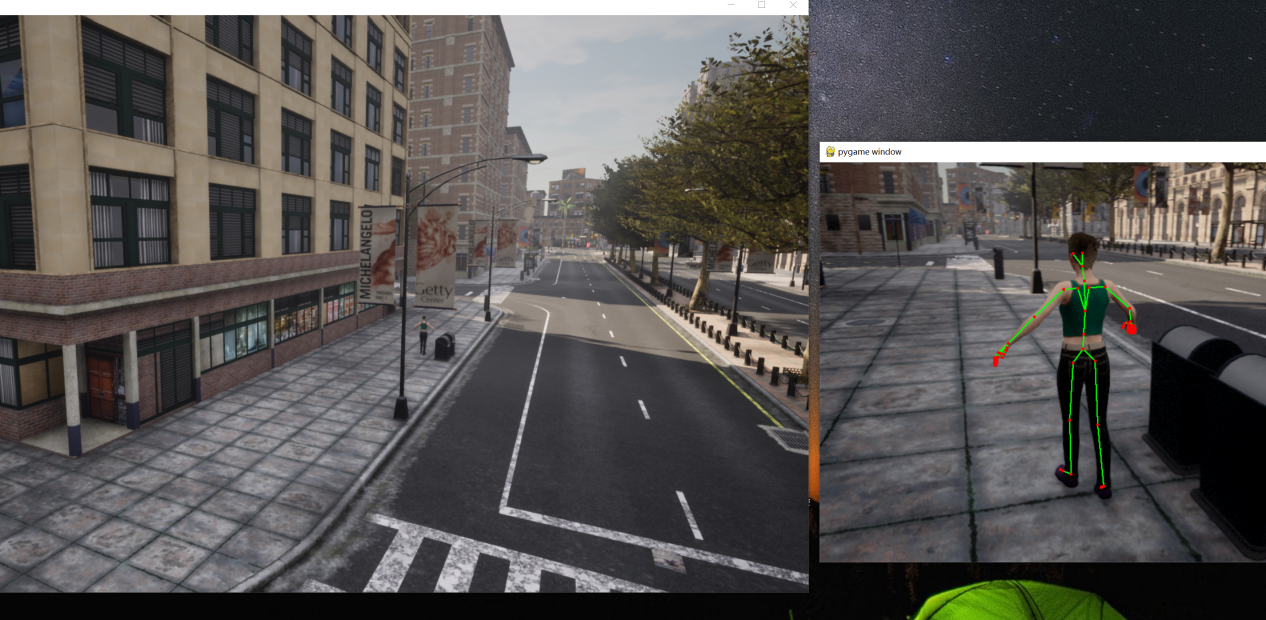
获取传感器数据，实际上就是sensor\_queue.get；

def \_retrieve\_data(self, sensor\_queue, timeout):  
 while True:  
 data = sensor\_queue.get(timeout=timeout)  
 if data.frame == self.frame:  
 return data  
# ---------------

Sensor\_queue是一个每帧渲染返回的数据

def make\_queue(register\_event):  
 q = queue.Queue()  
 register\_event(q.put)  
 self.\_queues.append(q)  
  
make\_queue(self.world.on\_tick)

### 相机跟随actor

# move camera around  
trans = ped.get\_transform()  
x = math.cos(turning) \* -3  
y = math.sin(turning) \* 3  
trans.location.x += x  
trans.location.y += y  
trans.location.z = 2  
trans.rotation.pitch = -16  
trans.rotation.roll = 0  
trans.rotation.yaw = -360 \* (turning/(math.pi\*2))  
camera.set\_transform(trans)

图像数据转array

def get\_image\_as\_array(image):  
 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]  
 # make the array writeable doing a deep copy  
 array2 = copy.deepcopy(array)  
 return array2

### 世界坐标转屏幕坐标

投影矩阵

def build\_projection\_matrix(w, h, fov):  
 focal = w / (2.0 \* np.tan(fov \* np.pi / 360.0))  
 K = np.identity(3)  
 K[0, 0] = K[1, 1] = focal  
 K[0, 2] = w / 2.0  
 K[1, 2] = h / 2.0  
 return K

get\_screen\_points(camera, K, image\_w, image\_h, points)

def get\_screen\_points(camera, K, image\_w, image\_h, points3d):  
   
 # get 4x4 matrix to transform points from world to camera coordinates  
 world\_2\_camera = np.array(camera.get\_transform().get\_inverse\_matrix())  
  
 # build the points array in numpy format as (x, y, z, 1) to be operable with a 4x4 matrix  
 points\_temp = []  
 for p in points3d:  
 points\_temp += [p.x, p.y, p.z, 1]  
 points = np.array(points\_temp).reshape(-1, 4).T  
   
 # convert world points to camera space  
 points\_camera = np.dot(world\_2\_camera, points)  
   
 # New we must change from UE4's coordinate system to an "standard"  
 # (x, y ,z) -> (y, -z, x)  
 # and we remove the fourth component also  
 points = np.array([  
 points\_camera[1],  
 points\_camera[2] \* -1,  
 points\_camera[0]])  
   
 # Finally we can use our K matrix to do the actual 3D -> 2D.  
 points\_2d = np.dot(K, points)  
  
 # normalize the values and transpose  
 points\_2d = np.array([  
 points\_2d[0, :] / points\_2d[2, :],  
 points\_2d[1, :] / points\_2d[2, :],  
 points\_2d[2, :]]).T  
  
 return points\_2d

### 初始化Transform方式

camera\_transform = carla.Transform(carla.Location(x=-5.5, z=2.8), carla.Rotation(pitch=-15))

### 添加lidar传感器

lidar\_bp = bp\_lib.filter("sensor.lidar.ray\_cast")[0]

if args.no\_noise:  
 lidar\_bp.set\_attribute('dropoff\_general\_rate', '0.0')  
 lidar\_bp.set\_attribute('dropoff\_intensity\_limit', '1.0')  
 lidar\_bp.set\_attribute('dropoff\_zero\_intensity', '0.0')

#lidar的fov上顶30  
lidar\_bp.set\_attribute('upper\_fov', str(args.upper\_fov))

#lidar的fov下限-25  
lidar\_bp.set\_attribute('lower\_fov', str(args.lower\_fov))

#lidar线束  
lidar\_bp.set\_attribute('channels', str(args.channels))  
lidar\_bp.set\_attribute('range', str(args.range))

#每秒发射的点云数量，10万个  
lidar\_bp.set\_attribute('points\_per\_second', str(args.points\_per\_second))

## 进阶

### Lidar\_to\_camera

初始化camera，vehicle，lidar

获取图像的输出

im\_array = np.copy(np.frombuffer(image\_data.raw\_data, dtype=np.dtype("uint8")))  
im\_array = np.reshape(im\_array, (image\_data.height, image\_data.width, 4))  
im\_array = im\_array[:, :, :3][:, :, ::-1]

获取lidar的输出

p\_cloud\_size = len(lidar\_data)  
p\_cloud = np.copy(np.frombuffer(lidar\_data.raw\_data, dtype=np.dtype('f4')))  
p\_cloud = np.reshape(p\_cloud, (p\_cloud\_size, 4))

intensity = np.array(p\_cloud[:, 3])  
  
# Point cloud in lidar sensor space array of shape (3, p\_cloud\_size).  
local\_lidar\_points = np.array(p\_cloud[:, :3]).T

# Add an extra 1.0 at the end of each 3d point so it becomes of  
# shape (4, p\_cloud\_size) and it can be multiplied by a (4, 4) matrix.  
local\_lidar\_points = np.r\_[  
 local\_lidar\_points, [np.ones(local\_lidar\_points.shape[1])]]  
  
# This (4, 4) matrix transforms the points from lidar space to world space.  
lidar\_2\_world = lidar.get\_transform().get\_matrix()

Lidar的局部坐标转换为世界坐标

world\_points = np.dot(lidar\_2\_world, local\_lidar\_points)

将lidar的世界坐标转换到图像坐标

world\_2\_camera = np.array(camera.get\_transform().get\_inverse\_matrix())  
  
# Transform the points from world space to camera space.  
sensor\_points = np.dot(world\_2\_camera, world\_points)

### 物理引擎

physics\_vehicle = vehicle.get\_physics\_control()

### 传感器安装

#### 初始化车辆

bp = world.get\_blueprint\_library().filter('charger\_2020')[0]  
vehicle = world.spawn\_actor(bp, random.choice(world.get\_map().get\_spawn\_points()))  
vehicle\_list.append(vehicle)  
vehicle.set\_autopilot(True)

#### DisPlayManager()

self.display = pygame.display.set\_mode(window\_size, pygame.HWSURFACE | pygame.DOUBLEBUF)

#### SensorManager

传感器类

传入world，dm，传感器的类型，放置的位置，看起来仿真中所有的传感器都可以放置到同一个位置，就不存在各个坐标系转换的问题，但是观察到的内容是如何变化的呢？

SensorManager(world, display\_manager, 'RGBCamera', carla.Transform(carla.Location(x=0, z=2.4), carla.Rotation(yaw=-90)),   
 vehicle, {}, display\_pos=[0, 0])

#### Init初始化构造，事件监听，并键入到displaymanager中；

实现render函数；

def render(self):  
 if self.surface is not None:  
 offset = self.display\_man.get\_display\_offset(self.display\_pos)  
 self.display\_man.display.blit(self.surface, offset)

sufface是将渲染出来的结果处理为pygame可渲染的对象

def save\_rgb\_image(self, image):  
 t\_start = self.timer.time()  
  
 image.convert(carla.ColorConverter.Raw)  
 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]  
  
 if self.display\_man.render\_enabled():  
 self.surface = pygame.surfarray.make\_surface(array.swapaxes(0, 1))

其他函数，如save image，lidar，semantilidar\_image,lidar\_image

### Gbuffer（当前报错-后续查看）

### 根据id获取蓝图中的实体类对象

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

### 跟随actor漫游模式

spectator = world.get\_spectator()

transform = vehicle.get\_transform()

spectator.set\_transform(carla.Transform(transform.location + carla.Location(z=50),

carla.Rotation(yaw=int(transform.rotation.yaw), roll=0, pitch=-90)))

### 绘制几何体

debugHelper =world.debug

debugHelper.draw\_line(carla.Location(0,0,0),carla.Location(100,100,0))

可以draw arrow point,box等