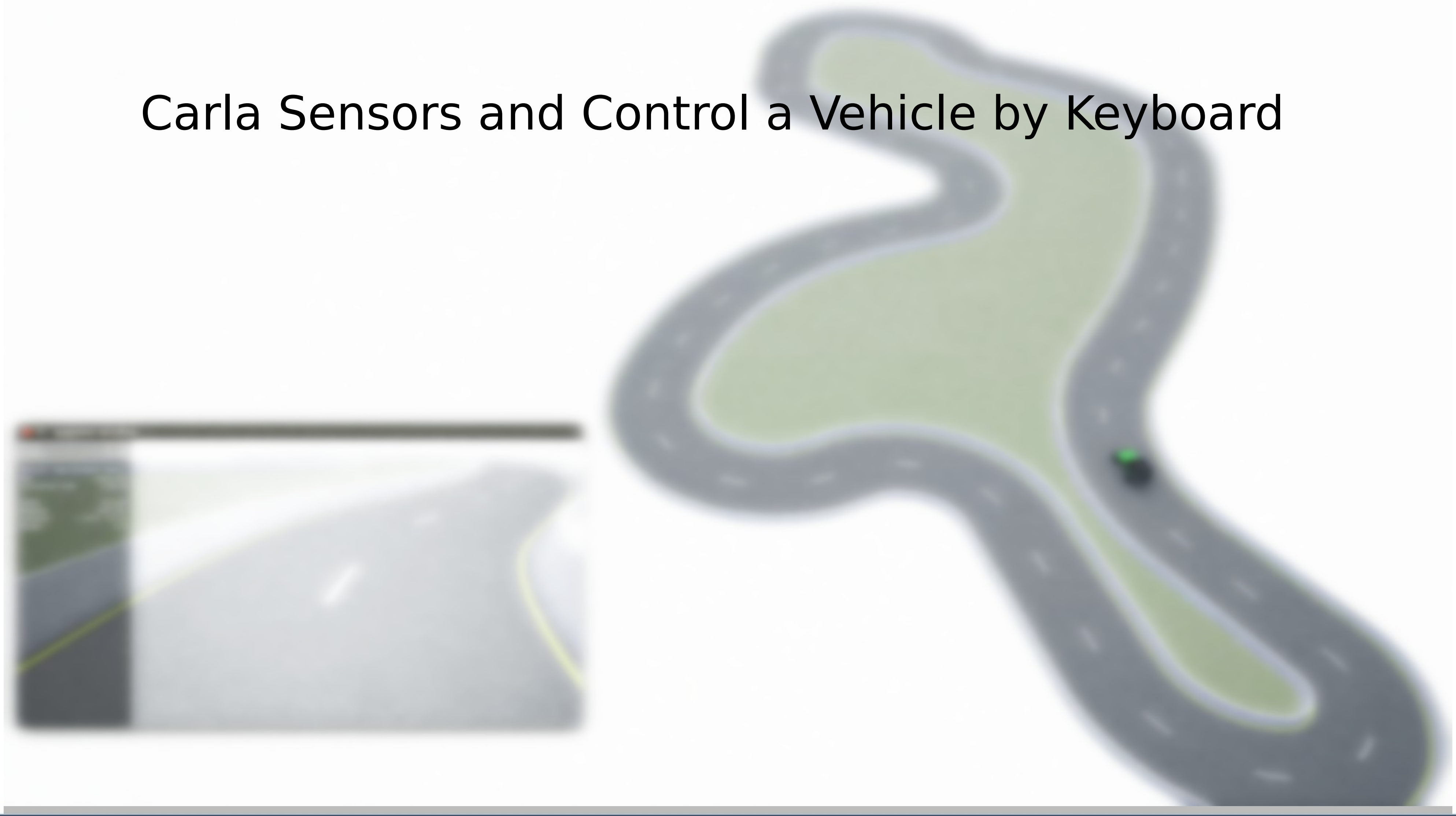
Carla Sensors and Control a Vehicle by Keyboard



Course contents

1. Carla RGB-camera Sensor
2. Time step (Frame per second FPS)
3. Synchronous and Asynchronous
4. Control a Vehicle by Keyboard
5. Sensors: Depth-camera, collision, GNSS …

1. Carla RGB-camera Sensor

- Use Carla Doc for sensors

CARLA Simulator

Search docs

Home

GETTING STARTED

Introduction

Quick start installation

BUILDING CARLA

Linux build

Windows build

Update CARLA

Build system

Running in a Docker

F.A.Q.

FIRST STEPS

Docs » References » Sensors reference

Sensors reference

- Collision detector
- Depth camera
- GNSS sensor
- IMU sensor
- Lane invasion detector
- LIDAR sensor
- Obstacle detector
- Radar sensor
- RGB camera
- RSS sensor
- Semantic LIDAR sensor
- Semantic segmentation camera
- DVS camera

RGB camera

- **Blueprint:** sensor.camera.rgb
- **Output:** carla.Image per step (unless `sensor_tick` says otherwise)..

The "RGB" camera acts as a regular camera capturing images from the scene. carla.colorConverter

If `enable_postprocess_effects` is enabled, a set of post-process effects is applied to the image for the sake of realism:

- **Vignette:** darkens the border of the screen.
- **Grain jitter:** adds some noise to the render.
- **Bloom:** intense lights burn the area around them.
- **Auto exposure:** modifies the image gamma to simulate the eye adaptation to darker or brighter areas.
- **Lens flares:** simulates the reflection of bright objects on the lens.
- **Depth of field:** blurs objects near or very far away of the camera.

The `sensor_tick` tells how fast we want the sensor to capture the data. A value of 1.5 means that we want the sensor to capture data each second and a half. By default a value of 0.0 means as fast as possible.

1. Carla RGB-camera Sensor

- Create RGB-camera sensor

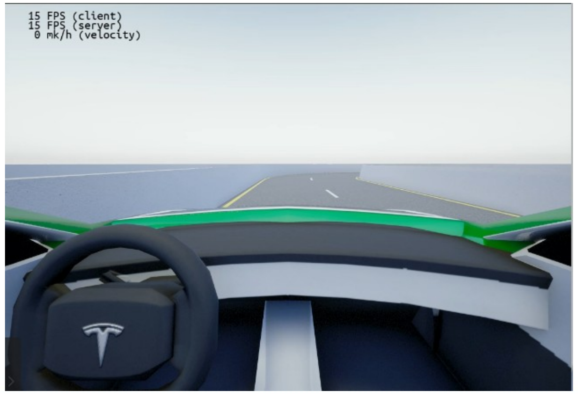
1. RGB-camera blueprint ;
2. Set the attribute of camera;
3. Add camera sensor to the vehicle;
4. Server listen to the data;
5. Add to actor list to destroy;

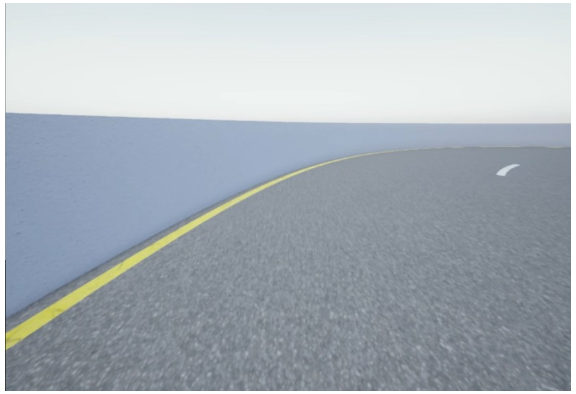
```
rgb_camera_bp = blueprint_library.find('sensor.camera.rgb')
```

```
rgb_camera_bp.set_attribute("image_size_x", %f"%(IMG_WIDTH))
rgb_camera_bp.set_attribute("image_size_y", "%f"%(IMG_HEIGHT))   # image height
rgb_camera_bp.set_attribute("fov", "110")          # Horizontal field of view in degrees
```

```
spawn_point = carla.Transform(carla.Location(x=2.5, y=0.0, z=1.0), carla.Rotation(pitch=-20.0, yaw=0.0, roll=0.0))
sensor = world.spawn_actor(rgb_camera_bp, spawn_point, attach_to=vehicle)
```

```
sensor.listen(lambda data: process_img(data))
```



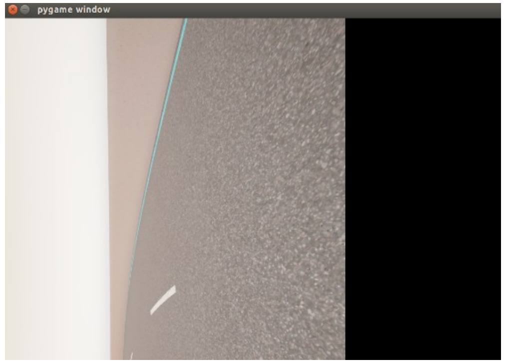


1. Carla RGB-camera Sensor

- RGB-camera sensor: listen to data

1. Raw image;
2. RGB channels;
3. Switch width and height;
4. Pygame shows the image;
5. Carla.SensorData->Carla.Image;

```python
def process_img(image):
    """
    process the image
    """
    global surface
    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] # switch r,g,b
    array = array.swapaxes(0, 1)  # exchange the width and height
    surface = pygame.surfarray.make_surface(array)   # Copy an array to a new surface
    # save the image
    if SHOW_CAM:
        cv2.imshow("RGB-image", array)
        cv2.imwrite('camera_3.png', array)
        cv2.waitKey(1)
```



Problems：

1. RGB-image Frame per second （FPS）？；
2. Server Carla simulator FPS？；
3. Server – Client Synchronous or Asynchronous ？

Process the image

2. Time step (Frame per second FPS)

- Client -- pygame

```
###################
# --- Running --- #
###################
pygame.init()
# Open a window on the screen
display  = pygame.display.set_mode([IMG_WIDTH, IMG_HEIGHT])
font = get_font()
# clock limits the frame
clock = pygame.time.Clock()

# server fps
world_fps = World_FPS()
world.on_tick(world_fps.on_server_tick)


while True:
    clock.tick_busy_loop(10)
    vehicle.apply_control(carla.VehicleControl(throttle=1.0, brake=0.0, steer=0.0))
    # show the fps
    display.blit(font.render('% 5d FPS (client)' % clock.get_fps(), True, (0, 0, 0)), (8, 10))
    display.blit(font.render('% 5d FPS (server)' % world_fps.server_fps, True, (0, 0, 0)), (8, 28))
    pygame.display.flip()  # update
    display.blit(surface, (0, 0))
```

1. Apply pygame clock

2. Tick 10 fps

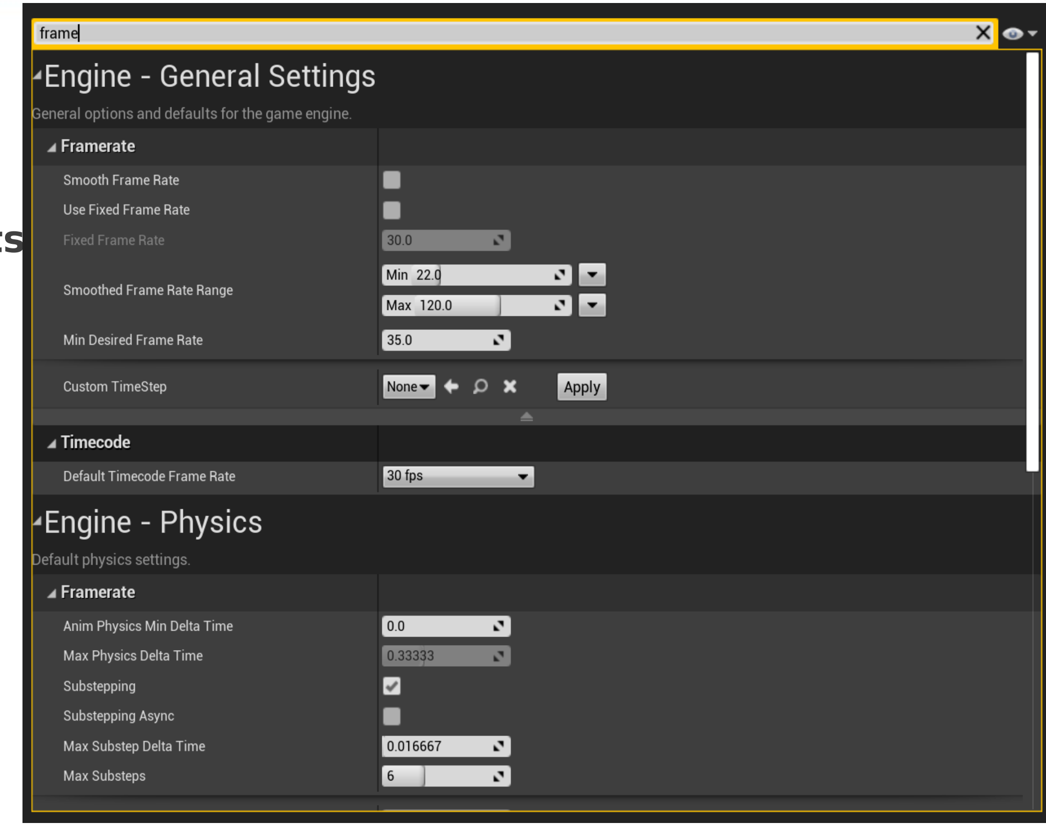
3. Highest FPS is up to your PC
4. Time step

2. Time step (Frame per second FPS)

- Server: Carla simulator

1. Simulation time-step
   ① The time span that went by between those **two simulation moments**
   ② Real time （client） and simulation time （server）；
   ③ Time-step can be **fixed** or **variable** depending on user preferences；
   ④ Limitation： If the time-step is greater than **0.1**,
   there will not be enough physical substeps.

2. Why simulation time-step
   ① Collect data；
   ② Simulation recording；



Time-step can be **fixed** or **variable** in UE4

3. Server – Client (**synchronous or asynchronous**)
   ① Sensor data: The simulator waits for the sensor data to be ready before sending the measurements.
   ② Control message: The simulator halts each frame until a control message is received.

3. Synchronous and Asynchronous

- Server: Carla simulator

Server(fps:15)/Client(fps:10)

Server(fps:15)/Client(fps:1)

Synchronous(fps:15)

1. By default, CARLA runs in **asynchronous mode**： The server runs the simulation as fast as possible, without waiting for the client.
2. **Synchronous mode**： the server waits for a client tick, a “ready to go” message, before updating to the following simulation step.
3. If the client is too slow and the server does not wait, there will be **an overflow of information**. many sensors and asynchrony, it would be impossible to know **if all the sensors are using data from the same moment in the simulation**.

3. Synchronous and Asynchronous

- Using synchronous mode

```python
##############################################
# --- Create a synchronous mode context ---#
##############################################
synchronous_fps = 15
with CarlaSyncMode(world, camera_rgb, fps=synchronous_fps) as sync_mode:

    while True:
        #  quit the while
        if should_quit():
            return

        # start clock
        clock.tick_busy_loop(synchronous_fps)
        # clock.tick(synchronous_fps)
        # Advance the simulation and wait for the data.
        snapshot, image_rgb = sync_mode.tick(timeout=2.0)

        # Control vehicle to move forward
        vehicle.apply_control(carla.VehicleControl(throttle=0.5, brake=0.0, steer=0.0))
```

1. Carla SyncMode class

```python
def __enter__(self):
    # some data about the simulation such as synchrony between client and server or rendering mode
    self._settings = self.world.get_settings()
    # ---- This is important carla.WorldSettings
    self.frame = self.world.apply_settings(carla.WorldSettings(
        no_rendering_mode=False,
        synchronous_mode=True,
        fixed_delta_seconds=self.delta_seconds))

    def make_queue(register_event):
        q = queue.Queue()
        register_event(q.put)
        self._queues.append(q)

    make_queue(self.world.on_tick)
    for sensor in self.sensors:
        make_queue(sensor.listen)
    return self
```

1.1 SyncMode

1.2 Queue

2. Client FPS

3. Sensor data

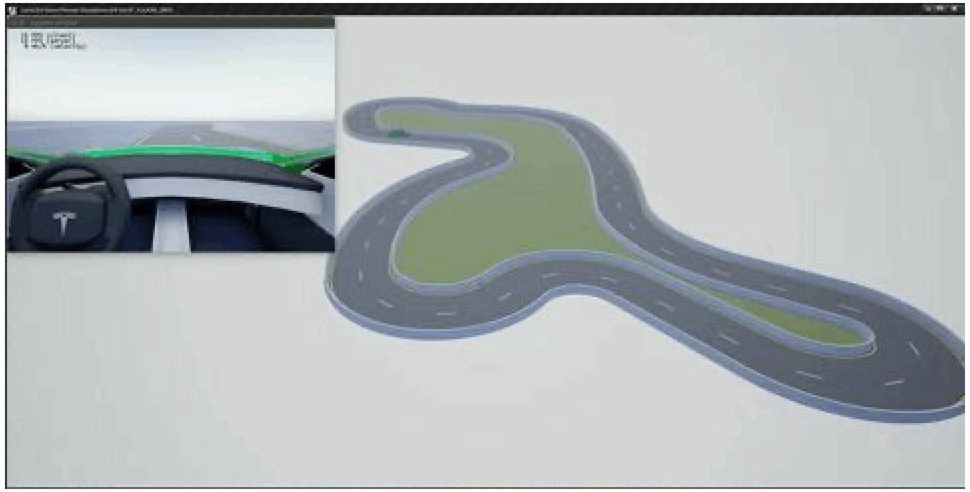
```python
def tick(self, timeout):
    # This method only has effect on synchronous mode, when both c
    # The client tells the server when to step to the next frame ar
    self.frame = self.world.tick()
    # get the data synchronous data
    data = [self._retrieve_data(q, timeout) for q in self._queues]
    assert all(x.frame == self.frame for x in data)
    return data
```

2.1 timestamp

4. Control a Vehicle by Keyboard

- Carla.VehicleControl and Pygame

1. Pygame keyboard event;
2. Carla.VehicleControl class;
3. Carla.Vehicle method apply_control;
4. Camera sensor position;

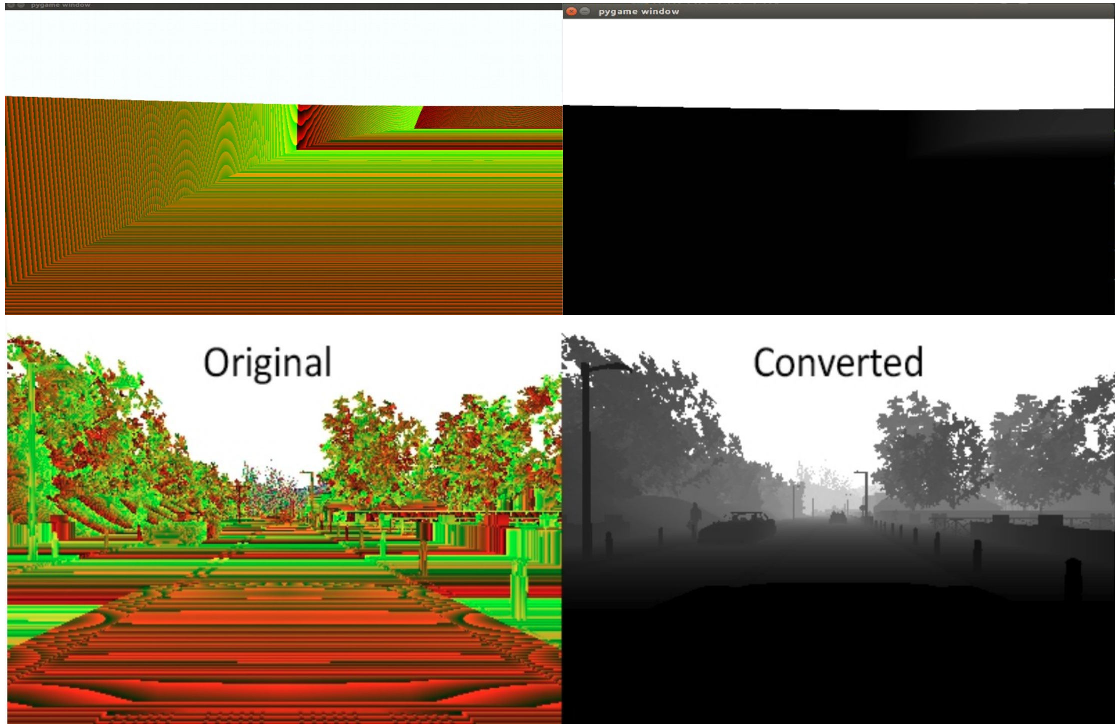


Camera sensor position

```python
class KeyboardControl(object):
    """Class that handles keyboard input."""
    def __init__(self, player):

        self._control = carla.VehicleControl()
        self._steer_cache = 0.0
        self.player = player


    def parse_events(self, clock):
        self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
        self.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        if  keys[K_w]:
            self._control.throttle = min(self._control.throttle + 0.01, 1)
        else:
            self._control.throttle = 0.0
            # fix the velocity
            # self._control.throttle = 0.40

        if keys[K_s]:
            self._control.brake = min(self._control.brake + 0.2, 1)
        else:
            self._control.brake = 0
```

- Depth Sensor

```
########################
# --- Depth sensor --- #
########################
depth_image_bp = blueprint_library.find("sensor.camera.depth")
depth_image_bp.set_attribute("image_size_x", "%f"%(IMG_WIDTH))
depth_image_bp.set_attribute("image_size_y", "%f"%(IMG_HEIGHT))
depth_image_bp.set_attribute("fov", "110")
depth_sensor = world.spawn_actor(depth_image_bp, spawn_point, attach_to=vehicle)
actor_list.append(depth_sensor)
depth_sensor.listen(lambda image: process_depth_img(image))
```




```
def process_depth_img(image):
    global depth_surface
    image.convert(carla.ColorConverter.LogarithmicDepth)          # ca
    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]
    depth_surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
    # save the image
    if SHOW_CAM:
        image_name = round(image.frame, 8)
        image.save_to_disk('./image/%d' % image_name)
```

- Collision detector

```
###########################
# --- Collision sensor --- #
###########################
collision_bp = blueprint_library.find('sensor.other.collision')
collision_sensor = world.spawn_actor(collision_bp, carla.Transform(), attach_to=vehicle)
actor_list.append(collision_sensor)
collision_sensor.listen(lambda event: process_collision(event))
```

```
def process_collision(event):
    """
    process collision
    """
    global collision_mark, cnts
    collision_mark = True
    cnts += 1
    print('Collision frame:  Collision times:', event.frame, cnts)
```

GNSS sensor and IMU sensor

GNSS sensor

- **Blueprint:** sensor.other.gnss
- **Output:** carla.GNSSMeasurement per step (unless `sensor_tick` says otherwise).

Reports current gnss position of its parent object. This is calculated by adding the metric position to an initial geo reference location defined within the OpenDRIVE map definition.

GNSS attributes

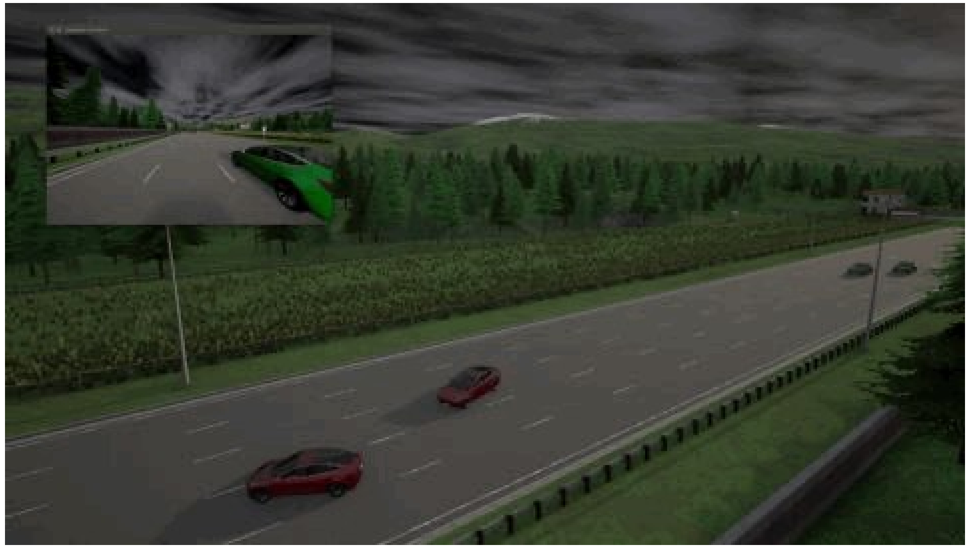
Blueprint attribute	Type	Default	Description
`noise_alt_bias`	float	0.0	Mean parameter in the noise model for altitude.
`noise_alt_stddev`	float	0.0	Standard deviation parameter in the noise model for altitude.
`noise_lat_bias`	float	0.0	Mean parameter in the noise model for latitude.
`noise_lat_stddev`	float	0.0	Standard deviation parameter in the noise model for latitude.
`noise_lon_bias`	float	0.0	Mean parameter in the noise model for longitude.
`noise_lon_stddev`	float	0.0	Standard deviation parameter in the noise model for longitude.
`noise_seed`	int	0	Initializer for a pseudorandom number generator.
`sensor_tick`	float	0.0	Simulation seconds between sensor captures (ticks).

Output attributes

Sensor data attribute	Type	Description
`frame`	int	Frame number when the measurement took place.
`timestamp`	double	Simulation time of the measurement in seconds since the beginning of the episode.
`transform`	carla.Transform	Location and rotation in world coordinates of the sensor at the time of the measurement.
`latitude`	double	Latitude of the actor.
`longitude`	double	Longitude of the actor.
`altitude`	double	Altitude of the actor.

5. Different Sensors

- Obstacle detector



Control distance

```
class ObstacleSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._history = []
        self._parent = parent_actor
        self._event_count = 0
        self.obstacle_distance = None
        self.sensor_transform = carla.Transform(carla.Location(x=1.6, z=1.7), carla.Rotation(yaw=0)) # Put this sensor on the windshield of the car.
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.obstacle')
        bp.set_attribute('distance', '5.0')      # sensor distance
        bp.set_attribute('hit_radius', '1.0')
        bp.set_attribute('only_dynamics', 'False')
        bp.set_attribute('debug_linetrace', 'False')
        bp.set_attribute('sensor_tick', '0.1')
        self.sensor = world.spawn_actor(bp, self.sensor_transform, attach_to=self._parent)
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: ObstacleSensor._process_event(weak_self, event))


    @staticmethod
    def _process_event(weak_self, event):
        self = weak_self()
        if not self:
            return
        if event.other_actor.type_id.startswith('vehicle.'):
            vehicle = event.other_actor.type_id
        self.obstacle_distance = event.distance
        print('------------------ Obstacle ---------------------')
        print ("Obstacle sensor Distance %f" % (self.obstacle_distance))
```

Carla Sensors and Control a Vehicle by Keyboard

Game over !