Mini Doc du simulateur CARLA pour construire un jeu de données synthétique de correspondance jour / nuit

Pierre Minier

Mai 2022

Sommaire

1 Les bases **1**
1.1 Une architecture client-serveur . . . 1
1.1.1 Interfaces . . . 1
1.1.2 Connexion . . . 1
1.1.3 Principe . . . 1
1.2 Les Acteurs . . . 1
1.2.1 Définir un acteur . . . 1
1.2.2 Ajouter d'un acteur à la simulation . . . 1
1.2.3 Précautions . . . 1
1.3 Les cartes . . . 2

2 La luminosité **2**
2.1 La météo . . . 2
2.2 Le jour et la nuit . . . 2

3 Les caméras **2**
3.1 Initialiser une caméra . . . 2
3.2 Attacher une caméra à une voiture . . . 3
3.2.1 Les 6 degrés de libertés pour placer la caméra . . . 3
3.3 Enregistrer les images prises . . . 3

4 Synchroniser les caméras **4**
4.1 Récupérer et modifier les paramètres du serveur . . . 4
4.2 Mise en place . . . 4
4.2.1 La classe Queue de Python . . . 5
4.2.2 Fonction Lambda . . . 5
4.2.3 Boucle principale . . . 5

5 Simulation déterministe **5**
5.1 Traffic Manager . . . 5
5.2 Les piétons . . . 5

6 Certifier la correspondance jour/nuit **5**

1 Les bases

1.1 Une architecture client-serveur

1.1.1 Interfaces

La simulation comporte deux interfaces:

- Le serveur (*World*)
- Le client (*Client*)

Le serveur se lance avec le script CarlaUE4.sh et un client avec un script python. On travaille du côté client dans ce projet.

1.1.2 Connexion

Un client se connecte au serveur grâce à une adresse IP et un port. On peut donc avoir plusieurs clients, mais un seul serveur pour une simulation donnée.

```
import carla
client = carla.Client(”localhost”, port=2000)
```

1.1.3 Principe

Concrètement, le serveur *World* simule les *frames* le plus rapidement possible et les envoient aux clients. Ces derniers les reçoivent et les traitent. Les clients sont également les modules qui permettent aux utilisateurs de CARLA d'interagir avec la simulation (faire avancer un véhicule, changer la météo...).

1.2 Les Acteurs

Est considéré comme Acteur tout ce qui joue un rôle dans une simulation (véhicules, piétons, capteurs, spectateur, feux de route...).

1.2.1 Définir un acteur

Chaque acteur est précisément décrit par un *blueprint* dans la librairie éponyme.

```
world = client.get_world()
blueprint_library = world.get_blueprint_library()
vehicle_bp = blueprint_library.filter(”model3”)[0]
```

Les *blueprint* ont des configurations par défaut modifiable grâce à leurs attributs. Ces derniers sont intégralement décris dans cette section de la documentation.

```
vehicle_bp.set_attribute(”color”, ”255,0,0”)  # a red vehicule
```

1.2.2 Ajouter d'un acteur à la simulation

Pour ajouter un acteur à la simulation, on choisit un point d'apparition et on envoie le *blueprint* configuré au serveur.

```
spawn_point = random.choice(world.get_map().get_spawn_points())
vehicle = world.spawn_actor(vehicle_bp, spawn_point)
```

1.2.3 Précautions

On définit toujours une liste des acteurs que l'on tient à jour. Car à la fin de la simulation, on se doit de tous les détruire.

```
actor_list = []
try:
    ...
    actor_list.append(vehicle)
    ...
finally:
    for vehicle in vehicle_list:
        try:
            vehicle.destroy()
        except RuntimeError:  # vehicle already destroyed
            pass
```

1.3 Les cartes

Les cartes représentent la ville. Il en existe 8 et portent le nom de *Town0x* (avec x un chiffre). Chacune a ses particularités en terme d'environnements urbains. En changeant de carte, on modifie le monde dans lequel la simulation s'effectue:

```
world = client.load_world("town01")
```

Par défaut, la carte utilisée porte le nom de *town10HD* (seule carte dérogeant à la règle de nomenclature). Les descriptions des cartes sont disponibles ici.

2 La luminosité

2.1 La météo

Pour définir une météo, on utilise

```
weather = carla.WeatherParameters(
    cloudiness=80.0,
    precipitation=30.0,
    sun_altitude_angle=70.0)

world.set_weather(weather)
```

2.2 Le jour et la nuit

Le simulateur considère que l'on est en mode nuit lorsque l'angle du soleil est négatif. On paramètre donc la nuit et le jour depuis la météo. Les lumières de villes s'adaptent en conséquence, mais ce n'est pas le cas pour les phares des voitures. Il faut spécifier leur activation.

```
#  Activation
carla.VehicleLightState.All

#  Desactivation
carla.VehicleLightState.NONE
```

3 Les caméras

3.1 Initialiser une caméra

Les caméras sont avant tout des acteurs. Il faut donc les charger avec leur blueprint et les paramétrer comme on le souhaite. Dans le cadre de notre projet, on utilisera ces deux types de caméras:

Figure 1: Caméra RGB

Figure 2: Caméra de segmentation sémantique

Une initialisation simple et classique pour la caméra RGB est:

```
# Find the blueprint of the sensor.
camera_bp = world.get_blueprint_library().find('sensor.camera.rgb')
# Modify the attributes of the blueprint to set image resolution and field of view.
camera_bp.set_attribute('image_size_x', '1920')
camera_bp.set_attribute('image_size_y', '1080')
camera_bp.set_attribute('fov', '110')
# Set the time in seconds between sensor captures
camera_bp.set_attribute('sensor_tick', '1.0')
```

D'autres paramètres sont à tester, pour augmenter la netteté des images:

- bloom_intensity: Intensity for the bloom post-process effect, 0.0 for disabling it.
- blur_amount: Strength/intensity of motion blur
- blur_radius: Radius in pixels at 1080p resolution to emulate atmospheric scattering according to distance from camera.
- motion_blur_intensity: Strength of motion blur [0,1].
- motion_blur_max_distortion: Max distortion caused by motion blur. Percentage of screen width.
- motion_blur_min_object_screen_size: Percentage of screen width objects must have for motion blur, lower value means less draw calls.

3.2 Attacher une caméra à une voiture

Les caméras doivent être attachées à un autre acteur. On précise leur position de manière relative à cet acteur support (qui est généralement le véhicule).

```
spawn_point = carla.Transform(carla.Location(x=2.5, y=0.1, z=80),
                              carla.Rotation(roll=0, pitch=-30, yaw=0))  # aerian view
camera = world.spawn_actor(camera_bp, spawn_point, attach_to=vehicle)
```

3.2.1 Les 6 degrés de libertés pour placer la caméra

Le jeu de paramètre (x, y, z) positionne la caméra relativement à son support (référencé par l'attribut *attach_to*). Les axes (Ox, Oy, Oz) sont respectivement associés aux angles (roll, pitch, yaw) atour desquels la caméra est inclinée. Ces angles sont exprimés en degrés. Une représentation est donnée en figure 3.

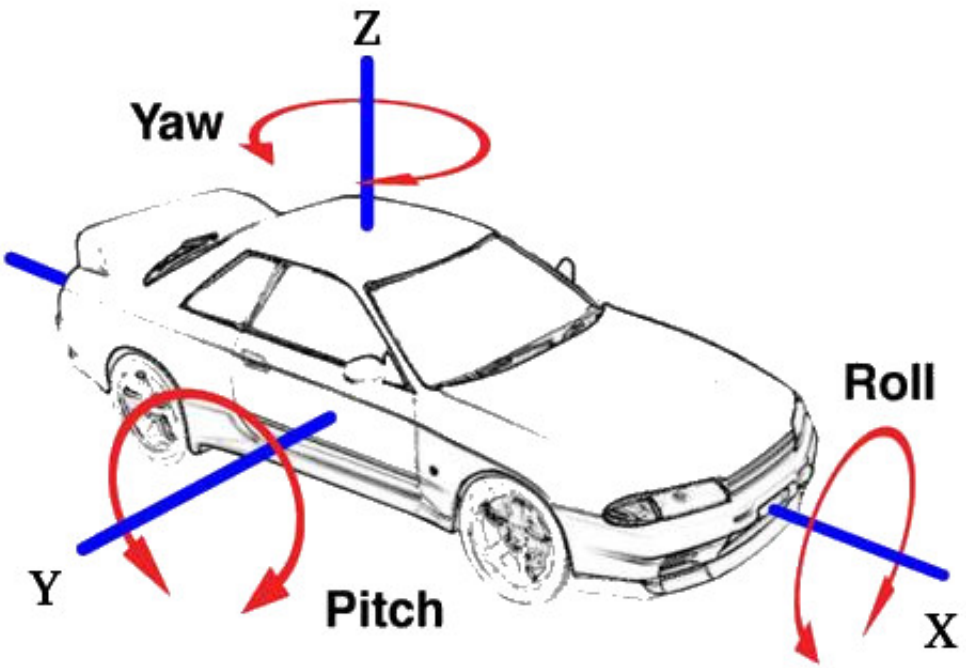

Figure 3: Les degrés de libertés autour de la voiture

3.3 Enregistrer les images prises

La méthode listen() est appelée chaque fois que le capteur génère des données, pour que l'on puisse les prendre en charge. Elle prend en paramètre une fonction lambda. Pour enregistrer les images sur le disque, on utilisera le code suivant pour le RGB:

```
camera.listen(lambda image: image.save_to_disk(f"_out/rgb_{image.frame}.png"))
```

Et pour la segmentation sémantique:

```
camera.listen(lambda image: image.save_to_disk(f"_out/seg_{image.frame}.png", carla.ColorConverter.CityScapesPalette)
```

4 Synchroniser les caméras

Pour synchroniser les différentes caméras, nous allons mettre en mode pause le serveur *World*. Pour cela, nous allons prendre le contrôle de la boucle principal et imposer le rythme du client au serveur. Cela se fait en passant en mode *synchronisé*. L'objectif est d'enregistrer au même instant les données des caméras. On travaille donc du côté client.

4.1 Récupérer et modifier les paramètres du serveur

Nous devons récupérer les paramètres originaux du serveur pour pouvoir les rétablir et finir proprement la simulation.

```
original_settings = world.get_settings()
settings = world.get_settings()
```

Ensuite, on configure le mode synchronisé

```
settings.fixed_delta_seconds = 0.1
settings.synchronous_mode = True
world.apply_settings(settings)
```

4.2 Mise en place

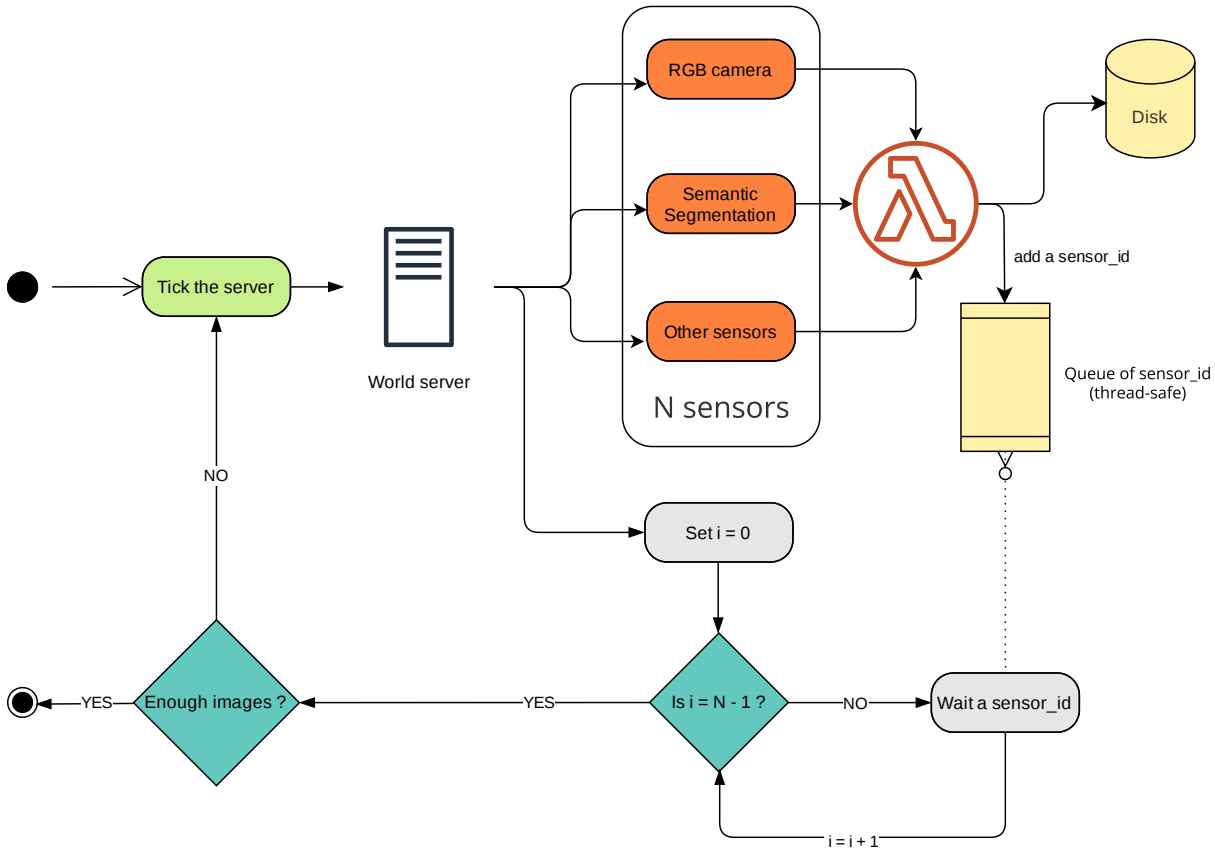

Figure 4: Principe de la synchronisation

La synchronisation s'effectue grâce à deux méthodes:

1. la méthode *tick()* autorisant le serveur à calculer une *frame* supplémentaire.
2. la méthode *get()* permettant de récupérer et de supprimer la première donnée disponible de la queue.

4.2.1 La classe Queue de Python

La classe Queue de Python est une liste d'attente synchronisée et *thread-safe*. On lui ajoute un élément avec la méthode *put()* et on lui retire son premier élément avec la méthode *get()*.

4.2.2 Fonction Lambda

La fonction fonction lambda appelée par les caméras lorsqu'elles reçoivent leurs données effectue deux tâches: l'enregistrement sur le disque de l'image et l'ajout d'un élément dans la *sensor_queue*:

```
def sensor_callback(image, sensor_queue, image_tag):
    image.save_to_disk(f"_out/{image.frame}_{image_tag}.png")
    sensor_queue.put((image.frame, image))
```

4.2.3 Boucle principale

L'ordre dans lequel arrive les données n'ont pas d'importance car chaque capteur est assigné à un thread et n'exécute sa fonction lambda que lorsque les données le concernant sont disponibles. La boucle principale doit donc juste attendre que l'ensemble des capteurs aient réalisé leurs tâches, avant de débloquer le serveur.

```
for _ in range(IM_NUMBER):
    world.tick()
    try:
        for _ in range(len(sensor_list)):
            s_frame = sensor_queue.get(block=True, timeout=1.0)
            print("Frame: %d   Sensor: %s" % (s_frame[0], s_frame[1]))

    except Empty:
        print("Some of the sensor information is missed")
```

En paramétrant la méthode *get* de manière bloquante avec un *timeout* de 1.0, on s'assure de ne pas manquer une donnée sous réserve qu'elle soit générée en moins de 1 seconde depuis le dernier *tick()*.

5 Simulation déterministe

L'objectif du projet est d'enregistrer des paires d'image jour / nuit de mêmes scènes. Pour y parvenir, on produit des simulations identiques en ne modifiant que les conditions d'éclairage.

5.1 Traffic Manager

Pour rendre le comportement des voitures non aléatoire, on utilise le *Traffic Manager* de CARLA. Toutes les voitures sont alors contrôlées par la même entité, et le comportement de cette dernière est rendue déterministe en fixant la *Seed* lui permettant de générer les séquences aléatoires dont elle a besoin. Enfin, pour que cela fonctionne, il est nécessaire de contrôler le débit du serveur. Cela a déjà été mis en place dans la section 4.1 traitant de la synchronisation des capteurs.

```
traffic_manager = client.get_trafficmanager(port=8000)
traffic_manager.set_synchronous_mode(True)
traffic_manager.set_random_device_seed(seed_value=1)

vehicle.set_autopilot(True, port=8000))
```

Par défaut, le *Traffic Manager* utilise le port 8000.

5.2 Les piétons

Le comportement des piétons n'est pas parfaitement déterministe car ces derniers sont gérés par la navigation système de Unreal, sur laquel nous n'avons pas entièrement la main[1]. Le risque majeur est alors qu'un piéton ait deux comportements distincts et influence différemment la prise de décision du véhicule porteur des caméras.

6 Certifier la correspondance jour/nuit

Pour assurer la correspondance des paires d'image, nous procédons de la sorte:

- synchronisation de la caméra rgb avec une caméra de segmentation

[1] Réponse d'un développeur CARLA sur GitHub

- comparaison des images de segmentation sémantique obtenues lors des deux simulations.

La nature d'un objet étant invariant selon la météo, on pourra détecter des paires d'image ne reproduisant pas les mêmes scènes.

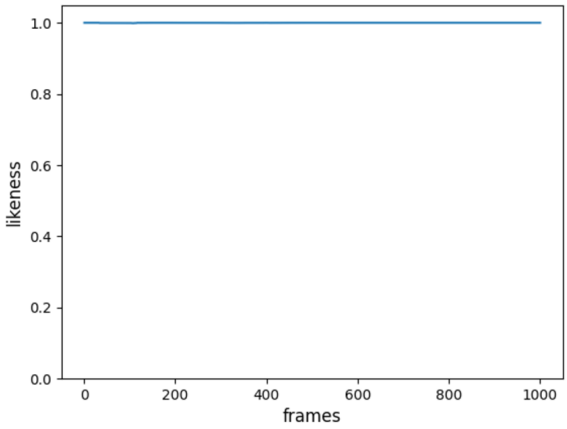

Figure 5: Deux simulations bien synchronisées

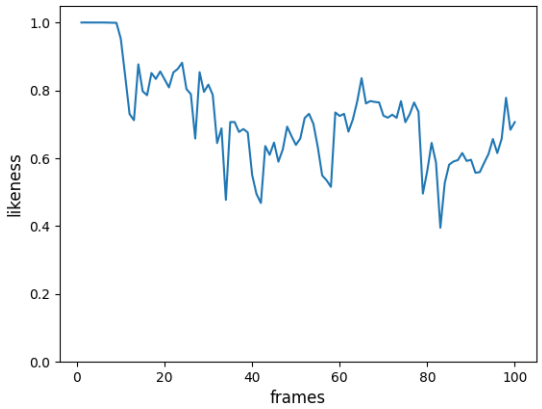

Figure 6: Deux simulations qui divergent

Références

[1] La documentation en ligne du simulateur CARLA v.0.9.13

[2] La documentation en ligne de Python3