Mini Doc du simulateur CARLA 0.9.X

Groupe CARLA

Mars 2022 (v. 1.3)

Sommaire

1 Les bases

1.1 Une architecture client-serveur

1.1.1 Interfaces

La simulation comporte deux interfaces:

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

Le serveur se lance avec le script CarlaUE4.sh et un client avec un script python. On travaillera donc 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, mais on peut les modifier. Pour cela, il faut prendre connaissance de leurs attributs sur cette section de la documentation.

```
vehicle_bp.set_attribute(”color”, ”255,0,0”)
```

1.2.2 Ajouter d'un acteur à la simulation

Pour ajouter un acteur à la simulation, on choisit une localisation 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 actor in actor_list:
        actor.destroy()
```

1.3 Les cartes

Les cartes représentent la ville. Il en existe 10 et portent le nom de *Townxx* (avec xx un numéro). Chacune a ses particularités en terme d'environnements urbains. Pour changer de carte, on peut utiliser le script *config.py* qui se trouve dans l'API python: *python3 config.py –map Town01*

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. Il existe un effet de bord à passer en nuit: les réverbères urbains s'allument automatiquement. Ce n'est en revanche pas automatique pour les lumières des voitures. Des codes d'exemple sont fournies sur cette section de la documentation pour comprendre comment paramétrer tout cela.

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 caméras:

- Caméra RGB
- Caméra de segmentation sémantique

L'initialisation de la caméra RGB est la suivante:

```
# 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')
```

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, z=0.7))
camera = world.spawn_actor(camera_bp, spawn_point, attach_to=vehicle)
```

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.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

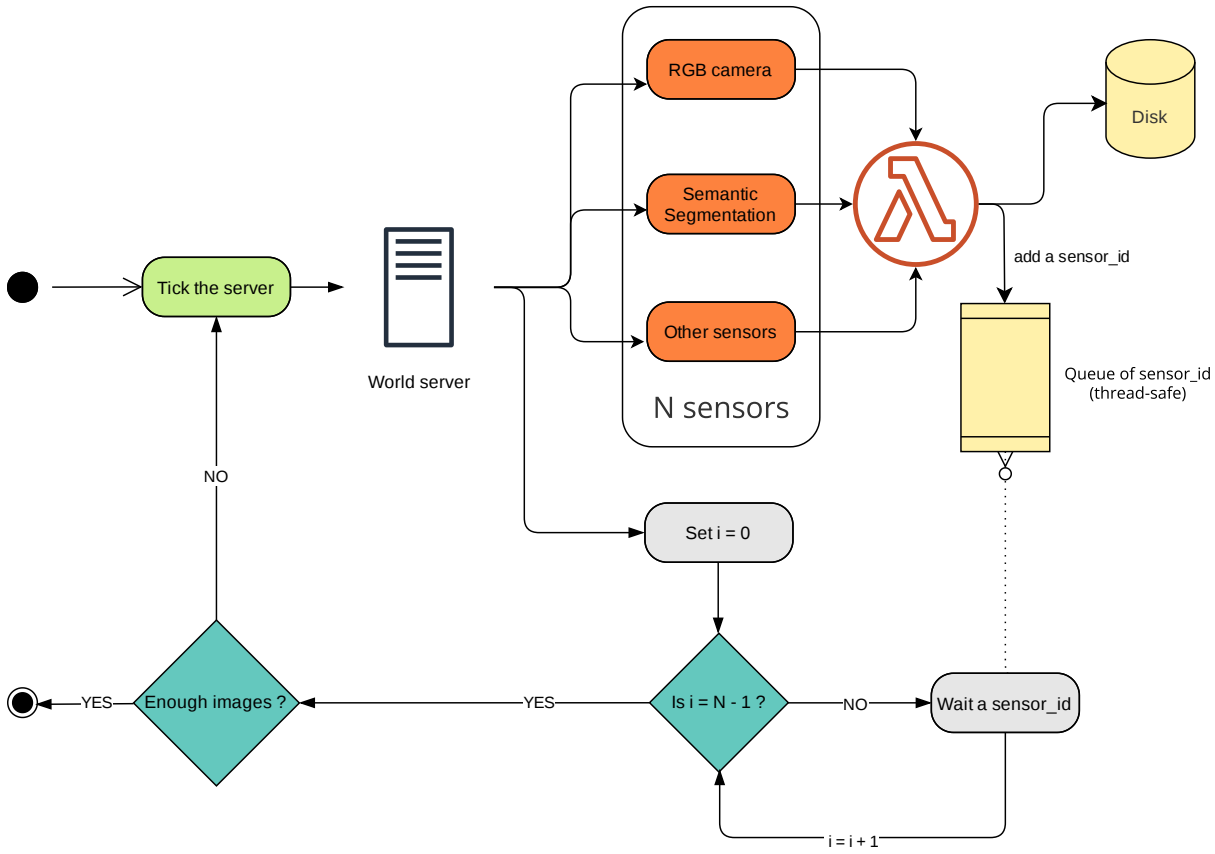

Figure 1: Principe de la synchronisation

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 Rejouer une simulation enregistrée

Cette section fait référence à cette partie de la documentation.

5.1 Introduction

5.1.1 Utilisation des ressources

Enregistrer et jouer en même temps une simulation est coûteux en ressource. On peut s'affranchir de l'affichage grâce au script *no_rendering_mode.py* qui est fourni dans l'API.
La documentation estime qu'une heure de simulation avec 100 voitures et 50 lumières nécessite 200Mb.

5.1.2 Ce qui est sauvegardé

Les données suivantes sont enregistrées: les acteurs, les lumières (feux des voitures, de signalisation, de ville, de rue, de bâtiment), les véhicules et les piétons.

5.2 Enregistrer

5.2.1 Lancer l'enregistrement

Pour lancer l'enregistrement, on utilise:

```
client.start_recorder("recorder/recording01.log", additional_data=True)
```

Si aucun chemin n'est spécifié, l'enregistrement est stocké dans le dossier CarlaUE4/Saved. Le simulateur n'enregistre que les informations qu'il juge "utile" pour pouvoir rejouer la simulation. On peut forcer le simulateur à tout enregistrer en ajoutant l'option *additional_data*.

5.2.2 Arrêter l'enregistrement

Pour arrêter l'enregistrement en cours, on utilise:

```
client.stop_recorder()
```

5.3 Altérer et rejouer la simulation

Après avoir altérer la météo (section 2.1), on peut rejouer la simulation:

```
client.replay_file("recorder/recording01.log", start=0, duration=0, camera=vehicle_id)
```

En mettant *start* et *duration* à 0, on rejoue intégralement la simulation enregistrée. L'option caméra demande l'id du véhicule sur lequel repose les caméras. L'id d'un acteur étant obtenu en retour de la fonction *world.spawn_actor* (cf section 1.2.2).

6 Modifier la météo durant la simulation

On ne peut pas mettre en pause la simulation, enregistrer des images, changer la météo et enregistrer de nouvelles photos. Car pour que la météo change, il faut laisser tourner le serveur. Au lieu de mettre en pause le serveur, on peut faire en sorte que les acteurs ne bougent pas. Pendant cet intervalle, la météo peut changer car le serveur n'est pas arrêté.

6.1 traffic manager

La vitesse des véhicules est contrôlée par le *traffic manager*. On l'initialise à l'aide d'un port:

```
tm_port = 8000
tm = client.get_trafficmanager(tm_port)
```

6.2 Arrêt et reprise d'un véhicule

On peut régler la vitesse d'un véhicule en fonction de la limitation en vigueur:

```
# vehicle drives 100% slower than the current speed limit
tm.vehicle_percentage_speed_difference(vehicle, 100)

# vehicle drives at the current speed limit
tm.vehicle_percentage_speed_difference(vehicle, 0)
```

7 Liens utiles

Lire cette partie de la doc et ce thread.

Références

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

[2] La documentation en ligne de Python3