

Mini Doc du simulateur CARLA 0.9.X

Groupe CARLA

Février - 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	6
7	Codes et bases de données	6

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 le client avec un script python se basant sur l'API fourni. Dans ce projet, nous travaillons uniquement du côté client

1.1.2 Connexion

Un client se connecte au serveur grâce à une adresse IP et un port. Pour une simulation donnée, il existe un unique serveur et de multiples clients.

```
1 import carla
2 client = carla.Client(host="localhost", port=2000)
```

1.1.3 Principe

Concrètement, le serveur *World* simule les *frames* le plus rapidement possible et les envoie aux clients. Ces derniers les reçoivent et les traitent via différents capteurs. Les clients interagissent également avec la simulation en faisant avancer un véhicule, en changeant la météo...

1.2 Les Acteurs

Est considéré comme acteur toute entité jouant 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.

```
1 world = client.get_world()
2 blueprint_library = world.get_blueprint_library()
3 vehicle_bp = blueprint_library.filter("model3")[0]
```

Les *blueprint* ont des configurations par défaut et modifiables grâce à leurs attributs. Ces derniers sont listés dans [cette section de la documentation](#).

```
1 vehicle_bp.set_attribute("color", "255,0,0") # a red vehicle
```

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 charge le *blueprint* configuré au serveur.

```
1 spawn_point = random.choice(world.get_map().get_spawn_points())
2 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. Sinon certains *spawn_point* peuvent rester occupés. Ce qui peut empêcher les prochaines simulations sur le même serveur de se dérouler convenablement.

```
1 actor_list = []
2 try:
3     ...
4     actor_list.append(vehicle)
5     ...
6 finally:
7     for actor in actor_list:
8         actor.destroy()
```

1.3 Les cartes

Les cartes représentent la ville. Il en existe 8 et portent le nom de *Town0x* (avec x un chiffre). Chacune présente ses particularités en terme d'environnement urbain. En changeant de carte, on modifie le monde dans lequel la simulation s'effectue, mais pas le serveur (les connexions serveur-clients ne sont pas interrompues).

```
1 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 différentes cartes sont [disponibles ici](#).

2 La luminosité

2.1 La météo

Pour définir une météo, on utilise

```
1 weather = carla.WeatherParameters(  
2     cloudiness=80.0,  
3     precipitation=30.0,  
4     sun_altitude_angle=70.0)  
5  
6 world.set_weather(weather)
```

2.2 Le jour et la nuit

Le rendu nuit correspond à un angle de soleil négatif. Les lumières de villes s'adaptent en conséquence, mais ce n'est pas le cas pour les phares des véhicules. Il faut spécifier leur activation.

```
1 # Activation  
2 carla.VehicleLightState.All  
3  
4 # Desactivation  
5 carla.VehicleLightState.NONE
```

3 Les caméras

3.1 Initialiser une caméra

Les caméras sont avant tout des acteurs. Il faut donc les charger avec leurs *blueprints* et les paramétrer. Dans le cadre de notre projet, nous utilisons 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:

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

D'autres paramètres existent 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 un véhicule).

```
1 spawn_point = carla.Transform(carla.Location(x=2.5, y=0.1, z=80),  
2                               carla.Rotation(roll=0, pitch=-30, yaw=0)) # aerie view  
3 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 paramètres (x, y, z) positionne la caméra relativement à son support (référéncé par l'attribut *attach_to*). Les axes (Ox, Oy, Oz) sont respectivement associés aux angles (roll, pitch, yaw) autour desquels la caméra est inclinée. Ces angles sont exprimés en degrés. Une représentation est donnée en figure 3.

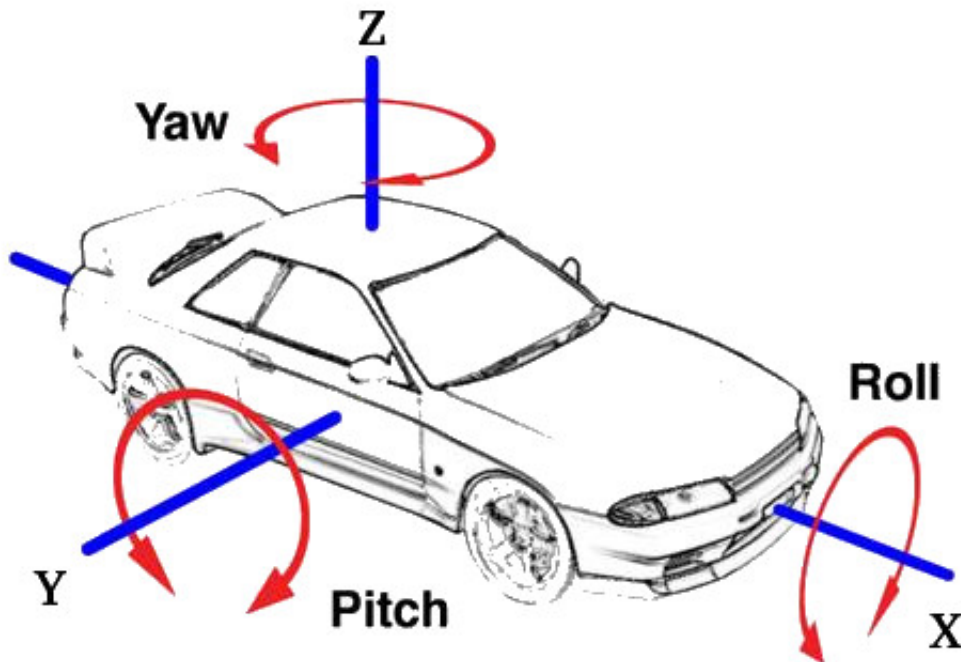


Figure 3: Les degrés de libertés autour de la voiture

3.3 Enregistrer les images prises

La méthode `listen()` est appelée à chaque fois que le capteur reçoit des données du serveur. Elle prend en paramètre une fonction `lambda`. Pour enregistrer les images RGB sur le disque, on utilisera le code suivant:

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

Et pour la segmentation sémantique:

```
1 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 simulons un mode pause sur le serveur *World*. Cela est possible en passant en mode *synchronisé*. L'objectif final étant d'arrêter le serveur, d'enregistrer les dernières données parvenues aux caméras, puis de relancer le serveur.

4.1 Récupérer et modifier les paramètres du serveur

Nous sauvegardons les paramètres originaux du serveur pour pouvoir les rétablir et finir proprement la simulation plus tard:

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

Ensuite, nous configurons le mode synchronisé:

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

La valeur *fixed_delta_seconds* est choisie à force d'expérimentation, mais est liée à deux autres paramètres: le *sensor_tick* des capteurs et le nombre *N* de scènes générées entre deux enregistrements. De plus cette valeur ne doit pas excéder 0.166667 pour garantir la répétabilité de la simulation.

$$\text{sensor_tick} = N * \text{fixed_delta_seconds}$$

Pour ce projet, nous enregistrons une image par seconde (temps simulateur), nous prenons donc $N = 10$.

4.2 Mise en place

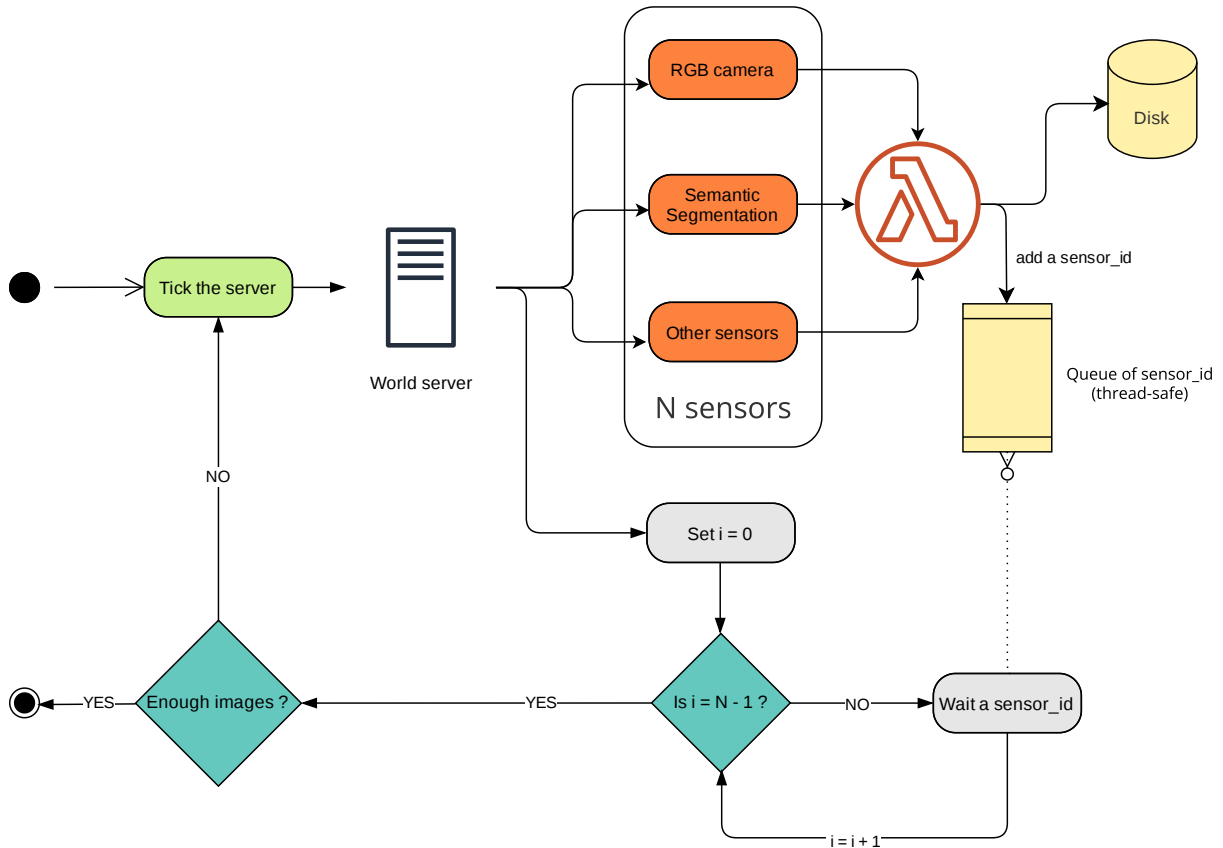


Figure 4: Principe de la synchronisation

La synchronisation s'effectue grâce à deux méthodes:

1. la méthode *tick()* autorisant le serveur à calculer la prochaine scène.
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()`. Grâce à cet objet nous simulons le comportement d'un sémaphore synchronisant les *threads* des capteurs.

4.2.2 Fonction Lambda

Une *fonction lambda* est appelée par les caméras lorsqu'elles reçoivent leurs données. Dans notre cas, elle effectue deux tâches: l'enregistrement sur le disque de l'image et l'ajout d'un élément dans la *sensor_queue*:

```
1 def sensor_callback(image, sensor_queue, image_tag):
2     image.save_to_disk(f'_{out}/{image.frame}_{image.tag}.png')
3     sensor_queue.put((image.frame, image))
```

4.2.3 Boucle principale

La boucle principale attend que l'ensemble des capteurs aient réalisé leurs tâches, avant de débloquent le serveur.

```
1 world.tick()
2 for _ in range(IM.NUMBER):
3     try:
4         for _ in range(len(sensor_list)):
5             sensor_queue.get(block=True, timeout=2.0)
6
7     except Empty:
8         print("Some of the sensor information is missed")
9
10    for _ in range(round(sensor_tick/fixed_delta_seconds)):
11        world.tick()
```

En paramétrant la méthode `get` de manière bloquante avec un *timeout* de 2.0, nous nous assurons de ne pas manquer une donnée sous réserve qu'elle soit générée et enregistrée en moins de 2 secondes.

5 Simulation déterministe

L'objectif du projet est d'enregistrer des paires d'image jour / nuit de mêmes scènes. Pour y parvenir, nous réalisons 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. Les choix pris sont rendus déterministes en fixant la même *seed* pour les deux simulations. Ce paramètre permet de générer les séquences pseudos-aléatoires nécessaires à la prise de certaines décisions. Enfin, si le serveur est en mode synchronisé, il faut veiller à ce qu'il en soit de même pour le *Traffic Manager*.

```
1 traffic_manager = client.get_trafficmanager(port=8000)
2 traffic_manager.set_synchronous_mode(True)
3 traffic_manager.set_random_device_seed(seed_value=1)
4
5 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 lequel nous n'avons pas entièrement la main¹. Le risque majeur est alors qu'un piéton ait deux comportements distincts et influence différemment la prise de décision du véhicule porteur des caméras.

¹Réponse d'un développeur CARLA sur GitHub

6 Certifier la correspondance jour/nuit

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

- synchronisation d'une caméra rgb avec une caméra de segmentation sémantique,
- 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.

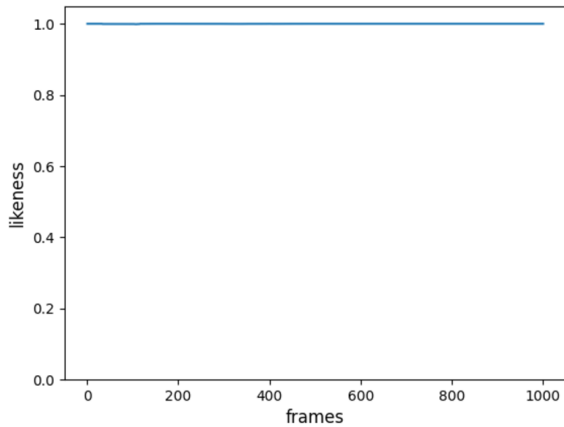


Figure 5: Deux simulations bien synchronisées

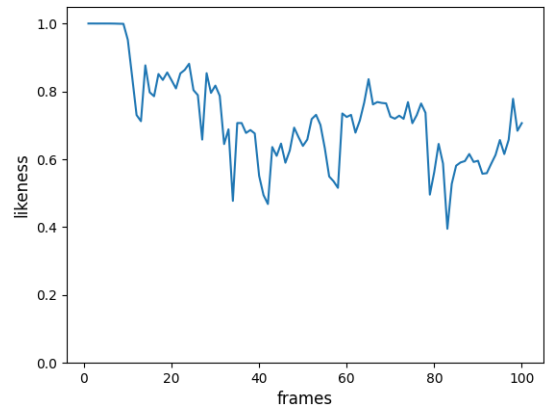


Figure 6: Deux simulations qui divergent

CARLA possède une vingtaine de classe, et certaines sont surreprésentées selon l'orientation des caméras (la route ou le ciel par exemples). Ainsi, un score de ressemblance de 90% n'est pas pas un "bon" score: il faut viser plus de 99%.

7 Codes et bases de données

Nos codes sont accessibles depuis ce dépôt [GitHub](#), et quelques paires d'images sont disponibles sur ce [Drive](#).

Références

- [1] [La documentation en ligne du simulateur CARLA v.0.9.13](#)
- [2] [La documentation en ligne de Python3](#)