



POLITECNICO DI BARI

DEI

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

AI for automation

Prof. Ing. Maria Pia Fanti

Prof. Ing. Gaetano Volpe

Prof. Ing. Agostino Marcello Mangini

Ing. Francesco Paparella

Progetto:

Digital twin di una simulazione SUMO con CARLA

Studenti:

Marcello Bari

Alessandro Mitola

ANNO ACCADEMICO 2023-2024

Abstract

In questo lavoro presentiamo lo sviluppo di un Digital Twin per le simulazioni del traffico SUMO (Simulation of Urban MObility) all'interno del simulatore di guida autonoma CARLA, con annessa interfaccia per la lettura dei dati ottenuti tramite TraCi. L'integrazione di SUMO con CARLA consente la rappresentazione accurata di scenari di traffico urbano complessi, fornendo una sincronizzazione in tempo reale tra ambienti virtuali e dati di simulazione. Il framework del Digital Twin permette di simulare condizioni di traffico dinamiche, comportamenti dei veicoli e interazioni con i pedoni, consentendo allo stesso tempo il monitoraggio, l'ottimizzazione e il testing degli algoritmi per veicoli autonomi in un ambiente controllato ma altamente realistico. Questa configurazione colma il divario tra la simulazione dei flussi di traffico e le dinamiche avanzate dei veicoli, potenziando le capacità di CARLA per scenari ad alta intensità di traffico e migliorando il processo di test e validazione dei sistemi di guida autonoma.

Contents

1	Digital Twins	3
2	CARLA: Setup di una build per Windows 11	4
2.1	Third-party software	4
2.2	Unreal Engine	6
2.3	Carla client e server	7
3	CARLA: Da 2D a 3D	9
3.1	Digital Twin tool di CARLA	9
3.2	Blender	12
4	CARLA-SUMO co-simulation	15
5	Creazione dell'interfaccia/Dashboard	19
6	Script per la telecamera	23
6.1	Inseguimento di un veicolo tramite ID	23
6.2	Inseguimento di veicoli casuali	24
6.3	Inseguimento con widget tk	25
7	Descrizione del codice	28
7.1	run_syncro.py	28
7.1.1	Inizializzazione della classe SimulationSynchronization (Righe 70-98)	28
7.1.2	Sincronizzazione delle informazioni dei veicoli (Righe 275-309)	28
7.1.3	Ciclo di sincronizzazione principale (Righe 389-409)	29
7.2	interface.py	29
7.2.1	Funzione per caricare i dati dal file JSON (Righe 11-20)	29
7.2.2	Aggiornamento delle metriche e visualizzazione dei dati (Righe 128-196)	29
7.3	camera.py	30
7.3.1	Connessione al server CARLA e impostazioni iniziali (Righe 6-13)	30
7.3.2	Ciclo principale per la selezione e il controllo della telecamera (Righe 16-27)	30
7.3.3	Posizionamento e orientamento della telecamera (Righe 30-41)	31
7.4	camera_with_tk.py	31
7.4.1	Funzione per leggere informazioni da un file JSON (Righe 14-27)	31
7.4.2	Funzione per aggiornare l'interfaccia utente (Righe 29-45)	32
7.4.3	Funzione per aggiornare la telecamera e l'interfaccia utente (Righe 47-67)	32
7.4.4	Selezione del veicolo da seguire tramite ID (Righe 69-83)	33
7.4.5	Selezione casuale di un veicolo da seguire (Righe 85-94)	33
7.5	modifyXML.py	33
7.5.1	Funzione per aggiungere un tipo di veicolo casuale nel file XML (Righe 23-33)	33
8	Conclusioni	34

1 Digital Twins

I **Digital Twin**, o "Gemelli Digitali", negli ultimi anni hanno trovato largo sviluppo all'interno di vari settori, come quello automotive, della sanità, e industriale. Inoltre, ricoprono fondamentale importanza nell'ambito delle Smart City e quindi, della **mobilità sostenibile**. La loro concezione e sviluppo è piuttosto recente e spesso vengono identificati tramite la seguente definizione:

Un Digital Twin è la rappresentazione virtuale di un oggetto fisico, di un'attività o di un processo, capace di rifletterne tutte le proprietà e i comportamenti utili all'interno di un determinato contesto.

Quindi, risulta possibile intuire come il funzionamento di un Digital Twin si basi sostanzialmente su un ciclo continuo di acquisizione e analisi dei dati. Informazioni sull'entità fisica di interesse vengono raccolte in tempo reale, e poi trasmesse a un modello virtuale, che a sua volta può trasmettere dati riguardo al suo stato interno verso l'esterno.

Il modello deve essere in grado di simulare il comportamento del sistema, aggiornandosi costantemente per riflettere le condizioni operative attuali.

Grazie a tecnologie come l'intelligenza artificiale e il **reinforcement learning**, il gemello digitale può anche eseguire analisi predittive, fornendo previsioni sui possibili scenari futuri e suggerendo interventi proattivi.

Nel contesto della gestione intelligente del traffico, un Digital Twin può essere creato attraverso l'utilizzo contemporaneo della piattaforma **CARLA** e del simulatore di traffico **SUMO**. Carla è un simulatore open-source che permette di creare ambienti urbani virtuali e di testare la dinamica di veicoli autonomi, mentre SUMO (Simulation of Urban MObility) è uno strumento di simulazione del traffico veicolare open-source, microscopico e multimodale che modella il comportamento di un'intera rete di trasporti.

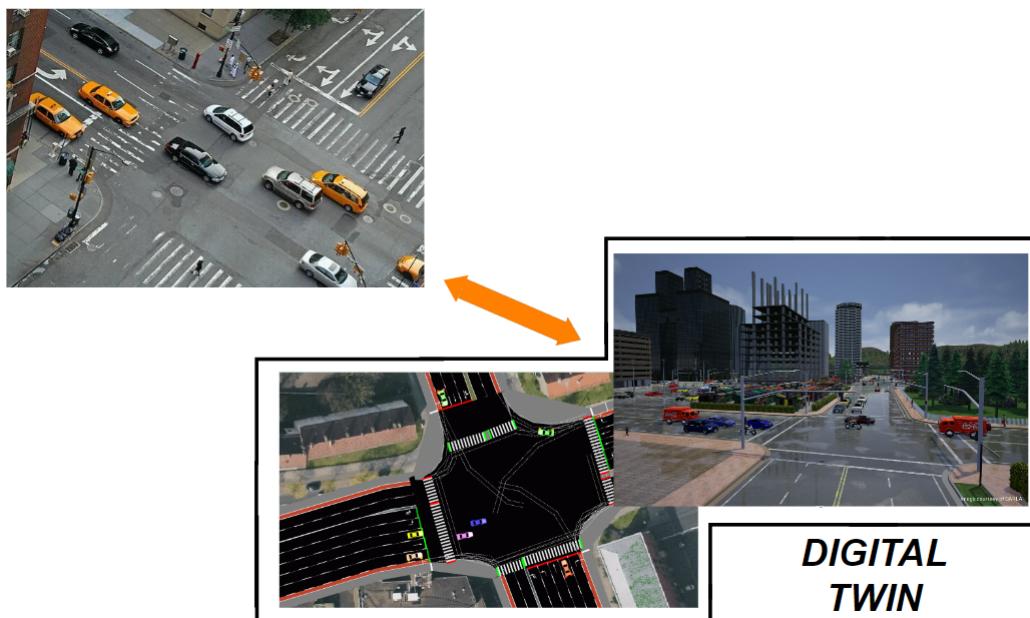


Figure 1: Interazione fra SUMO e CARLA

L'interazione tra i due sistemi è consentita dal continuo trasferimento di dati mediante opportuni messaggi di rete, che seguono il protocollo **TCP/IP** per garantire una comunicazione affidabile e sincronizzata. Mentre Carla simula un singolo veicolo o un gruppo di veicoli autonomi, SUMO

aggiorna costantemente il traffico circostante, influenzando la simulazione di Carla in tempo reale. Allo stesso tempo, le azioni dei veicoli simulati in Carla influenzano il comportamento del traffico in SUMO. Questo ciclo di feedback crea un gemello digitale che replica fedelmente il comportamento sia dei singoli veicoli autonomi che dell'intero sistema di trasporto urbano.

2 CARLA: Setup di una build per Windows 11

A differenza di SUMO, l'installazione di Carla può risultare ostica, trattandosi di un software particolarmente oneroso sotto il punto di vista delle specifiche hardware. Esistono, infatti, numerose versioni di Carla, ciascuna con funzioni e occupazione di memoria differenti, in modo da consentire agli sviluppatori di alleggerire il carico computazionale dei vari task della simulazione.

Per il nostro progetto, tuttavia, risulta pressoché obbligatorio utilizzare la "Source build" di Carla (la più completa e pesante), in quanto si tratta dell'unica opzione che permette di introdurre nuove mappe, e quindi nuovi scenari di traffico, all'interno del simulatore.

La guida che proponiamo è da intendersi come un aggiornamento e miglioramento di quella già presente sulla documentazione ufficiale di Carla, consultabile sul sito: https://carla.readthedocs.io/en/latest/build_windows/.

È importante specificare che affinché il processo di installazione vada a buon fine, e per permettere al simulatore di operare con buone prestazioni, è necessario un pc con almeno 200 Gb di spazio in memoria libero (il supporto di memorizzazione deve essere necessariamente una SSD) e 6 Gb di GPU. Sono ammissibili anche GPU da 4 Gb per progetti che non richiedono né l'uso massivo di machine learning né scenari tridimensionali complicati. Valori di GPU più bassi renderebbero il software sostanzialmente inutilizzabile.

Il processo di installazione di Carla si compone di 3 fasi, ciascuna delle quali verrà affrontata nel seguito.

2.1 Third-party software

I software di terze parti che saranno necessari per gestire le cartelle e i file scaricati dalla repository GitHub contenente il codice sorgente di Carla sono:

- **GPU driver**

Assicurarsi di aggiornare i driver della propria scheda grafica alla versione più recente. Per vedere che tipo di scheda è installata sul proprio pc:

Start > Impostazioni > Sistema > Gestione dispositivi > Schede video

In base alla casa produttrice, visitare il sito di [AMD](#) o [NVIDIA](#), e cercare i driver appropriati per la propria scheda grafica.

- **DirectX** è una collezione di API per lo sviluppo di videogiochi su Windows. Fa parte del sistema operativo, quindi assicurarsi che la versione utilizzata sia la 12, digitando nella barra di ricerca di Windows *dxdiag.exe* e eseguendo l'applicativo. Nel caso la versione non fosse corretta, aggiornare Windows dalla sezione Windows Update nelle impostazioni. Si noti che le versioni "packaged" di Carla richiedono DirectX 11, quindi, in quel caso è necessario il flag *- -dx11* ogni volta che si avvia il server.

- **CMake** è un progetto open-source che permette la compilazione, testing, packaging, e distribuzione di software cross-platform. Scaricare la versione più recente da [qui](#).

- **Git** è un sistema di controllo versione distribuito, ovvero un software che consente di gestire il codice sorgente di un progetto durante lo sviluppo collaborativo senza bisogno del supporto di un server centrale. Lo si può scaricare da [qui](#). Per controllare che sia stato installato correttamente è sufficiente digitare il comando *git - -version* nella PowerShell di Windows.

- **Make** è uno strumento software che controlla la generazione di eseguibili e altri file di un programma. Installare la sua versione 3.81 da [qui](#). Qualsiasi altra versione farebbe fallire la compilazione di Carla, pertanto controllare che l'installazione abbia avuto successo digitando `make --version` nella PowerShell di Windows.
- **7Zip** è un programma per la gestione e creazione di file compressi. Automatizza la decompressione degli asset di Carla. Scaricare la versione 64 bit da [qui](#).
- **Python3** è il linguaggio di programmazione utilizzato da Carla. La versione di Python da installare è decisa dal pacchetto di librerie "Boost", che verrà compilato per il client di Carla. Per *Boost 1.80.0* abbiamo verificato il corretto funzionamento di [Python 3.7.9](#) e [Python 3.8.10](#), quindi installare uno dei due. Inoltre disinstallare ogni altra versione di Python presente sul pc, compresi gli ambienti creati con "Conda", e preferibilmente "Conda" stesso.

Terminata l'installazione di Python eseguire i seguenti 3 comandi da terminale:

```
pip3 install --upgrade pip
pip3 install --user setuptools
pip3 install --user wheel
```

- **Visual Studio 2022** è un ambiente di sviluppo integrato (o IDE) sviluppato da Microsoft. Se non è stato già installato, è possibile installare la sua versione gratuita, ovvero [Visual Studio Community](#). Una volta aperto il "Visual Studio Installer" premurarsi di installare le componenti aggiuntive mostrate nell'immagine sotto.

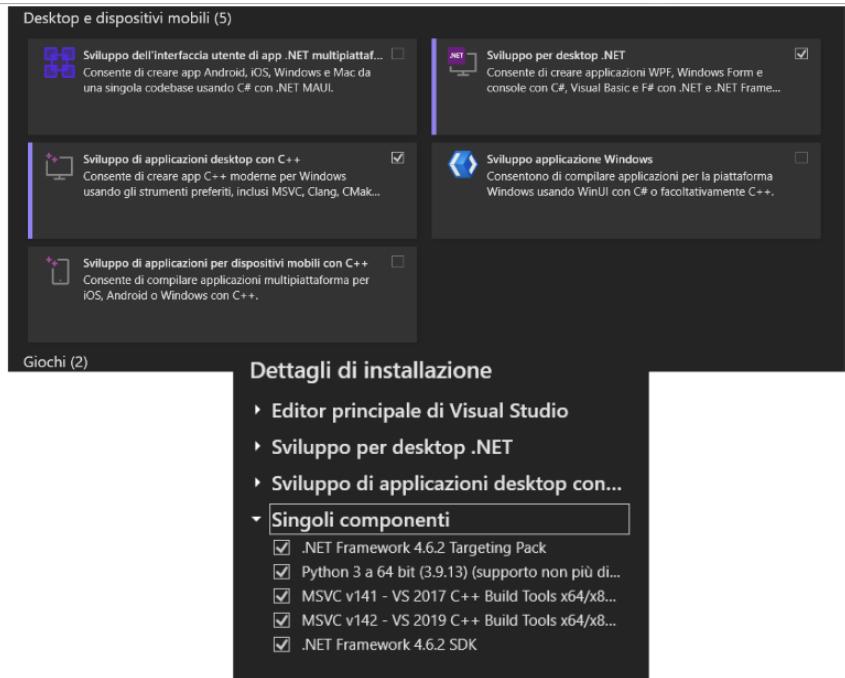


Figure 2: Add-ons di Visual Studio

Per quel che riguarda le componenti individuali, cercarle digitando il nome visibile in figura 2, nella barra di ricerca e spuntando le caselle corrispondenti.

Una volta scaricati tutti i software elencati, digitare nella barra di ricerca di Windows "Modifica le variabili di ambiente relative al sistema" e cliccare sul primo risultato. Nella schermata che appare cliccare su "Variabili d'ambiente" e nella sezione "Variabili di sistema" selezionare la voce "Path". Quindi, aggiungere i percorsi delle applicazioni installate, avendo cure di porle nella posizione più elevata possibile nell'elenco degli indirizzi, come mostrato in figura 3:

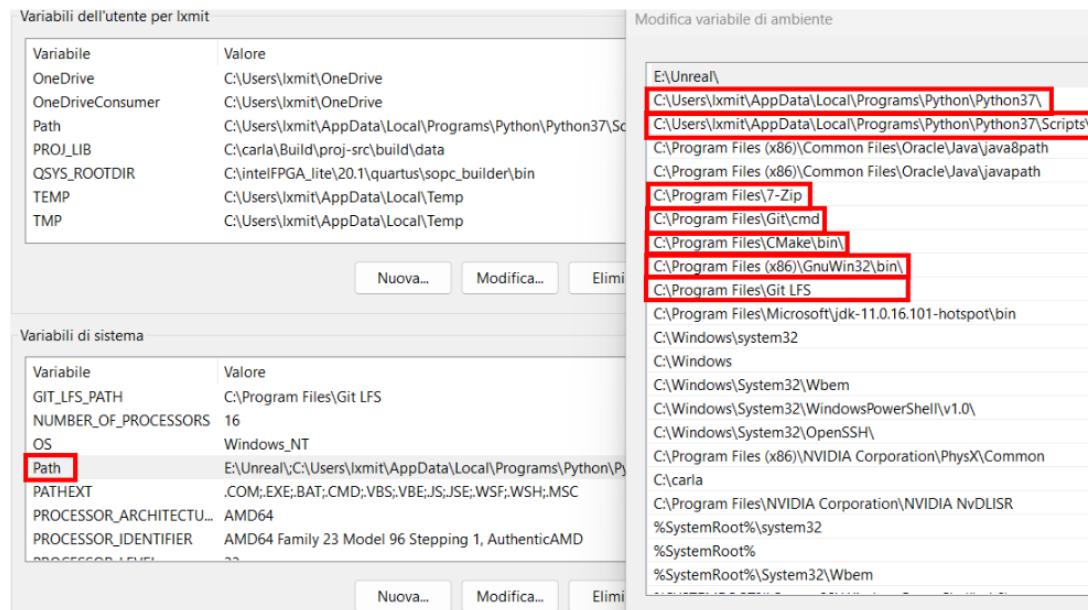


Figure 3: Variabili d'ambiente per il software di terze parti

2.2 Unreal Engine

La compilazione di una versione modificata dell'Unreal Engine, creata appositamente per Carla, dovrebbe procedere senza troppi problemi una volta effettuati i passaggi a cui si fa riferimento nella sezione precedente.

Per poter procedere è necessario collegare il proprio account GitHub a quello di Unreal Engine, in modo da poter accedere al suo codice sorgente. Per fare ciò, è sufficiente seguire la [guida ufficiale](#) di Epic Games.

Fatto questo, mediante terminale di Windows posizionarsi nella cartella che deve contenere Unreal, il cui unico requisito è quello di essere quanto più vicina a *C://*, e digitare il comando:

```
git clone --depth 1 -b carla https://github.com/CarlaUnreal/UnrealEngine.git .
```

Finita la clonazione della repository, rimanendo nella stessa cartella, che ora contiene tutti i file di Unreal, avviare i seguenti script da terminale:

```
Setup.bat
GenerateProjectFiles.bat
```

Ora cercare nella cartella il file *UE4.sln* e aprirlo. Si aprirà un progetto di Visual Studio assieme a un menu a tendina. Cliccare con il tasto destro su *UE4 (Visual Studio 2019)* e poi *Compila*.

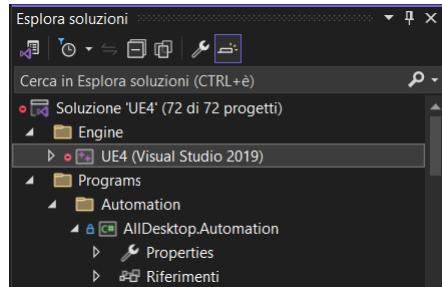


Figure 4: Progetto dell'Unreal Engine in Visual Studio

In base al proprio pc, il processo di compilazione di Unreal potrebbe richiedere da 1 a 2 ore. Alla fine di tutto, esattamente come fatto per le variabili di ambiente nella sezione precedente introdurre una nuova variabile di sistema chiamata *UE4_ROOT* che ha come valore l'indirizzo della cartella ove si è installato Unreal.

Variabili di sistema	
Variabile	Valore
SUMO_HOME	C:\Program Files (x86)\Eclipse\Sumo\
TEMP	C:\WINDOWS\TEMP
TMP	C:\WINDOWS\TEMP
UE4_ROOT	E:\Unreal\
USERNAME	SYSTEM
VBOX_MSL_INSTALL_PATH	C:\Program Files\Oracle\VirtualBox\
VXIPNPPATH	C:\Program Files (x86)\IVI Foundation\VISA\

2.3 Carla client e server

Come fatto per l'Unreal Engine, posizionarsi in una nuova cartella tramite terminale e clonare la repository GitHub di Carla con il comando seguente:

```
git clone https://github.com/carla-simulator/carla
```

A clonazione completata è finalmente possibile iniziare la compilazione del client di Carla. Per prima cosa aprire il terminale denominato *x64 Native Tools Command Prompt for VS 2022* e posizionarsi nella cartella "root" di Carla. Da ora in poi, ogni comando dovrà essere immesso da tale terminale. Avviare lo script seguente:

```
make PythonAPI GENERATOR="Visual Studio 17 2022"
```

Tale comando comincerà un processo automatico di compilazione di tutti i pacchetti che compongono il client di Carla e li memorizzerà nella cartella *Build* contenuta nella "root", in questa maniera:

Nome	Ultima modifica	Tipo
boost-1.80-install	11/08/2024 05:56	Cartella di file
boost-1.80-source	07/08/2024 09:50	Cartella di file
eigen-3.1.0	07/08/2024 08:58	Cartella di file
eigen-install	11/08/2024 05:56	Cartella di file
gtest-install	11/08/2024 05:53	Cartella di file
gtest-src	07/08/2024 08:45	Cartella di file
libcarla-visualstudio	27/08/2024 15:22	Cartella di file
libpng-1.2.37-install	11/08/2024 05:53	Cartella di file
libpng-1.2.37-source	07/08/2024 08:45	Cartella di file
osm2odr-source	10/05/2023 10:46	Cartella di file
osm2odr-visualstudio	27/08/2024 15:24	Cartella di file

Tuttavia, con elevata probabilità, il terminale segnalerà un errore, con il seguente messaggio:

[B2 ERROR] An error occurred while installing using "b2.exe"

b2.exe è il file eseguibile del pacchetto "Boost", quindi è un problema presente nella prima cartella della figura precedente.

I motivi possono essere molteplici, per questo raccomandiamo di provare con i seguenti passaggi:

- Nella cartella *Build* eliminare tutto il contenuto eventualmente presente e scaricare manualmente il file zip di *Boost* da [qui](#). La versione che consigliamo è la 1.80.0, ma visto il continuo aggiornamento della repository di Carla, la versione giusta potrebbe variare. Terminato il download, estrarre la cartella e inserirla all'interno di *Build* rinominandola; "boost-1.80.0-source". Provare a ripetere la compilazione del client, con il comando precedentemente illustrato.
 - Se l'errore si ripresenta, eliminare nuovamente tutto il contenuto presente nella cartella *Build*, eccezione fatta per la cartella "boost-1.80.0-source" precedentemente installata. Ora partendo dalla cartella *Build* seguire il percorso:

Build > boost-1.80.0-source > tools > build > src > tools

E aprire il file *msvc.jam*. In esso arrivare alla linea 1122 e controllare la presenza del seguente sezione di codice:

```
1 if [ MATCH "(14.4)" : $(version) ]
2 {
3     if $(.debug-configuration)
4     {
5         ECHO "notice: [generate-setup-cmd] $(version) is 14.4x" ;
6     }
7     parent = [ path.native [ path.join $(parent) "...\\..\\..\\..\\..\\..\\..\\..\\Auxiliary\\Build" ] ] ;
8 }
9 }
```

Nel caso non sia presente, aggiungerla e provare a ricompilare da terminale con lo stesso comando precedentemente descritto.

- Qualora l'errore si ripeta nuovamente, non rimane che provare a cambiare toolset di Visual Studio e quindi riavviare la compilazione tramite il seguente script:

```
make PythonAPI GENERATOR="Visual Studio 17 2022" -T "msvc-14_2"
```

Provare anche con $m_{SUSY} = 1/1$ e $m_{SUSY} = 1/3$

Se nulla sembra funzionare, consultare la sezione di [supporto](#) della repository GitHub.

Una volta compilato il client, non dovrebbero esservi più grossi problemi. Non rimane che scaricare gli asset di lavoro di Carla eseguendo da terminale, nella cartella "root" di Carla:

Update bat

Infine, avviare il server di Carla con il comando:

```
make launch GENERATOR="Visual Studio 17 2022"
```

Ciò rende Carla effettivamente operativo, infatti, è ora possibile accedere e interagire con ogni sua funzione e comando.

3 CARLA: Da 2D a 3D

Il primo passo per la costruzione di un digital twin, usando le funzionalità combinate di CARLA e SUMO, è la creazione di mappe virtuali 3D dell'area urbana scelta per la simulazione di traffico. Come è noto, SUMO impiega file con estensione **.xml** per rappresentare la rete stradale. Essi permettono la creazione di una struttura bidimensionale composta da un grafo con nodi (incroci) e archi (corsie), sui quali vengono definiti i veicoli e i loro percorsi.

Carla, invece necessita di due tipi di file differenti per creare le sue mappe:

- Un file **.fbx** che definisce la struttura dei corpi tridimensionali che popolano gli scenari.
- Un file **.xodr** che definisce la rete stradale, specificandone la geometria e la logica di attraversamento.

Per garantire la coerenza tra le mappe utilizzate in CARLA e SUMO e consentire l'esecuzione di simulazioni senza disallineamenti tra i due ambienti, abbiamo individuato 3 iter diversi che, partendo dallo stesso file **.osm** ottenuto con [OpenStreetMap](#), permettono la creazione di tutti i file necessari.

Dei 3 metodi sopra citati, l'unico che non verrà trattato nel dettaglio è quello che utilizza **Road-Runner**.

Sebbene offra strumenti avanzati per la generazione di reti stradali e ambienti realistici, trattandosi di un software di proprietà di Mathworks, presenta un costo elevato.

Per questo abbiamo scelto di concentrarci su soluzioni open-source, che permettono una maggiore flessibilità e accessibilità.

3.1 Digital Twin tool di CARLA

CARLA mette a disposizione, nativamente, un tool di generazione procedurale di ambienti, che utilizza dati forniti da OpenStreetMap, quali elevazione del terreno, posizione degli edifici e del pavimento stradale, per riprodurre fedelmente scenari reali.

Il tool utilizza una libreria interna a CARLA per recuperare gli asset di costruzione da usare per la creazione delle mesh. Ovviamente, è possibile cambiare la libreria per meglio adattare l'aspetto grafico della simulazione.

Per cominciare a creare lo scenario, avviare il server di Carla e posizionarsi, tramite il Content Browser, nella cartella *CarlaTools Content/OnroadMapGenerator*, e aprire il file *UW_OnRoadMainWidget* cliccando su di esso con il tasto destro e selezionando la prima opzione. Verrà aperta l'interfaccia del Widget:

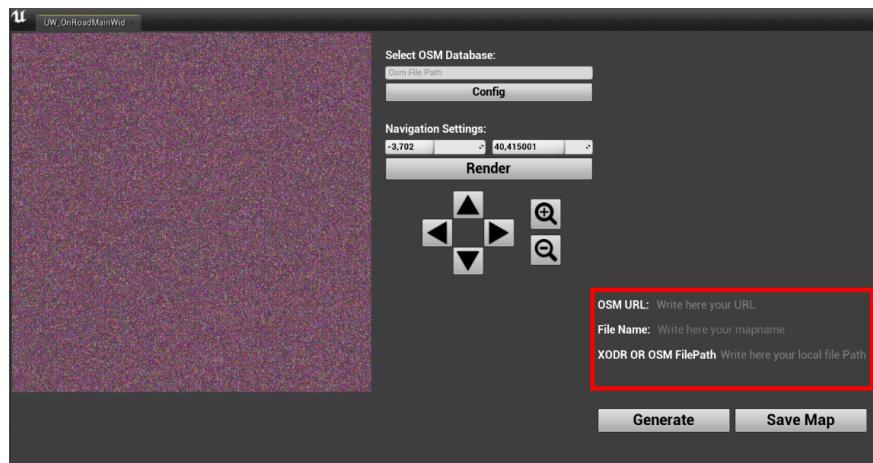


Figure 5: Interfaccia del Digital twin tool

Si consiglia di interagire solo tramite la porzione evidenziata in rosso di figura 5, in quanto il tool è ancora nelle prime fasi di sviluppo e risulta instabile.

Una volta cliccato il tasto "Generate" CARLA si occuperà di generare l'ambiente 3D, e provvederà alla creazione dei file .fbx e .xodr e al loro allineamento all'interno del mondo virtuale.

Il risultato finale si presenta nel modo seguente:



Figure 6: Mappa creata con il Digital twin tool

Il vantaggio principale di questo metodo è la sua facilità d'uso e velocità, CARLA infatti gestisce autonomamente ogni aspetto della creazione della mappa: dalla generazione degli asset, alla creazione della rete stradale in formato OpenDRIVE.

Il problema principale, invece, risiede nella grandezza delle mappe che è possibile realizzare. Per Windows, il limite di estensione massimo è di circa $2 \times 2 \text{ km}^2$, su Linux le aree possono avere una superficie maggiore. Inoltre, come accennato prima, il Digital Twin tool è stato introdotto recentemente, e quindi non è raro il verificarsi di bug o crash improvvisi. Ad esempio può capitare che qualche edificio scompaia dal modello virtuale o che si compenetri con altri elementi di scena. Per questo una revisione manuale della mappa è sempre necessaria.

Ciò che rimane da fare ora è creare il file di configurazione di SUMO (file .sumocfg), per permettere ai due simulatori di operare sulla stesso modello virtuale. Per cominciare è necessario recuperare il file .xodr, creato da CARLA, e convertirlo nei formati .net.xml e .rou.xml. Quindi, posizionarsi all'interno della cartella "root" di CARLA e seguire il percorso seguente:

Unreal > CarlaUE4 > Content > CustomMaps > "Nome della mappa creata" > OpenDRIVE

Qui è localizzato il file in formato OpenDRIVE. Ora, da terminale posizionarsi nella cartella *C : /CARLA 'root'/Co – Simulation/Sumo/util* e digitare il comando seguente:

```
python netconvert_carla.py --output OUTPUT_PATH XODR_FILE_PATH
```

Ciò creerà il file .net.xml nella cartella OUTPUT_PATH.

Scaricare, in questa stessa cartella, gli script *randomTrips.py* e *modifyXML.py* dalla nostra [repository GitHub](#) ed eseguirli in questo ordine:

```
python randomTrips.py -n yourCity.net.xml -r yourGeneratedTraffic.rou.xml --end N ...
    insertion-density N
python modifyXML.py --xml yourGeneratedTraffic.rou.xml
```

Il codice *randomTrips.py* genera un insieme di percorsi casuali per una data rete e gli memorizza in *yourGeneratedTraffic.rou.xml*, mentre *modifyXML.py* modifica il file .rou.xml precedente, aggiungendovi i veicoli usati da CARLA. Ovviamente, le proprietà fisiche dei veicoli di CARLA, per poter essere riconosciute da SUMO, devono essere specificate, grazie al tag *<vType>* all'interno di un altro file .rou.xml. Anch'esso è reperibile dalla nostra repository (*carlavtypes.rou.xml*), ma può essere creato manualmente attraverso il comando *create_sumo_vtypes.py* di CARLA.

Ottenuti tutti file necessari, aprire il .net.xml in Netedit e creare il file di configurazione .sumocfg, salvandolo nella stessa cartella usata sino ad ora. Quindi aprire il .sumocfg tramite editor di testo e apportare le seguenti modifiche al contenuto del tag input:

```
<input>
    <net-file value="yourCity.net.xml"/>
    <route-files value="carlavtypes.rou.xml, yourGeneratedTraffic.rou.xml"...
  />
</input>
</configuration>
```

Il risultato visibile su SUMO è il seguente:



Figure 7: Rete stradale su SUMO

Notare la corrispondenza fra la rete stradale mostrata da SUMO e quella mostrata in figura 6

3.2 Blender

Questo metodo richiede l'installazione aggiuntiva di [Blender](#) e tre dei suoi add-on: [Blosm](#), [Buildify](#) e [LoopTools](#). Essi permettono la creazione dei modelli e degli assets tridimensionali dello scenario che si vuole simulare.

Per avviare la modellazione, una volta aperto Blender, selezionare l'interfaccia del tool Blosm, e importare il terreno della zona di interesse, selezionando la mappa .osm dal proprio pc o online seguendo le indicazioni a schermo, e impostando "terrain" come tipo di dati da ottenere. Il risultato dovrebbe essere una superficie ondulata come in figura 8:

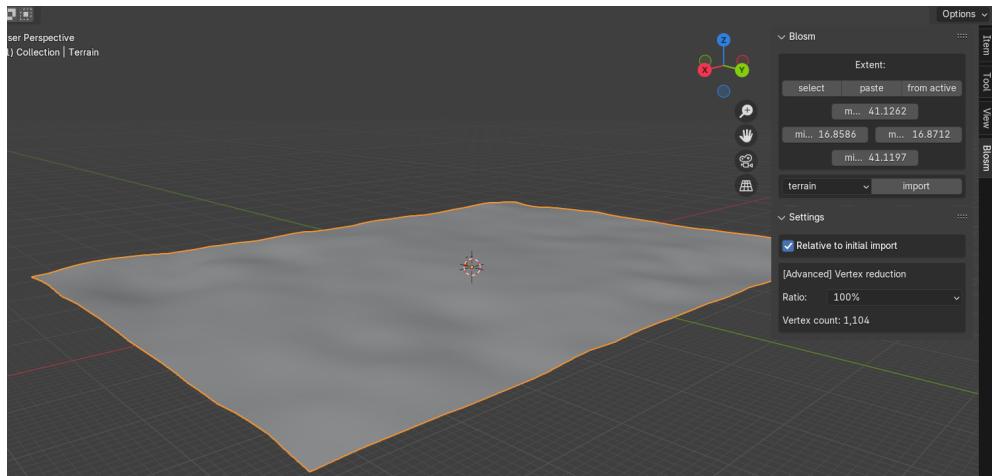


Figure 8: Terreno generato con Blender

Il terreno creato attraverso i dati geografici reali comporta un utilizzo maggiore di risorse computazionali, dato che la sua conta poligonale è maggiore. Inoltre, in CARLA, diventa molto più complesso allineare il percorso stradale con i modelli tridimensionali deglie edifici. Quindi, consigliamo di appiattire la superficie utilizzando l'opzione "Flatten" dell'add-on LoopTools. Per farlo, passare in "edit-mode", cliccare con tasto destro sulla superficie da modificare e selezionare LoopTools e "Flatten". Finita la generazione del terreno, è possibile creare il manto stradale.



Figure 9: Rete stradale creata con Blender

Nel menu di Blosm, cambiare l'opzione "Terrain" in "OpenStreetMap", selezionare la voce "3D" e spuntare tutte le caselle, eccezion fatta per "Import buildings" e "Import railways". Infine cliccare su "Import". Sul terreno appariranno tutte le strade, i parchi e i percorsi pedonali reperibili dalla mappa di OpenStreetMap, come mostrato in figura 9. Per importare anche gli edifici, invece, selezionare la voce "2D", spuntare solo la casella "Import buildings" e nella sezione "[Geometry Nodes]" scrivere il percorso del file di installazione di Buildify, ovvero *buildify_1.0.blend*. Infine cliccare di nuovo su "import".



Figure 10: Edifici generati con Blender

Si ottengono edifici molto dettagliati, e perfettamente allineati con la rete stradale. Per esempio, mappe come quella mostrata in figura 10 possono arrivare a pesare anche oltre una decina di gigabyte. Per diminuirne il peso è possibile ridurre il grado di dettaglio degli edifici non usando Buildify, e quindi selezionando "Import Buildings" quando è impostata la voce "3D". Il risultato sono edifici, che rispettano la planimetria presente sulla mappa, ma sono rappresentati come semplici parallelepipedi.

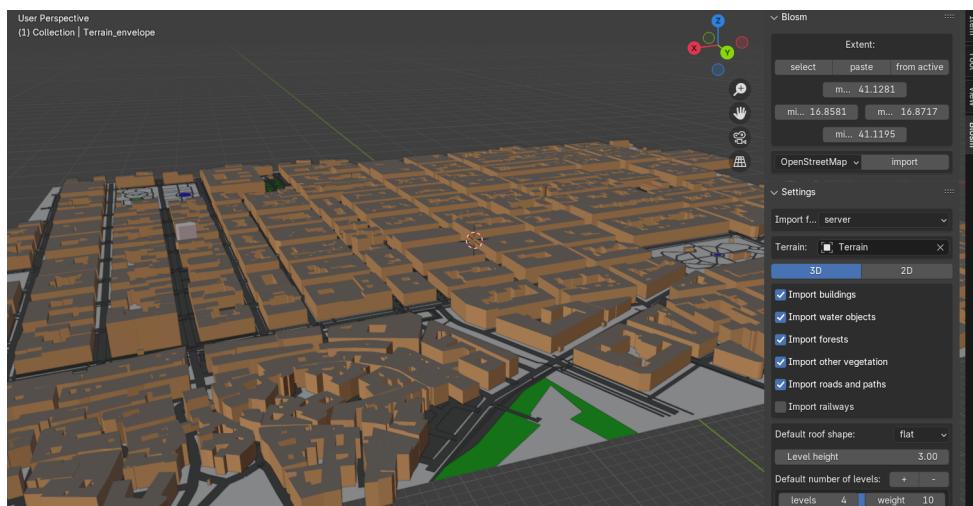


Figure 11: Edifici semplificati con Blender

Ora non resta che esportare, come file .fbx, la mappa appena creata.

La generazione del file .xodr, a partire da un .osm, può avvenire attraverso un qualsiasi strumento di conversione. Consigliamo di usare lo script python messo a disposizione da CARLA, ovvero osm_to_xodr.py nella cartella PythonAPI/util.

L'importazione in CARLA avviene tramite terminale, e una volta spostati i due file .fbx e .xodr nella cartella import, che deve essere necessariamente vuota, digitare il comando:

```
make import GENERATOR="Visual Studio 17 2022"
```

Il modello virtuale verrà creato e salvato nella cartella di CARLA denominata Unreal/CarlaUE4 /Content/map_package/Maps. Lo svantaggio principale che deriva dall'introdurre una mappa con questo metodo è che i modelli tridimensionali non vengono allineati automaticamente con la rete stradale definita dal file in formato OpenDRIVE. Di fatti, sarà necessario spostare manualmente l'intero modello, processo che, soprattutto per mappe molto grandi, può risultare molto complesso. Però, a differenza di quel che accadeva con l'utilizzo del Digital Twin tool, si ha maggiore controllo sull'intero processo di creazione, avendo la possibilità di scegliere il grado di dettaglio di edifici e strade.

La controparte SUMO si realizza nello stesso modo descritto in precedenza, nel capitolo 3.1.

4 CARLA-SUMO co-simulation

Questa sezione tratta l'integrazione di due simulatori avanzati per la mobilità, CARLA e SUMO, e, quindi, del codice creato dagli sviluppatori di CARLA presso il Computer Vision Center del Universitat Autònoma de Barcelona. Questo script consente la sincronizzazione in tempo reale tra i due simulatori, assicurando che eventi come la creazione, la distruzione o lo stato dei veicoli e dei semafori siano coerenti tra SUMO e CARLA. La sincronizzazione avviene tramite l'interfaccia TraCI (Traffic Control Interface), che attraverso un protocollo client-server, permette di interagire con i vari elementi della simulazione in tempo reale.

Il cuore del progetto risiede nella classe `SimulationSynchronization`, che è responsabile del mantenimento della coerenza tra le due simulazioni.

```
class SimulationSynchronization(object):

    def __init__(self, sumo_simulation,
                 carla_simulation,
                 tls_manager='none',
                 sync_vehicle_color=False,
                 sync_vehicle_lights=False):{...}

    def tick(self):{...}

    def close(self):{...}
```

La classe `SimulationSynchronization` è progettata per gestire la sincronizzazione tra le simulazioni di SUMO e CARLA, due piattaforme utilizzate per simulare veicoli. Nel costruttore della classe, vengono accettati vari parametri come le simulazioni SUMO e CARLA, il gestore dei semafori (`tls_manager`) e due opzioni booleane per la sincronizzazione del colore e delle luci dei veicoli tra le due simulazioni.

```
def __init__(...):
    ...

    if tls_manager == 'carla':
        self.sumo.switch_off_traffic_lights()
    elif tls_manager == 'sumo':
        self.carla.switch_off_traffic_lights()

    self.sumo2carla_ids = {} # Contains only actors controlled by sumo.
    self.carla2sumo_ids = {} # Contains only actors controlled by carla.
    BridgeHelper.blueprint_library = self.carla.world
                                .get_blueprint_library()
    BridgeHelper.offset = self.sumo.get_net_offset()
    settings = self.carla.world.get_settings()
    settings.synchronous_mode = True
    settings.fixed_delta_seconds = self.carla.step_length
    self.carla.world.apply_settings(settings)
    traffic_manager = self.carla.client.get_trafficmanager()
    traffic_manager.set_synchronous_mode(True)
    self.vehicle_data = {}
    self.output_file = "vehicles_data.json"
    self.step_counter = 0
    with open(self.output_file, 'w') as file:
        json.dump({}, file)
```

A seconda del valore di `tls_manager`, viene gestito il controllo dei semafori: se impostato su 'carla', i semafori di SUMO vengono disattivati, mentre se impostato su 'sumo', sono quelli di CARLA

a essere spenti. La classe tiene traccia degli ID degli attori controllati da ciascuna simulazione, memorizzandoli rispettivamente negli attributi `sumo2carla_ids` e `carla2sumo_ids`. Successivamente, viene configurata la simulazione di CARLA in modalità sincrona. Viene prima recuperata la libreria dei blueprint di CARLA e salvata in `BridgeHelper.blueprint_library`, e viene ottenuto l'offset della rete di SUMO, memorizzato in `BridgeHelper.offset`. La modalità sincrona di CARLA viene attivata modificando le impostazioni del mondo, impostando `synchronous_mode` a `True` e stabilendo una lunghezza fissa per ogni step della simulazione con `fixed_delta_seconds`, che corrisponde alla durata di ciascun frame. Queste impostazioni vengono poi applicate al mondo di CARLA. Il Traffic Manager di CARLA viene anch'esso configurato per operare in modalità sincrona.

Per quanto riguarda la gestione dei dati, viene creato un file JSON, chiamato `vehicles_data.json`, che serve per memorizzare i dati relativi ai veicoli simulati. All'avvio della simulazione, il file viene inizializzato come vuoto, sovrascrivendo qualsiasi contenuto esistente. Un'altra nuova variabile introdotta nel codice è un contatore degli step di simulazione, `step_counter`, utile nel plotting dei grafici nell'interfaccia, in particolare nel padding degli array `speed` di ogni veicolo.

La funzione `tick()` gestisce il processo di sincronizzazione tra SUMO e CARLA a ogni step della simulazione, con aggiornamenti degli attori e delle informazioni sui veicoli.

```

def tick(self):
    self.step_counter += 1
    self.sumo.tick()
    sumo_spawned_actors = self.sumo.spawned_actors
                - set(self.carla2sumo_ids.values())
    for sumo_actor_id in sumo_spawned_actors:
        self.sumo.subscribe(sumo_actor_id)
        sumo_actor = self.sumo.get_actor(sumo_actor_id)

        carla_blueprint = BridgeHelper.get_carla_blueprint(...)
        if carla_blueprint is not None:
            carla_transform = BridgeHelper.get_carla_transform(...)

            carla_actor_id =
                self.carla.spawn_actor(carla_blueprint, carla_transform)
            if carla_actor_id != INVALID_ACTOR_ID:
                self.sumo2carla_ids[sumo_actor_id] = carla_actor_id
            else:
                self.sumo.unsubscribe(sumo_actor_id)

```

Dopo aver incrementato il contatore degli step della simulazione (`self.step_counter`), viene eseguito un "tick" di SUMO, che aggiorna lo stato della simulazione SUMO. Se nel simulatore SUMO viene generato un veicolo, questo deve essere replicato anche nel simulatore CARLA, e per far ciò è necessario recuperare un "blueprint", utilizzando la funzione `BridgeHelper.get_carla_blueprint()`, per ottenere le informazioni per ricreare il modello 3D del veicolo in questione. Se il blueprint è valido, l'attore viene aggiunto a CARLA con la sua trasformazione (posizione e orientamento) ottenuta da SUMO. Gli ID degli attori sincronizzati, cioè gli ID dei veicoli generati su entrambi i simulatori, vengono quindi memorizzati in un dizionario. È importante notare che, a causa della struttura del codice, l'ID del veicolo non coincide nei due simulatori.

```

for sumo_actor_id in self.sumo.destroyed_actors:
    if sumo_actor_id in self.sumo2carla_ids:
        self.carla.destroy_actor(self.sumo2carla_ids.pop(
            sumo_actor_id))

    # Updating sumo actors in carla.
    for sumo_actor_id in self.sumo2carla_ids:

```

```

carla_actor_id = self.sumo2carla_ids[sumo_actor_id]

sumo_actor = self.sumo.get_actor(sumo_actor_id)
carla_actor = self.carla.get_actor(carla_actor_id)

carla_transform =
    BridgeHelper.get_carla_transform(...)
if self.sync_vehicle_lights:
    carla_lights = BridgeHelper.get_carla_lights_state(...)
else:
    carla_lights = None

self.carla
    .synchronize_vehicle(carla_actor_id, carla_transform, ...
carla_lights)

```

In parallelo, la funzione si occupa di eliminare gli attori in CARLA corrispondenti a quelli distrutti in SUMO. Se un attore in SUMO è stato rimosso, perché arrivato a destinazione, l'attore corrispondente in CARLA deve venir distrutto. Poi, gli attori SUMO presenti in CARLA vengono aggiornati. Questo processo implica l'aggiornamento delle loro trasformazioni e, se necessario, delle luci dei veicoli, sincronizzando tali informazioni tra le due simulazioni.

Se la gestione dei semafori è affidata a SUMO (`self.tls_manager == 'sumo'`), vengono individuati i semafori presenti in entrambe le simulazioni e il loro stato viene aggiornato in CARLA sulla base delle informazioni fornite da SUMO. E' possibile anche il caso opposto, in cui lo stato dei semafori è aggiornato in base alle informazioni fornite da CARLA.

```

try:

    vehicle_ids = traci.vehicle.getIDList()

    for vehicle_id in vehicle_ids:
        carla_actor_id = self.sumo2carla_ids.get(vehicle_id, None)

        if carla_actor_id is not None:
            vehicle_type = traci.vehicle.getTypeID(vehicle_id)
            vehicle_speed = traci.vehicle.getSpeed(vehicle_id)
            ...
            NoiseEmission = traci.vehicle.getNoiseEmission(vehicle_id)

            if carla_actor_id not in self.vehicle_data:
                self.vehicle_data[carla_actor_id] = {
                    "type": vehicle_type,
                    "speed": [round(vehicle_speed,3)],
                    ...
                    "step_count": self.step_counter
                }
            else:
                self.vehicle_data[carla_actor_id].update({
                    "type": vehicle_type,
                    ...
                    "step_count": self.step_counter
                })
                self.vehicle_data[carla_actor_id]["speed"]
                    .append(round(vehicle_speed,3))

    with FileLock(f"{self.output_file}.lock"):
        with open(self.output_file, 'w') as file:
            json.dump(self.vehicle_data, file, indent=4)

```

Terminate le operazioni di sincronizzazione tra i due simulatori, vengono raccolte le informazioni sui veicoli presenti in SUMO tramite il modulo TraCI (Traffic Control Interface). Per ciascun veicolo, vengono ottenute diverse informazioni come tipo, accelerazione, velocità, posizione, ID della corsia e le emissioni (CO₂, CO, PM, ecc.). Questi dati vengono organizzati in un dizionario `self.vehicle_data` e successivamente vengono salvati in un file JSON (`vehicles_data.json`), con un meccanismo di blocco tramite file di lock per garantire che le operazioni di scrittura siano sicure in presenza di accessi concorrenti.

`close()` è l'ultima funzione definita nella classe `SimulationSynchronization`. Questa funzione disattiva la sincronizzazione tra SUMO e CARLA e rimuove gli attori sincronizzati. Prima, ri-configura la simulazione di CARLA passando dalla modalità sincrona all'asincrona, disattivando il tempo fisso per ogni step. Poi, distrugge gli attori sincronizzati, ripulendo la simulazione al termine.

Il main utilizza il modulo `argparse` per gestire gli argomenti della riga di comando e configurare diversi parametri della simulazione e chiamata la funzione `synchronization_loop(arguments)`, che avvia il ciclo di sincronizzazione tra SUMO e CARLA usando i parametri passati.

```
if __name__ == '__main__':
    argparser = argparse.ArgumentParser(description=__doc__)
    ...
    arguments = argparser.parse_args()

    synchronization_loop(arguments)
```

La funzione `synchronization_loop(arguments)` contiene il ciclo `while` su cui iterà la co-simulazione. Inoltre vengono creati gli oggetti `sumo_simulation`, `carla_simulation` e `synchronization`, di tipo `SimulationSynchronization`, su cui si chiama la funzione `tick()` prima descritta.

```
def synchronization_loop(args):
    sumo_simulation = SumoSimulation(args.sumo_cfg_file, args.step_length,
                                      args.sumo_host, args.sumo_port,
                                      args.sumo_gui, args.client_order)
    carla_simulation = CarlaSimulation(args.carla_host, args.carla_port,
                                         args.step_length)

    synchronization = SimulationSynchronization(sumo_simulation,
                                                carla_simulation,
                                                args.tls_manager,
                                                args.sync_vehicle_color,
                                                args.sync_vehicle_lights)

    try:
        while True:
            start = time.time()
            synchronization.tick()
            end = time.time()
            elapsed = end - start
            if elapsed < args.step_length:
                time.sleep(args.step_length - elapsed)

    except KeyboardInterrupt:
        logging.info('Cancelled by user.')

    finally:
        logging.info('Cleaning synchronization')
        synchronization.close()
```

Per concludere, `SumoSimulation` e `CarlaSimulation` sono due classi definite in altri script che includono le definizioni di stati dei semafori e segnali dei veicoli, e contengono classi per gestire i semafori per i rispettivi simulatori. Il gestore dei semafori controlla programmi e fasi dei semafori,

associando segnali a landmark. Sono script contenuti diverse funzioni essenziali per il corretto funzionamento dell'intero sistema co-simulazione, ma essendo molto lunghi e, anche complessi in alcune parti, non verranno approfonditi più di così.

5 Creazione dell'interfaccia/Dashboard

L'interfaccia sviluppata per la co-simulazione tra SUMO e CARLA è stata realizzata per ottimizzare il monitoraggio e l'analisi delle dinamiche dei veicoli durante o al termine dell'esecuzione della simulazione. Implementata tramite il framework Streamlit, fornisce un ambiente interattivo che consente di visualizzare in tempo reale i dati relativi alla velocità dei veicoli e alle condizioni del traffico simulato. L'utente ha la possibilità di selezionare un veicolo specifico e ottenere informazioni dettagliate e informazioni aggiuntive sull'intera simulazione. L'interfaccia è progettata per essere semplice e intuitiva, ma al contempo utile per un'analisi accurata.

Streamlit ha a disposizione molteplici elementi grafici da poter utilizzare per personalizzare ulteriormente la dashboard.

Per le richieste sottoposte, l'interfaccia presentata è più che sufficiente a visualizzare tutte le informazioni utili.

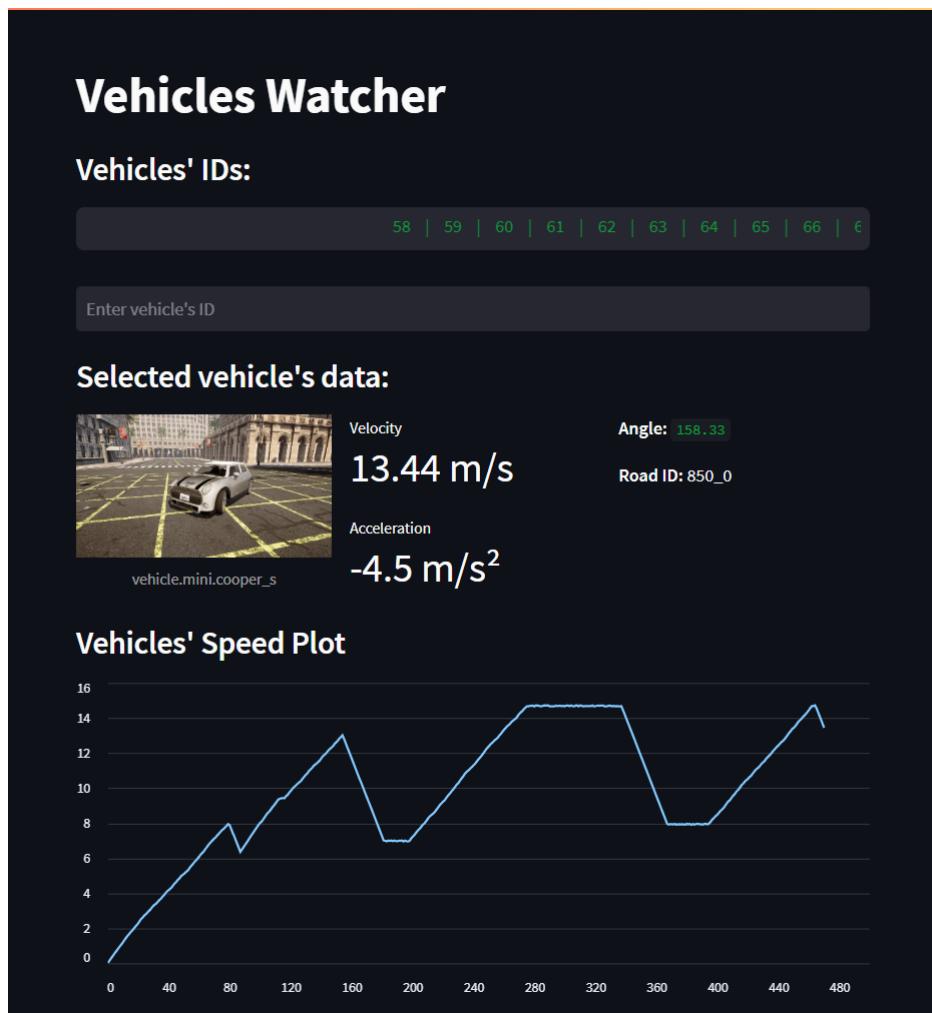


Figure 12: Interfaccia parte 1

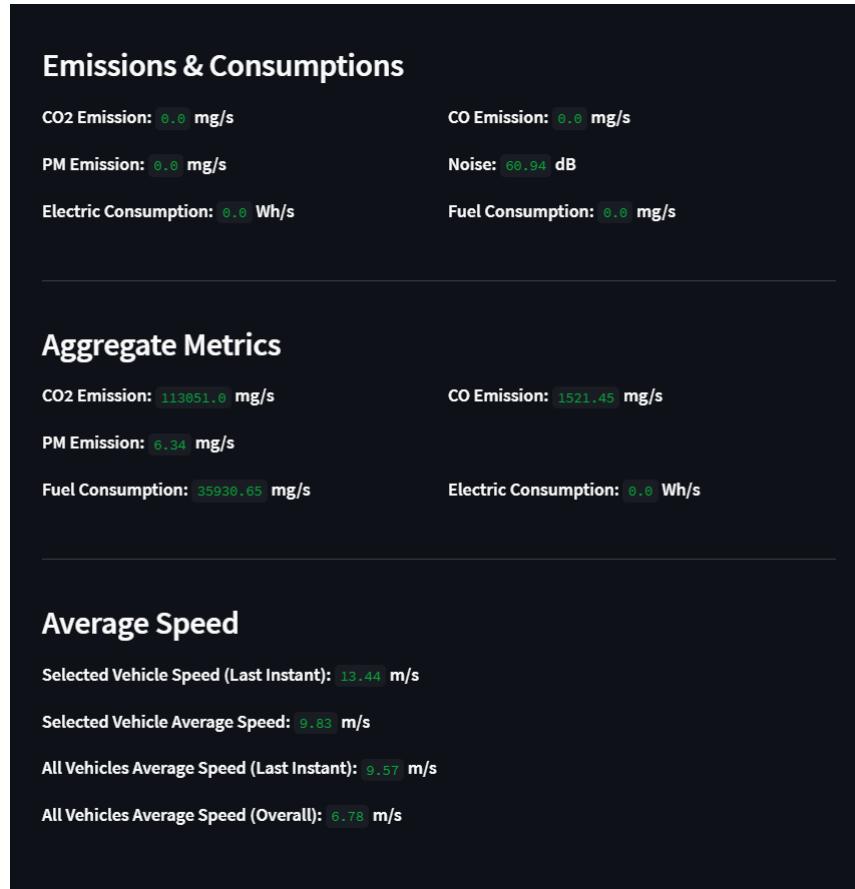


Figure 13: Interfaccia parte 2

I dati utilizzati dall’interfaccia vengono estratti da un file JSON generato durante la co-simulazione tra SUMO e CARLA, contenente informazioni aggiornate in tempo reale sui veicoli.

```
def load_data():
    with FileLock('C:/....CARLA_0.9.15/WindowsNoEditor/Co-Simulation/Sumo/...
    vehicles_data.json.lock'):
        with open('C:/....CARLA_0.9.15/WindowsNoEditor/Co-Simulation/Sumo/...
    vehicles_data.json') as f:
        data = json.load(f)
    records = []
    for key, value in data.items():
        record = {'id': key}
        record.update(value)
        records.append(record)
    return pd.DataFrame(records)
```

Per gestire correttamente la sincronizzazione tra i due simulatori e il file JSON, viene adottato un meccanismo di controllo della concorrenza basato su file di lock. Questo sistema garantisce l’accesso esclusivo al file JSON da parte degli script, impedendo conflitti durante le operazioni di lettura e scrittura. Prima di accedere al file, ogni script python acquisisce un lock che ne garantisce l’uso esclusivo, prevenendo così sovrapposizioni di dati o corruzione delle informazioni. Una volta completata l’operazione, il lock viene rilasciato, consentendo l’accesso al file da parte dell’altro simulatore. Tale metodo assicura la coerenza dei dati contenuti nel file JSON e la correttezza delle informazioni fornite all’interfaccia.

Inizialmente, il metodo di gestione della sincronizzazione tramite file di lock è stato interamente sviluppato manualmente, ricreando da zero il processo di acquisizione e rilascio del lock per garantire l'accesso esclusivo al file JSON. Tuttavia, in una fase successiva, è stata individuata una libreria Python che offre una soluzione già consolidata per la gestione dei file di lock.

Attraverso delle funzioni *get*, i dati della simulazione vengono salvate in delle variabili che vengono aggiornate ad ogni iterazione.

```
def update_metrics(veicolo_selezionato):
    speed = veicolo_selezionato['speed'].values[0]
    acceleration = veicolo_selezionato['acceleration'].values[0]
    angle = veicolo_selezionato['angle'].values[0]
    lane_id = veicolo_selezionato['lane_id'].values[0]

    return speed, acceleration, angle, lane_id
```

```
while True:
    ...
    speed, acceleration, angle, lane_id = update_metrics(
        ... veicolo_selezionato)

    co2_emission, co_emission, pm_emission, electricity_consumption, ...
    fuel_consution, noise = update_additional_metrics(veicolo_selezionato)

    total_co2, total_co, total_pm, total_fuel_consumption, ...
    total_electricity_consumption = aggregate_metrics(data)

    avg_speed_selected, avg_last_speed_all, avg_speed_all = ...
    calculate_speed_metrics(data, veicolo_selezionato)

    cumulative_avg_speed_array = calculate_cumulative_avg_speed(
        ... veicolo_selezionato)
    ...
```

Ad ogni ciclo iterativo, le variabili vengono aggiornate e il nuovo valore aggiornato viene mostrato sull'interfaccia. L'array `cumulative_avg_speed_array` contiene i valori di velocità media ad ogni istante di simulazione del veicolo selezionato.

In particolare l'*i*-esimo elemento dell'array è la velocità media calcolata con gli *i*-esimi valori precedenti del vettore `speed` del veicolo.

Utilizzando Streamlit è possibile anche plottare delle grandezze, e quindi studiarne l'andamento nel tempo. Se non si installano librerie aggiuntive, non è possibile modificare il plot come sarebbe possibile invece su matplotlib, ma è comunque possibile plottare più grandezze sullo stesso grafico per confrontarne l'andamento nel tempo.

```
speed_data = pd.DataFrame({
    'Selected Vehicle Speed': add_padding_to_speed(speed, data),
    'All Vehicles Average Speed': avg_speed_all_vehicles_per_instant
})

st.write("### Vehicle Speed and Average Speed")
st.line_chart(speed_data)
```

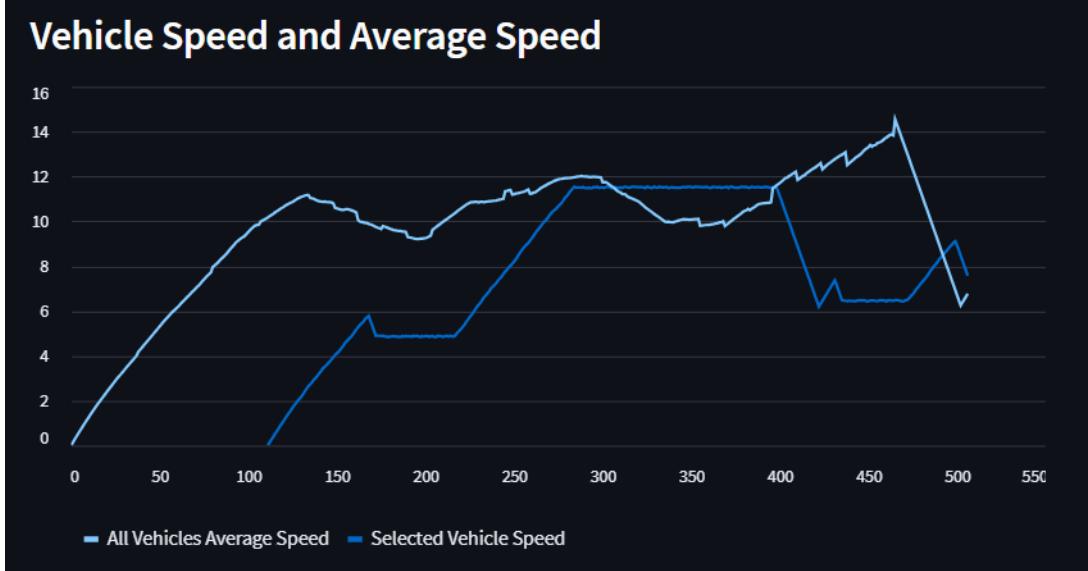


Figure 14: Plot della velocità del veicolo (blu) e della velocità media in simulazione (celeste)

Perchè il plot attraverso `st.line_chart` funzioni, è necessario che tutti gli array nel `pd.DataFrame` abbiano la stessa dimensione.

`All Vehicles Average Speed` è il risultato di questa funzione:

```
def calculate_all_vehicles_avg_speed_per_instant(data):
    all_speeds = []
    for _, vehicle in data.iterrows():
        all_speeds.append(vehicle['speed'])

    max_length = data['step_count'].values[0]
    padded_speeds = np.pad(vehicle_speed, (0, max_length - len(vehicle_speed...)), 'constant', constant_values=np.nan)
    for vehicle_speed in all_speeds:
        speeds_array = np.array(padded_speeds)
    avg_speed_all_vehicles_per_instant = np.nanmean(speeds_array, axis=0)

    return avg_speed_all_vehicles_per_instant
```

Per ciascun veicolo, viene estratto il relativo vettore di velocità e aggiunto alla lista `all_speeds`. La lunghezza massima degli array di velocità è definita dal numero totale di interazioni della co-simulazione, rappresentato da `step_count`. Per garantire che tutti gli array di velocità abbiano la stessa lunghezza, si utilizza la funzione `np.pad` per riempire gli array con valori NaN (Not a Number) fino a raggiungere questa lunghezza massima (`max_length`). Gli array di velocità, ora uniformati in lunghezza, vengono trasformati in un array NumPy denominato `speeds_array`. Su questo array, si applica la funzione `np.nanmean` per calcolare la velocità media di tutti i veicoli per ciascun istante di tempo, ignorando i valori NaN, che indicano dati mancanti. L'array risultante ha una dimensione pari a `step_count`, corrispondente al numero totale di iterazioni nella co-simulazione.

Va notato che la dimensione dell'array di velocità di ciascun veicolo varia a seconda dell'istante in cui viene creato durante la simulazione. Pertanto, per allinearlo alla dimensione del primo array, si applica un padding con NaN, utilizzando la funzione `add_padding_to_speed(speed, data)`.

6 Script per la telecamera

Il concetto di digital twin si basa sulla possibilità di analizzare il funzionamento di un sistema, anche attraverso rappresentazioni visive. In questo contesto, la telecamera di CARLA, denominata "spectator", consente di spostarsi all'interno del sistema 3D che riproduce la città e di osservare, durante la simulazione, il movimento e le interazioni dei veicoli, ad esempio agli incroci. Lo spectator è un attore (actor) che viene generato automaticamente all'avvio del server e può essere controllato facilmente tramite i tasti WASD e il mouse.

Questa modalità risulta utile per esplorare la città, ma poco pratica quando si desidera seguire un veicolo specifico. A tale scopo, sono stati sviluppati script che permettono allo spectator di seguire i veicoli durante le simulazioni.

Sono stati creati tre script, ciascuno progettato per soddisfare una specifica esigenza.

La parte comune a tutti e tre, è la definizione dell'oggetto `spectator`:

```
client = carla.Client('localhost', 2000)
client.set_timeout(10.0)

world = client.get_world()
spectator = world.get_spectator()
```

Il codice stabilisce una connessione con il server CARLA in esecuzione locale sulla porta 2000, creando un client che interagisce con la simulazione. Imposta un timeout di 10 secondi per evitare che il programma rimanga bloccato in caso di ritardi nella comunicazione. Successivamente, il client recupera il "mondo" attualmente attivo nella simulazione, che contiene tutti gli attori e le informazioni ambientali. Infine, accede allo spectator, la telecamera utilizzata per visualizzare la scena in terza persona, permettendo così di manipolare la visuale della simulazione.

6.1 Inseguimento di un veicolo tramite ID

Nel caso in cui sia necessario seguire un veicolo di cui si conosce l'ID, ad esempio per ottenere una conferma visiva delle informazioni visualizzate sull'interfaccia, lo script da utilizzare è `follow_by_id.py`, seguito dall'opzione `-id` e dall'ID del veicolo da inseguire.

```
def follow_vehicle_by_id(world, vehicle_id):
    vehicle = world.get_actor(vehicle_id)
    if vehicle is None:
        print(f"Nessun veicolo trovato con l'ID: {vehicle_id}")
        return

    spectator = world.get_spectator()

    while True:
        transform = vehicle.get_transform()
        offset = carla.Location(x=-10, z=5)
        location = transform.location
            + transform.rotation.get_forward_vector() * offset.x
            + carla.Location(z=offset.z)
        rotation = transform.rotation
        rotation.pitch = -20
        rotation.roll = 0

        spectator.set_transform(carla.Transform(location, rotation))
        time.sleep(0.1)
```

La funzione principale dello script è `follow_vehicle_by_id`, che accetta due parametri: il mondo della simulazione (`world`) e l'ID del veicolo da seguire (`vehicle_id`), quest'ultimo ottenuto tramite

`argparse` dalla variabile `-id`. All'interno di questa funzione, il veicolo con l'ID specificato viene cercato utilizzando il metodo `world.get_actor(vehicle_id)`. Se il veicolo non viene trovato, la funzione stampa un messaggio di errore e si interrompe. In caso contrario, la telecamera dello spettatore, che è l'entità responsabile per la visualizzazione del mondo in terza persona, viene ottenuta attraverso `world.get_spectator()`.

Dopo aver identificato il veicolo, la funzione entra in un ciclo infinito che aggiorna continuamente la posizione e l'orientamento della telecamera per seguirlo. La posizione della telecamera viene calcolata aggiungendo un offset dietro e sopra il veicolo, e l'orientamento viene adattato per migliorare l'inquadratura. Ogni aggiornamento viene applicato tramite `spectator.set_transform()`, con una pausa di 0,1 secondi tra le iterazioni per evitare un aggiornamento eccessivamente rapido.

Per interrompere lo script, e quindi l'inseguimento del veicolo, basta interrompere l'esecuzione con `CTRL+C` sul terminale.

6.2 Inseguimento di veicoli casuali

Il secondo script proposto consente di seguire un veicolo selezionato casualmente nella simulazione di CARLA, utilizzando un thread per aggiornare la posizione della telecamera e permettendo all'utente di cambiare veicolo in tempo reale premendo `ENTER`. Il design basato sui thread garantisce che l'inseguimento avvenga in parallelo all'interazione con l'utente, senza bloccare l'esecuzione del programma.

La classe `VehicleFollower` gestisce l'inseguimento del veicolo nella simulazione. Il costruttore accetta lo spettatore e inizializza un thread per l'inseguimento, controllato dalla variabile `flag`.

```
class VehicleFollower:
    def __init__(self, spectator):
        self.spectator = spectator
        self.following_thread = None
        self.flag = False
```

La funzione principale, `follow_vehicle`, entra in un ciclo continuo che aggiorna la posizione della telecamera dietro e sopra il veicolo, regolando anche la rotazione per puntare verso di esso con un'inclinazione specifica. Il ciclo viene rallentato con una pausa di 50 millisecondi per evitare aggiornamenti troppo frequenti.

```
class VehicleFollower:
    ...
    def follow_vehicle(self, vehicle):
        self.flag = True
        while self.flag:
            transform = vehicle.get_transform()
            offset = carla.Location(x=-10, z=5)
            location = transform.location
                + transform.rotation.get_forward_vector() * offset.x
                + carla.Location(z=offset.z)

            rotation = transform.rotation
            rotation.pitch = -20
            rotation.roll = 0
            self.spectator.set_transform(carla.Transform(location, rotation))

            time.sleep(0.05)
```

La funzione `main`, dopo aver stabilito una connessione con il server CARLA e recuperato lo spettatore, crea un'istanza della classe `VehicleFollower`.

Un ciclo `while` attende che l'utente prema `ENTER`. Quando ciò avviene, l'inseguimento del veicolo corrente viene interrotto, se era attivo, e viene ottenuta la lista di tutti i veicoli presenti nella

simulazione. Se sono disponibili veicoli, viene selezionato casualmente uno di essi e viene avviato un nuovo thread per gestire l'inseguimento del veicolo selezionato.

```
def main():
    ...

    # Connessione al server CARLA e individuazione spectator
    ...

    vehicle_follower = VehicleFollower(spectator)

    try:
        while True:
            input()

            vehicle_follower.stop_following()

            vehicles = world.get_actors().filter('vehicle.*')

            if vehicles:
                vehicle = random.choice(vehicles)
                print(f"La telecamera sta
                      seguendo il veicolo con ID: {vehicle.id}")

                vehicle_follower.following_thread =
                    threading.Thread(target=vehicle_follower.follow_vehicle,
                                     args=(vehicle,))
                vehicle_follower.following_thread.start()
```

La funzione `stop_following()`, definita nella classe `VehicleFollower`, controlla se è attivo un thread di inseguimento (`following_thread`). Se presente, imposta la variabile `flag` su False per interrompere il ciclo di inseguimento all'interno del thread. Infine, utilizza il metodo `join()` per attendere la conclusione del thread, assicurando così che l'inseguimento venga fermato in modo ordinato.

6.3 Inseguimento con widget tk

L'ultimo script proposto per l'inseguimento di veicoli è `camera_with_tk.py`. Questo codice presenta le caratteristiche dei due script prima descritti e aggiunge la possibilità di osservare le informazioni utili del veicolo inseguito direttamente su un widget, oltre che sull'interfaccia Streamlit. È possibile inseguire veicoli casuali ogniqualvolta si prema ENTER oppure, se lo script viene fatto partire con `-id ID`, un veicolo specifico.

La caratteristica che lo contraddistingue dai due script precedenti è l'uso della libreria `tkinter` che permette di creare e personalizzare un widget, utilizzato, in questa applicazione, per mostrare le informazioni più importanti del veicolo inseguito.

Come negli altri due script, si crea una connessione al server Carla e se ne ricavano gli attori, ed in più si inizializzano due variabili globali (`flag` e `vehicle`), per far funzionare il resto del codice. Le informazioni che sono mostrate sul widget tk provengono del file JSON che viene generato ed aggiornato durante la co-simulazione tramite lo script `run_syncro.py`.

```
def read_vehicle_info_from_json(file_path, vehicle_id):
    try:
        with open(file_path, 'r') as file:
            data = json.load(file)
            vehicle_info = data.get(str(vehicle_id))
            if vehicle_info:
                return vehicle_info
```

```

        else:
            print(f"Veicolo con ID {vehicle_id} non trovato nel file JSON...")
        return None
    except Exception as e:
        print(f"Errore durante la lettura del file JSON: {e}")
    return None

```

Questa funzione legge le informazioni del veicolo dal file JSON, cercando l'ID del veicolo specificato. Se il veicolo viene trovato nel file, le informazioni vengono restituite, altrimenti viene stampato un messaggio di errore. Questo permette di ottenere dinamicamente i dati del veicolo dalla simulazione e mostrare informazioni aggiornate.

```

root = tk.Tk()
root.title("Vehicle Info")

id_label = tk.Label(root, text="ID: N/A", font=("Helvetica", 16))
id_label.pack(pady=5)

# Creazione delle altre etichette: type_label, speed_label, ecc.)

root.bind('<Return>', lambda event: on_enter_press(event, world, spectator, ...
    speed_label, acceleration_label, type_label, lane_id_label))

...

if args.id:
    follow_vehicle_by_id(world, spectator, args.id, speed_label, ...
        acceleration_label, type_label, lane_id_label, id_label)

else:
    root.bind('<Return>', lambda event: on_enter_press(event, world, ...
        spectator, speed_label, acceleration_label, type_label, lane_id_label, ...
        id_label))

...

root.after(100, update_camera_and_ui, world, spectator, speed_label, ...
    acceleration_label, type_label, lane_id_label, id_label)

root.mainloop()

```

Nel main si crea un'interfaccia grafica utilizzando Tkinter per visualizzare le informazioni di un veicolo nella simulazione CARLA. Viene inizializzata una finestra (`root`) con il titolo "Vehicle Info" e vengono aggiunte diverse etichette (`id_label`, `speed_label`, ecc.) per mostrare i dettagli del veicolo come ID, tipo, velocità, e accelerazione. L'interfaccia risponde all'input dell'utente: se l'ID del veicolo è fornito come argomento, inizia automaticamente a seguire il veicolo specificato tramite la funzione `follow_vehicle_by_id`. Se non è specificato un ID, l'utente può premere il tasto ENTER (Return) per avviare l'inseguimento di un veicolo casuale, tramite la funzione `on_enter_press()`. La funzione `root.after` viene utilizzata per aggiornare continuamente la telecamera e l'interfaccia ogni 100 millisecondi, mantenendo le informazioni del veicolo sempre aggiornate. Infine, il ciclo principale di Tkinter (`root.mainloop()`) gestisce la finestra e gli eventi.

```

def follow_vehicle_by_id(world, spectator, vehicle_id, speed_label, ...
    acceleration_label, type_label, lane_id_label, id_label):
    global flag, vehicle
    if vehicle_id is not None:
        vehicle = world.get_actor(vehicle_id)

```

```

if vehicle is not None:
    flag = True
    print(f"La telecamera sta seguendo il veicolo con id: {vehicle.id}...")
)
    update_camera_and_ui(world, spectator, speed_label, ...
acceleration_label, type_label, lane_id_label, id_label)
else:
    print(f"Nessun veicolo trovato con l'ID: {vehicle_id}")

```

```

def on_enter_press(event, world, spectator, speed_label, acceleration_label, ...
type_label, lane_id_label):
    vehicles = world.get_actors().filter('vehicle.*')
    if vehicles:
        vehicle = random.choice(vehicles)
        print(f"La telecamera sta seguendo il veicolo con id: {vehicle.id}")
        flag = True
        update_camera_and_ui(world, spectator, speed_label, ...
acceleration_label, type_label, lane_id_label)

```

Indipendentemente dalla modalità di utilizzo, lo script utilizza la funzione `update_camera_and_ui()` per aggiornare la posizione dello spectator e le informazioni mostrate sul widget. Per spostare lo spectator si usano le stesse operazioni mostrate in precedenza mentre per aggiornare le informazioni si usa la funzione `update_ui()`

```

def update_ui(file_path, vehicle_id, speed_label, acceleration_label, ...
type_label, lane_id_label, id_label):
    vehicle_info = read_vehicle_info_from_json(file_path, vehicle_id)
    if vehicle_info:
        try:
            speed_array = vehicle_info.get("speed", [])
            acceleration = vehicle_info.get("acceleration", "N/A")
            vehicle_type = vehicle_info.get("type", "N/A")
            lane_id = vehicle_info.get("lane_id", "N/A")
            speed = speed_array[-1] if speed_array else "N/A"

            id_label.config(text=f"ID: {vehicle_id}")
            type_label.config(text=f"Type: {vehicle_type}")
            speed_label.config(text=f"Speed: {speed:.2f} km/h")
            lane_id_label.config(text=f"Lane ID: {lane_id}")
        except Exception as e:
            print(f"Errore durante l'aggiornamento dell'UI: {e}")

```

Questa funzione aggiorna l'interfaccia utente con le informazioni del veicolo lette dal file JSON. Viene estratta la velocità (prendendo l'ultimo valore disponibile dall'array delle velocità), l'accelerazione, il tipo di veicolo e l'ID della corsia, quindi questi dati vengono visualizzati sulle etichette dell'interfaccia (Tkinter).

7 Descrizione del codice

In questa sezione verranno spiegate le parti di codice cruciali per il corretto funzionamento dell'intero software:

7.1 run_syncro.py

Questo script sincronizza le simulazioni di traffico tra CARLA e SUMO, assicurando che i veicoli, i semafori e le informazioni siano allineati in entrambi gli ambienti. I dati dei veicoli, inclusi accelerazione, velocità e emissioni, vengono raccolti e salvati in un file JSON per ogni tick della simulazione. E' importante far notare che questo codice è una versione modificata di uno script già sviluppato da CARLA.

7.1.1 Inizializzazione della classe SimulationSynchronization (Righe 70-98)

```
class SimulationSynchronization(object):
    def __init__(self, sumo_simulation, carla_simulation, tls_manager='none', ...
                 sync_vehicle_color=False, sync_vehicle_lights=False):
        self.sumo = sumo_simulation
        self.carla = carla_simulation
        self.tls_manager = tls_manager
        self.sync_vehicle_color = sync_vehicle_color
        self.sync_vehicle_lights = sync_vehicle_lights
        ...
        # Creazione del file per salvare i dati dei veicoli
        self.vehicle_data = {}
        self.output_file = "vehicles_data.json"

        # Pulizia del file JSON all'avvio
        with open(self.output_file, 'w') as file:
            json.dump({}, file)
```

La classe `SimulationSynchronization` gestisce la sincronizzazione tra SUMO e CARLA. In questo costruttore, vengono salvate le configurazioni iniziali per gestire i veicoli e i semafori. Inoltre, viene creato un file JSON per salvare i dati dei veicoli.

7.1.2 Sincronizzazione delle informazioni dei veicoli (Righe 275-309)

```
try:
    vehicle_ids = traci.vehicle.getIDList()
    for vehicle_id in vehicle_ids:
        carla_actor_id = self.sumo2carla_ids.get(vehicle_id, None)
        if carla_actor_id is not None:
            vehicle_type = traci.vehicle.getTypeID(vehicle_id)
            vehicle_speed = traci.vehicle.getSpeed(vehicle_id)
            ...
            self.vehicle_data[carla_actor_id].update({
                "type": vehicle_type,
                "speed": [round(vehicle_speed, 3)],
                "position": vehicle_position,
                ...
            })
    with FileLock(f"{self.output_file}.lock"):
        with open(self.output_file, 'w') as file:
            json.dump(self.vehicle_data, file, indent=4)
```

Ogni tick del simulatore, questa sezione raccoglie dati sui veicoli da SUMO (velocità, tipo, emissioni ecc.) e li aggiorna nel file JSON, proteggendo il processo con un FileLock.

7.1.3 Ciclo di sincronizzazione principale (Righe 389-409)

```
def synchronization_loop(args):
    sumo_simulation = SumoSimulation(args.sumo_cfg_file, args.step_length, ...)
    ...
    carla_simulation = CarlaSimulation(args.carla_host, args.carla_port, args...
    .step_length)
    synchronization = SimulationSynchronization(sumo_simulation, ...
    carla_simulation, ...)
    try:
        while True:
            start = time.time()
            synchronization.tick()
            elapsed = time.time() - start
            if elapsed < args.step_length:
                time.sleep(args.step_length - elapsed)
    except KeyboardInterrupt:
        logging.info('Cancelled by user.')
    finally:
        synchronization.close()
```

Questo è il loop principale che gestisce la sincronizzazione tra SUMO e CARLA. Viene eseguito un tick ogni ciclo, che esegue l'aggiornamento di veicoli e semafori.

7.2 interface.py

Il codice è un'applicazione Streamlit che carica i dati di vari veicoli da un file JSON in tempo reale, mostrandone le metriche. L'applicazione consente di selezionare un veicolo e visualizzarne le informazioni specifiche, come velocità, accelerazione, angolo e ID della corsia. Vengono inoltre visualizzate metriche ambientali come emissioni di CO2, CO e PM, nonché consumi di elettricità e carburante.

7.2.1 Funzione per caricare i dati dal file JSON (Righe 11-20)

```
def load_data():
    with FileLock('C:/...../CARLA_0.9.15/WindowsNoEditor/Co-Simulation/Sumo/...
    vehicles_data.json.lock'):
        with open('C:/...../CARLA_0.9.15/WindowsNoEditor/Co-Simulation/Sumo/...
    vehicles_data.json') as f:
        data = json.load(f)
    records = []
    for key, value in data.items():
        record = {'id': key}
        record.update(value)
        records.append(record)
    return pd.DataFrame(records)
```

Questa funzione utilizza un lock per caricare i dati da un file JSON specificato. Dopo aver letto il file, converte i dati in un DataFrame di Pandas, semplificando ulteriori manipolazioni e analisi.

7.2.2 Aggiornamento delle metriche e visualizzazione dei dati (Righe 128-196)

```

while True:
    data = load_data()

    ...

    if not veicolo_selezionato.empty:
        st.subheader('Selected vehicle\'s data:')

        vehicle_type = veicolo_selezionato['type'].values[0]

        speed, acceleration, angle, lane_id = update_metrics(
            veicolo_selezionato)
        co2_emission, co_emission, pm_emission, electricity_consumption, ...
        fuel_consumption, noise = update_additional_metrics(veicolo_selezionato)
        total_co2, total_co, total_pm, total_fuel_consumption, ...
        total_electricity_consumption = aggregate_metrics(data)
        avg_speed_selected, avg_last_speed_all, avg_speed_all = ...
        calculate_speed_metrics(data, veicolo_selezionato)

        ...

    time.sleep(1)

```

In questa sezione, il codice esegue un ciclo continuo che aggiorna le metriche dei veicoli selezionati ogni secondo. Se il veicolo selezionato esiste, vengono estratte e visualizzate varie metriche. Viene mostrata un'immagine del veicolo, seguita da metriche relative alla velocità, alle emissioni e al consumo. Inoltre, vengono visualizzate metriche relative all'intera simulazione, come emissioni nell'intera area o velocità medie.

7.3 camera.py

Il codice si connette al server CARLA e permette di selezionare casualmente un veicolo presente nella simulazione per seguirlo con la telecamera. Dopo aver premuto ENTER, la telecamera inizia a seguire il veicolo, mantenendosi dietro e sopra di esso. L'utente può cambiare veicolo premendo nuovamente ENTER, e il processo continua finché non viene interrotto con CTRL + C.

7.3.1 Connessione al server CARLA e impostazioni iniziali (Righe 6-13)

```

client = carla.Client('localhost', 2000)
client.set_timeout(10.0) # Timeout di 10 secondi per evitare che il ...
# programma si blocchi

world = client.get_world()
blueprint_library = world.get_blueprint_library()

spectator = world.get_spectator()

```

Il codice inizia con la connessione al server CARLA locale, utilizzando il client carla.Client. Viene impostato un timeout di 10 secondi per evitare che il programma si blocchi in caso di problemi di connessione. La variabile world rappresenta il mondo corrente della simulazione e blueprint_library contiene tutti i modelli disponibili nel mondo. Infine, il riferimento all'attore "spettatore" (spectator) è recuperato per gestire la telecamera.

7.3.2 Ciclo principale per la selezione e il controllo della telecamera (Righe 16-27)

```

while True:
    input() # Attendi che l'utente prema ENTER

    vehicles = world.get_actors().filter('vehicle.*')

    if vehicles:
        vehicle = random.choice(vehicles)
        print(f"La telecamera sta seguendo il veicolo con id: {vehicle.id}")
        flag=True

```

Un ciclo while True attende che l'utente prema il tasto ENTER. Una volta premuto, vengono recuperati tutti gli attori di tipo veicolo presenti nella simulazione tramite il metodo `world.get_actors().filter('vehicle.*')`, che filtra solo gli attori con blueprint di tipo veicolo. Se ci sono veicoli nella simulazione, viene selezionato casualmente un veicolo da seguire usando `random.choice()`. Il programma segnala l'ID del veicolo selezionato e imposta la variabile `flag` su True per controllare il tracking della telecamera.

7.3.3 Posizionamento e orientamento della telecamera (Righe 30-41)

```

while flag:
    transform = vehicle.get_transform()

    offset = carla.Location(x=-10, z=5)

    location = transform.location + transform.rotation.get_forward_vector() *...
               offset.x + carla.Location(z=offset.z)

    rotation = transform.rotation
    rotation.pitch = -20
    rotation.roll = 0

    spectator.set_transform(carla.Transform(location, rotation))

```

In questo blocco, la telecamera viene posizionata per seguire il veicolo selezionato. Utilizza la posizione e la rotazione correnti del veicolo (`vehicle.get_transform()`) per calcolare una nuova posizione della telecamera con un offset dietro e sopra il veicolo. L'orientamento della telecamera viene impostato con una rotazione che inclina la visuale leggermente verso il basso (`rotation.pitch = -20`).

7.4 camera_with_tk.py

Il codice permette di selezionare e seguire un veicolo nella simulazione CARLA, visualizzando i suoi dati (velocità, accelerazione, tipo e ID di corsia) in un'interfaccia grafica. Si può selezionare il veicolo tramite un ID fornito o casualmente tramite un tasto.

7.4.1 Funzione per leggere informazioni da un file JSON (Righe 14-27)

```

def read_vehicle_info_from_json(file_path, vehicle_id):
    try:
        with open(file_path, 'r') as file:
            data = json.load(file)
            vehicle_info = data.get(str(vehicle_id))
            if vehicle_info:
                return vehicle_info
            else:

```

```

        print(f"Veicolo con ID {vehicle_id} non trovato nel file JSON...")
    .")
    return None
except Exception as e:
    print(f"Errore durante la lettura del file JSON: {e}")
    return None

```

Questa funzione legge i dati dal file JSON e cerca le informazioni del veicolo specificato dall'ID. Se trova il veicolo, restituisce le sue informazioni, altrimenti stampa un messaggio d'errore.

7.4.2 Funzione per aggiornare l'interfaccia utente (Righe 29-45)

```

def update_ui(file_path, vehicle_id, speed_label, acceleration_label, ...
type_label, lane_id_label, id_label):
    vehicle_info = read_vehicle_info_from_json(file_path, vehicle_id)
    if vehicle_info:
        try:
            speed = vehicle_info.get("speed)[-1], "N/A")
            acceleration = vehicle_info.get("acceleration", "N/A")

            ...

        except Exception as e:
            print(f"Errore durante l'aggiornamento dell'UI: {e}")

```

Questa funzione aggiorna i dati nell'interfaccia utente (Tkinter) con le informazioni del veicolo (velocità, accelerazione, tipo e ID di corsia). Prima recupera i dati dal file JSON tramite la funzione `read_vehicle_info_from_json`.

7.4.3 Funzione per aggiornare la telecamera e l'interfaccia utente (Righe 47-67)

```

def update_camera_and_ui(world, spectator, speed_label, acceleration_label, ...
type_label, lane_id_label, id_label):
    global flag, vehicle
    if flag and vehicle is not None:
        try:
            transform = vehicle.get_transform()
            offset = carla.Location(x=-10, z=5)
            location = transform.location + transform.rotation...
            get_forward_vector() * offset.x + carla.Location(z=offset.z)
            rotation = transform.rotation
            rotation.pitch = -20
            rotation.roll = 0
            spectator.set_transform(carla.Transform(location, rotation))
            update_ui(json_file_path, vehicle.id, speed_label, ...
            acceleration_label, type_label, lane_id_label, id_label)
        except Exception as e:
            print(f"Errore durante l'aggiornamento della telecamera: {e}")

    root.after(100, update_camera_and_ui, world, spectator, speed_label, ...
    acceleration_label, type_label, lane_id_label, id_label)

```

Questa funzione controlla il posizionamento della telecamera per seguirlo in base alla sua posizione attuale. Inoltre, aggiorna l'interfaccia utente con le informazioni del veicolo. Se il flag è attivo, la telecamera segue il veicolo selezionato, e il ciclo viene ripetuto ogni 100 millisecondi usando `root.after`.

7.4.4 Selezione del veicolo da seguire tramite ID (Righe 69-83)

```
def follow_vehicle_by_id(world, spectator, vehicle_id, speed_label, ...
    acceleration_label, type_label, lane_id_label, id_label):
    global flag, vehicle
    if vehicle_id is not None:
        vehicle = world.get_actor(vehicle_id)
        if vehicle is not None:
            flag = True
            print(f"La telecamera sta seguendo il veicolo con id: {vehicle.id}...")
        else:
            print(f"Nessun veicolo trovato con l'ID: {vehicle_id}")
    else:
        on_enter_press(None, world, spectator, speed_label, ...
            acceleration_label, type_label, lane_id_label, id_label)
```

Questa funzione cerca un veicolo con un ID specificato e, se trovato, imposta il flag per seguire quel veicolo.

7.4.5 Selezione casuale di un veicolo da seguire (Righe 85-94)

```
def on_enter_press(event, world, spectator, speed_label, acceleration_label, ...
    type_label, lane_id_label, id_label):
    global vehicle, flag
    vehicles = world.get_actors().filter('vehicle.*')
    if vehicles:
        vehicle = random.choice(vehicles)
        print(f"La telecamera sta seguendo il veicolo con id: {vehicle.id}")
        flag = True
        update_camera_and_ui(world, spectator, speed_label, ...
            acceleration_label, type_label, lane_id_label, id_label)
    else:
        print("Nessun veicolo trovato nella simulazione.")
```

Questa funzione viene chiamata quando l'utente preme il tasto ENTER. Seleziona casualmente un veicolo tra quelli presenti nella simulazione e imposta il flag per seguire quel veicolo.

7.5 modifyXML.py

Il codice modifica un file XML che contiene elementi <vehicle>, aggiungendo un attributo type a ciascun veicolo con un valore casuale da una lista predefinita di tipi di veicoli. Il risultato è un nuovo file XML chiamato modified_<input_file>.xml con i nuovi attributi di tipo veicolo.

7.5.1 Funzione per aggiungere un tipo di veicolo casuale nel file XML (Righe 23-33)

```
def add_vehicle_type(xml_file, output_file):
    tree = ET.parse(xml_file)
    root = tree.getroot()

    for vehicle in root.findall('vehicle'):
        vehicle_type = random.choice(vehicle_types)
        vehicle.set('type', vehicle_type)

    tree.write(output_file)
```

Questa funzione svolge il compito principale di modificare il file XML: carica e analizza il file XML usando ET.parse() trovando tutti gli elementi <vehicle> con root.findall('vehicle'). Quindi assegna un tipo casuale da vehicle_types a ciascun veicolo usando random.choice() e salva il file modificato in output_file usando tree.write().

8 Conclusioni

La co-simulazione tra CARLA e SUMO rappresenta un passo significativo verso la creazione di ambienti di simulazione sempre più realistici e complessi per lo studio del traffico veicolare e delle interazioni urbane. L'integrazione tra i due simulatori permette di combinare le capacità avanzate di gestione del traffico di SUMO con il realismo fisico e grafico offerto da CARLA, creando un contesto di sperimentazione versatile e completo. Un primo passo verso una maggiore autenticità nella riproduzione degli ambienti, potrebbe consistere nell'utilizzo del software RoadRunner, capace di offrire una migliore integrazione fra le funzionalità di creazione delle mappe e quelle simulative di CARLA.

L'implementazione degli script per la gestione della telecamera, che segue specificamente un veicolo durante la simulazione, ha permesso di migliorare l'osservazione e l'analisi dei comportamenti in situazioni reali, rendendo il sistema adatto a testare algoritmi di guida autonoma e scenari complessi di traffico. L'interfaccia sviluppata facilita il controllo e la visualizzazione delle informazioni, rendendo l'interazione tra i simulatori fluida e accessibile. Vi sono, tuttavia, margini di miglioramento per quel che riguarda la quantità di informazioni visualizzate e l'interazione con l'utente. Ad esempio, sarebbe possibile aggiungere una funzionalità che permetta agli utenti di richiedere autonomamente quale dato mostrare.

In conclusione, grazie a questa implementazione, è possibile non solo studiare i singoli veicoli e i loro comportamenti dinamici, ma anche monitorare in tempo reale l'intero flusso del traffico, consentendo una valutazione più accurata delle prestazioni del sistema. L'approccio proposto costituisce una base solida per future estensioni e migliorie, come l'introduzione di veicoli autonomi o l'implementazione di scenari di traffico più complessi. Si potrebbero utilizzare tecniche più complesse per il controllo del traffico, come il reinforcement learning, per permettere la simulazione di scenari che prevedano ostacoli mobili o statici, rappresentati da un oggetto creato e controllato da CARLA.

References

- [1] Angira Sharma et al. "Digital Twins: State of the art theory and practice, challenges, and open research questions". In: *Journal of Industrial Information Integration* 30 (2022), p. 100383. ISSN: 2452-414X. DOI: <https://doi.org/10.1016/j.jii.2022.100383>. URL: <https://www.sciencedirect.com/science/article/pii/S2452414X22000516>.
- [2] Aidan Fuller et al. "Digital Twin: Enabling Technologies, Challenges and Open Research". In: *IEEE Access* PP (May 2020), pp. 1–1. DOI: [10.1109/ACCESS.2020.2998358](https://doi.org/10.1109/ACCESS.2020.2998358).
- [3] Zijin Wang et al. "Towards Next Generation of Pedestrian and Connected Vehicle In-the-Loop Research: A Digital Twin Co-Simulation Framework". In: *IEEE Transactions on Intelligent Vehicles* 8.4 (2023), pp. 2674–2683. DOI: [10.1109/TIV.2023.3250353](https://doi.org/10.1109/TIV.2023.3250353).