Politecnico di Torino

**Politecnico di Torino**

Ingegneria Informatica

A.a. 2025/2026

Sessione di laurea Marzo 2026

# Colmare il gap tra dataset sintetici e reali in Computer Vision

## Un confronto tra diversi rendering engine

Relatori:

prof. Andrea Sanna

prof. Federico Manuri

ing. Luca Francesco Rossi

Candidato:

Lorenzo Pindinello

# Sommario

L'evoluzione della Computer Vision nell'ultimo decennio è stata segnata dal passaggio dall'ingegnerizzazione manuale delle feature all'apprendimento automatico tramite Deep Learning, introducendo una forte dipendenza da dataset reali, vasti e accuratamente etichettati. L'analisi dei benchmark standard, come Microsoft COCO, evidenzia limiti strutturali quali l'elevata onerosità delle annotazioni manuali, i *bias* ambientali e la cronica assenza di scenari critici per la robustezza dei modelli.

Per rispondere a questa scarsità di dati, la presente ricerca adotta il paradigma *Sim-to-Real*, utilizzando motori grafici per generare dataset sintetici dotati di annotazioni automatizzate. Tuttavia, i dati sintetici sono visivamente differenti rispetto ai dati reali, risultando talvolta meno efficaci per l'addestramento di modelli destinati a operare nel mondo reale. In questa tesi viene affrontata la sfida del *Domain Gap* coniugando l'elevato fotorealismo del *Path Tracing* con tecniche di *Domain Randomization*, variando proceduralmente l'illuminazione e le inquadrature per favorire la generalizzazione della rete. Oltre ai vantaggi tecnici, i dati sintetici garantiscono una tracciabilità dei dati certificata e rispondono ai requisiti del GDPR tramite il principio della *Privacy by Design*.

L'infrastruttura tecnologica sviluppata si basa sul framework **BlenderProc**, utilizzato per orchestrare l'intero processo di generazione dei dati tramite script Python per il campionamento stocastico delle telecamere e l'estrazione di annotazioni *pixel-perfect* pronte per l'addestramento del modello **YOLO26**. All'interno di questa pipeline sono stati impiegati diversi motori di rendering per la generazione delle immagini sintetiche: **Blender EEVEE**, scelto per la sua elevata velocità di rendering, **Unreal Engine 5**, sfruttato tramite la pipeline di **Deferred Rendering** e il **Path Tracer**, e infine **Blender Cycles**, utilizzato come riferimento per il rendering fisicamente accurato e ad alto fotorealismo.

Dal punto di vista operativo, sono state generate 10.000 immagini per ogni motore, mantenendo una rigida coerenza tra i motori nelle pose e nell'illuminazione HDRI. L'analisi delle performance ha evidenziato divari significativi: Cycles ha richiesto 146 ore totali, mentre EEVEE e Unreal Engine hanno completato il compito tra le 5 e le 10 ore. In particolare, Unreal Path Tracer è stato testato sia con una configurazione ottimizzata a basso campionamento che con una ad alta

fedeltà, permettendo di mappare l'efficienza dei diversi approcci.

In conclusione, i risultati ottenuti suggeriscono che un maggiore livello di fotorealismo può contribuire a migliorare la qualità delle predizioni del modello. Tuttavia, l'incremento estremo della qualità visiva comporta costi computazionali molto elevati a fronte di benefici limitati nelle metriche di object detection. L'analisi evidenzia quindi un chiaro *trade-off* operativo tra fedeltà visiva e scalabilità del processo di generazione dei dati. Nel contesto osservato, la produzione di dataset più ampi e diversificati tramite motori di rendering veloci o configurazioni a basso campionamento è sembrata complessivamente più efficace rispetto alla generazione di dataset ridotti ma fotorealistici. Il binomio Unreal Engine con Path Tracer a basso campionamento ha mostrato, in questo contesto sperimentale, un buon compromesso tra correttezza fisica e velocità produttiva. Infine, i risultati del **Fine-Tuning** supportano, nel contesto analizzato, l'utilità del dato sintetico come base di pre-training: l'integrazione di sole 1.000 immagini reali ha triplicato le performance rispetto al solo addestramento sintetico, mentre il pre-training sintetico, seguito da 12.000 immagini reali, ha superato l'addestramento da zero sulla detection, suggerendo che l'approccio possa rappresentare una strategia utile di data augmentation preliminare.

# Indice

**Elenco delle figure** IX

**1 Introduzione** 1
- 1.1 Oggetto e Motivazione del lavoro . . . 1
- 1.2 Obiettivi e Scopo della ricerca . . . 2
- 1.3 Relazione con altri lavori . . . 2
- 1.4 Metodo seguito e Struttura dell'elaborato . . . 3

**2 Stato dell'arte** 5
- 2.1 Object Detection . . . 5
  - 2.1.1 L'approccio Two-Stage: R-CNN e la sua evoluzione . . . 5
  - 2.1.2 L'approccio One-Stage: YOLO (You Only Look Once) . . . 7
  - 2.1.3 Il Benchmark standard: Microsoft COCO . . . 9
- 2.2 Soluzione dei dataset sintetici: L'approccio Sim-to-Real . . . 10
  - 2.2.1 Dai videogiochi ai simulatori dedicati . . . 10
- 2.3 Colmare il gap tra sintetico e reale . . . 12
  - 2.3.1 Realismo visivo e Domain Randomization . . . 12
- 2.4 Aspetti etici e legali . . . 14
  - 2.4.1 Licensing e la crisi della Data Provenance . . . 14
  - 2.4.2 Privacy, GDPR e anonimizzazione . . . 14

**3 Analisi delle tecnologie** 17
- 3.1 Blender . . . 17
  - 3.1.1 Cycles . . . 19
  - 3.1.2 EEVEE . . . 19
- 3.2 The City Generator . . . 20
  - 3.2.1 Generazione procedurale via Geometry Nodes . . . 20
  - 3.2.2 Simulazione del traffico e coerenza semantica . . . 21
  - 3.2.3 Interoperabilità ed esportazione . . . 21
- 3.3 BlenderProc: Framework per la generazione di dati sintetici . . . 22
  - 3.3.1 Architettura modulare . . . 23

3.3.2 Generazione automatica delle annotazioni (Ground Truth) . 23
3.4 Unreal Engine 5 . . . . . 24
3.4.1 Deferred Rendering . . . . . 24
3.4.2 Path Tracing . . . . . 25
3.4.3 Movie Render Queue (MRQ) . . . . . 26
3.5 Ultralytics YOLO26 . . . . . 27
3.5.1 Innovazioni architetturali e principi di design . . . . . 28
3.5.2 Ottimizzazioni specifiche per i task di visione . . . . . 28

**4 Sviluppo** 31
4.1 Setup sperimentale e hardware . . . . . 31
4.2 Creazione della scena in Blender . . . . . 32
4.2.1 Modellazione procedurale della città . . . . . 32
4.2.2 Estensione e ottimizzazione del parco veicoli . . . . . 36
4.2.3 Configurazione dell'illuminazione ambientale . . . . . 38
4.2.4 Simulazione del traffico . . . . . 40
4.2.5 Esportazione della scena verso Unreal Engine . . . . . 42
4.3 Replica della scena in Unreal Engine 5 . . . . . 44
4.3.1 Importazione degli asset . . . . . 44
4.3.2 Mantenimento della coerenza visiva . . . . . 45
4.3.3 Project settings e gestione della skin cache . . . . . 46
4.4 Pipeline di rendering Cycles - BlenderProc . . . . . 47
4.4.1 Caricamento scena e assegnazione semantica . . . . . 47
4.4.2 Applicazione delle mappe HDRI . . . . . 48
4.4.3 Posizionamento stocastico della camera . . . . . 48
4.4.4 Rendering e generazione output . . . . . 51
4.4.5 Generazione delle annotazioni Bounding Box . . . . . 53
4.4.6 Generazione delle annotazioni per la segmentazione di istanza 54
4.4.7 Analisi delle performance computazionali (Cycles) . . . . . 55
4.5 Pipeline di rendering EEVEE . . . . . 57
4.5.1 Allineamento dinamico e lettura pose della camera . . . . . 57
4.5.2 Configurazioni ottimali e gestione delle trasparenze . . . . . 57
4.5.3 Rendering headless e registrazione dei tempi . . . . . 58
4.5.4 Analisi delle performance computazionali (EEVEE) . . . . . 58
4.6 Pipeline di rendering Unreal Engine 5 . . . . . 60
4.6.1 Sviluppo del sistema di importazione pose . . . . . 60
4.6.2 Configurazione del Sequencer e ottimizzazione delle animazioni 61
4.6.3 Configurazione della Movie Render Queue . . . . . 61
4.6.4 Automazione del rendering e registrazione dei tempi . . . . . 62
4.6.5 Analisi delle performance computazionali (Unreal Path Tracer) 62
4.6.6 Analisi delle performance computazionali (Unreal Deferred) 64

4.7 Confronto qualitativo tra i motori di rendering . . . . . 65
4.8 Addestramento dei modelli YOLO26 . . . . . 68
4.8.1 Progettazione dei dataset per il confronto . . . . . 68
4.8.2 Addestramento . . . . . 68

**5 Analisi dei risultati** 71
5.1 Metriche di valutazione . . . . . 72
5.2 Analisi Sim-to-Real . . . . . 72
5.2.1 Analisi a parità di numero di immagini . . . . . 72
5.2.2 Analisi a parità di budget temporale . . . . . 74
5.2.3 Confronto Unreal Path Tracer: 64 vs 4096 sample . . . . . 76
5.2.4 Analisi delle metriche sul dataset massimale . . . . . 78
5.2.5 Precision vs Recall . . . . . 80
5.2.6 Confronto qualitativo dell'inferenza su dati reali . . . . . 80
5.2.7 Conclusioni sul Sim-to-Real e sul confronto tra motori di rendering . . . . . 82
5.3 Cross-Engine validation . . . . . 82
5.4 Ottimizzazione mediante Fine-Tuning . . . . . 84
5.4.1 Analisi dei risultati di detection . . . . . 86
5.4.2 Analisi dei risultati di segmentazione . . . . . 88
5.4.3 Discussione sul volume dei dati e salto prestazionale . . . . . 90
5.5 Conclusioni sul trade-off metodologico e sviluppi futuri . . . . . 90
5.5.1 Sviluppi futuri . . . . . 91

**Bibliografia** 93

# Elenco delle figure

2.1 Pipeline dell'architettura R-CNN. Il sistema estrae circa 2000 region proposals, le ridimensiona e le classifica tramite una CNN. Fonte: Girshick et al. [1]. . . . . 6
2.2 Architettura di Faster R-CNN. L'innovazione principale è la Region Proposal Network (RPN) che sostituisce la ricerca selettiva lenta, permettendo alla rete di imparare dove guardare direttamente dalle feature map. Fonte: Ren et al. [7]. . . . . 7
2.3 Funzionamento di YOLO. L'immagine viene ridimensionata e processata da una singola rete convoluzionale che restituisce simultaneamente le coordinate dei box e le probabilità di classe. Fonte: Redmon et al. [8]. . . . . 8
2.4 Esempi dal dataset Microsoft COCO. Si noti la complessità delle scene, le occlusioni e la varietà degli oggetti annotati con maschere di segmentazione. Fonte: Lin et al. [2]. . . . . 9
2.5 Esempio di estrazione automatica delle annotazioni da un motore grafico (GTA V). A sinistra l'immagine RGB, a destra la Ground Truth di segmentazione semantica. Fonte: Richter et al. [9]. . . . . 10
2.6 Esempio di dati generati da Virtual KITTI 2. Il simulatore produce automaticamente dati multimodali perfettamente allineati: immagine RGB (in alto a sx), mappa di profondità (in alto a dx) e maschere di segmentazione semantica (in basso), permettendo di testare diverse condizioni ambientali in modo controllato. Fonte: Cabon et al. [5]. . . . . 11
2.7 Output multimodale generato tramite pipeline sintetica (BlenderProc). Oltre all'immagine RGB (in alto a sx), è possibile ottenere "gratuitamente" le normali alle superfici (in alto a dx), la mappa di profondità (in basso a sx) e la segmentazione semantica (in basso a dx). Fonte: Denninger et al. [6]. . . . . 13

3.1 Visualizzazione della viewport durante la fase di editing. Fonte: The City Generator - Super Hive Market. [19]. . . . . 22

3.2 Il pannello (a destra) con parametri di controllo. A sinistra, il confronto tra traiettorie procedurali (in alto) e il rendering finale (in basso). Fonte: The City Generator - Super Hive Market. [19]. . . . 22
3.3 Cinematic Viewport con visualizzazione in modalità Lit. Fonte: Rendering High Quality Frames with Movie Render Queue [23]. . . 25
3.4 "Virtual tour in Unreal Engine" by ARCHVYZ. Design by Toledano Architects. Fonte: Path Tracer[24]. . . . . 26
3.5 Confronto delle prestazioni tra YOLO26 e i principali modelli di object detection della famiglia YOLO e concorrenti. Si nota la superiorità di YOLO26 in termini di mAP COCO rispetto alla latenza (ms/img). Fonte: Ultralytics Documentation [4]. . . . . 27

4.1 Vista dall'alto del layout urbano. . . . . 33
4.2 Vista isometrica della città. . . . . 34
4.3 Vista dall'alto del layout urbano completo. . . . . 35
4.4 Vista isometrica della città completa. . . . . 36
4.5 Visualizzazione wireframe di una Nissan 350Z (a sinistra) e di un'Audi A7 (a destra) ottimizzate. . . . . 37
4.6 Confronto dei quattro scenari di illuminazione ambientale. . . . . 40
4.7 Visualizzazione dall'alto della rete di nodi e percorsi spline per la simulazione del traffico procedurale. . . . . 41
4.8 Visualizzazione prospettica della città in Unreal Engine. . . . . 45
4.9 Rappresentazione visiva dei bounding box generati automaticamente in BlenderProc, sovrapposti a render sintetici in Cycles. . . . . 54
4.10 Rappresentazione visiva della maschera di segmentazione generata automaticamente in BlenderProc rispetto al render Cycles. . . . . 55
4.11 Rappresentazione visiva di 4 render in Cycles. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto. . . 56
4.12 Rappresentazione visiva di 4 render in EEVEE. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto. . . 59
4.13 Rappresentazione visiva di 4 render in Unreal Path Tracing. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto. . . . . 64
4.14 Rappresentazione visiva di 4 render in Unreal Deferred. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto. . . . . 65

4.15 Confronto visivo della stessa istanza di simulazione renderizzata con i quattro motori della pipeline. . . . . 67

5.1 Andamento del mAP50-95 al variare del numero di immagini (500, 1.000, 5.000 e 10.000) per i task di Detection e Segmentation . . . . 73
5.2 Andamento del mAP50-95 al variare del tempo di rendering (1h, 2h, 5h, 12h e 24h) per i task di Detection e Segmentation . . . . . 75
5.3 Andamento del mAP50-95 al variare del numero di immagini per i task di Detection e Segmentation per Unreal Path Tracing confrontando il campionamento a 64 e 4096. Il grafico include tutti gli addestramenti effettuati (1h, 2h, 5h, 12h, 24h, 500, 1.000, 5.000, 10.000) . . . . . 77
5.4 Confronto delle metriche mAP50-95, mAP50, Precision e Recall per tutti i motori su dataset a 10000 immagini . . . . . 79
5.5 Confronto delle metriche di Precision e Recall per tutti i motori su dataset a 10000 immagini . . . . . 80
5.6 Matrice di confronto qualitativo: ogni riga rappresenta la medesima immagine reale processata dai cinque modelli addestrati sui diversi motori di rendering (10.000 immagini). La comparazione visiva permette di apprezzare la diversa accuratezza nelle maschere di segmentazione. . . . . 81
5.7 Matrici tra domini sintetici (mAP50-95). Le colonne rappresentano il motore di addestramento (*modello*), mentre le righe rappresentano il dataset di test. . . . . 83
5.8 Confronto della metrica mAP50-95 tra i due scenari di fine-tuning rispetto al baseline reale. La linea continua blu è relativa al fine tuning da 12.000 immagini per la detection, la linea continua gialla è relativa al fine tuning da 12.000 per la segmentazione; la linea tratteggiata blu indica il fine tuning da 1.000 immagini per la detection, mentre la linea tratteggiata gialla indica il fine tuning da 1.000 immagini per la segmentazione. Il punto di dispersione blu rappresenta il risultato dell'allenamento reale from scratch per la detection, mentre quello rosso indica quello per la segmentazione . . 86
5.9 Confronto dei risultati di Detection tra i due set di immagini reali utilizzati per il fine-tuning. Le linee tratteggiate indicano il riferimento dell'addestramento sul dataset reale da 10.000 immagini from scratch . . . . . 87
5.10 Confronto dei risultati di Segmentation tra i due set di immagini reali utilizzati per il fine-tuning. Le linee tratteggiate indicano il riferimento dell'addestramento sul dataset reale da 10.000 immagini from scratch . . . . . 89

# Capitolo 1

# Introduzione

## 1.1 Oggetto e Motivazione del lavoro

L'evoluzione della Computer Vision negli ultimi dieci anni è stata guidata da un cambio di paradigma. Si osserva, nel lavoro di **Girshick et al.** [1], il passaggio dall'ingegnerizzazione manuale delle feature all'apprendimento automatico delle stesse tramite Deep Learning. Tuttavia, questo processo ha portato alla luce una nuova criticità: la massiva dipendenza da dataset vasti, eterogenei e accuratamente etichettati.

L'analisi dei benchmark standard di riferimento, come Microsoft COCO [2] o Cityscapes [3], evidenzia infatti limiti strutturali difficilmente valicabili nel dominio reale. In primo luogo, l'elevata onerosità delle annotazioni manuali: la creazione di maschere di segmentazione *pixel-perfect* richiede molte ore di lavoro umano specializzato, rendendo la scalabilità dei dataset estremamente costosa. In secondo luogo, i dataset fotografici soffrono di *bias* di selezione intrinseci; le immagini tendono a rappresentare condizioni meteorologiche favorevoli e buona illuminazione, rendendo difficile per le reti neurali generalizzare su scenari critici o rari, i cosiddetti *corner case* (es. nebbia fitta, incidenti stradali).

In questo scenario, la generazione di dati sintetici tramite motori grafici emerge come una soluzione strategica. Questo approccio permette di creare ambienti virtuali controllati in cui ogni singolo pixel può essere etichettato automaticamente e con precisione assoluta (*Ground Truth*). Ciò non solo azzera i costi di annotazione, ma permette di generare variazioni infinite di una scena, garantendo quella varietà statistica necessaria per addestrare modelli robusti e sicuri.

## 1.2 Obiettivi e Scopo della ricerca

Il presente lavoro di tesi si propone di progettare, implementare e validare una pipeline multi-engine per la generazione di dataset sintetici, analizzando in modo sistematico l'impatto della qualità del rendering sulle capacità predittive dei modelli di visione artificiale. Lo scopo della ricerca risiede nell'esplorare il compromesso tra fedeltà fisica del rendering e capacità di generazione, nel contesto specifico della detection e segmentazione di veicoli in un ambiente urbano sintetico.

Nello specifico, la ricerca mira a risolvere le seguenti problematiche:

- **Analisi del fotorealismo:** Valutare se l'adozione di tecniche di *Path Tracing* (fisicamente corrette ma lente) offra vantaggi tangibili rispetto alla *Rasterizzazione* (veloce ma approssimata) nell'addestramento di un modello moderno come **YOLO26** (You Only Look Once)[4].
- **Ottimizzazione del campionamento:** Determinare se un basso numero di campioni nel calcolo della luce sia sufficiente per la generalizzazione della rete, riducendo drasticamente i tempi di produzione del dato sintetico.
- **Validazione Sim-to-Real:** Verificare la capacità di trasferimento dei modelli addestrati esclusivamente in ambiente virtuale su immagini appartenenti al dominio reale, misurando il *Domain Gap* risultante dalle diverse tecnologie di rendering.

## 1.3 Relazione con altri lavori

La ricerca si inserisce nel consolidato filone del paradigma *Sim-to-Real*. Si richiama la letteratura fondamentale riguardante l'uso di simulatori per la guida autonoma, come **Virtual KITTI** [5], e l'impiego di framework procedurali per l'automazione della Ground Truth come **BlenderProc** [6].

Rispetto ai lavori esistenti, che spesso si focalizzano sull'utilizzo di un singolo motore grafico, questa tesi propone un approccio comparativo *cross-engine*. Viene sviluppata una pipeline che permette di replicare la medesima istanza simulativa (stessa camera, stessa dinamica del traffico, stessa illuminazione HDRI) attraverso ecosistemi differenti: l'ambiente open-source di **Blender** e quello industriale di **Unreal Engine 5**. Tale confronto permette di isolare la variabile "rendering" da quella "contenuto", offrendo una valutazione inedita sull'efficienza computazionale dei diversi algoritmi di generazione.

## 1.4 Metodo seguito e Struttura dell'elaborato

Il metodo sperimentale seguito si articola in quattro fasi sequenziali. Inizialmente, è stata progettata una scena urbana procedurale e dinamica. Successivamente, è stata sviluppata una pipeline di sincronizzazione per garantire che le inquadrature fossero perfettamente sovrapponibili tra i vari motori di rendering testati. Nella terza fase, sono stati generati dataset multimodali (RGB ed etichette) per un totale di 10.000 campioni per ciascuna configurazione. Infine, i modelli YOLO26m sono stati addestrati da zero (*from scratch*) per confrontarne le performance metriche. Inoltre è stato svolto un lavoro di *Fine tuning* con i modelli pre-addestrati sul sintetico su un dataset reale.

L'elaborato è organizzato come segue:

- Il **Capitolo 2** esamina lo stato dell'arte dell'Object Detection, i limiti dei benchmark reali e le tecniche per colmare il *Domain Gap*.
- Il **Capitolo 3** analizza le architetture software utilizzate, approfondendo il funzionamento di Blender e i suoi motori grafici Cycles ed EEVEE, l'estensione The City Generator, il framework BlenderProc, Unreal Engine 5 e YOLO26.
- Il **Capitolo 4** descrive lo sviluppo pratico della pipeline, dalla creazione della scena urbana in Blender, le sfide dell'interoperabilità tra i software, cercando di garantire la coerenza visiva della scena tra Blender e Unreal Engine e la logica di automazione delle annotazioni.
- Il **Capitolo 5** presenta i risultati sperimentali, discutendo i tempi di calcolo, le curve di addestramento e l'efficacia del test su dati reali.

# Capitolo 2

# Stato dell'arte

In questo capitolo verrà analizzato lo stato dell'arte delle tecnologie di Object Detection, evidenziando i limiti attuali dei benchmark reali. Successivamente, verrà discusso l'emergere della generazione di dati sintetici come soluzione al problema della scarsità di dati, con un'attenzione particolare sul dibattito scientifico riguardante il trade-off tra fotorealismo e velocità di generazione, tema centrale di questo lavoro di tesi.

## 2.1 Algoritmi di Object Detection e limiti dei dataset reali

Il compito dell'Object Detection estende la semplice classificazione delle immagini richiedendo la localizzazione spaziale precisa delle istanze tramite bounding box. Lo stato dell'arte in questo dominio si ramifica in due macro-categorie: i detector a due stadi (Two-Stage)[1, 7], che privilegiano l'accuratezza, e i detector a stadio singolo (One-Stage)[8], ottimizzati per la velocità di inferenza real-time. Tuttavia, le performance di queste architetture sono intrinsecamente vincolate alla qualità e alla vastità dei dati di addestramento. Pertanto, in questa sezione verranno analizzati non solo gli algoritmi, ma anche i benchmark reali di riferimento (in particolare Microsoft COCO)[2], evidenziandone le limitazioni intrinseche — quali gli elevati costi di annotazione, il bias di selezione e l'assenza di scenari critici (corner cases) — che motivano la necessità di integrare dataset sintetici procedurali.

### 2.1.1 L'approccio Two-Stage: R-CNN e la sua evoluzione

Il paradigma dei detector a due stadi trova il suo fondamento nell'articolo di Girshick et al.[1],Rich feature hierarchies for accurate object detection and semantic segmentation. Questo metodo ha segnato lo storico ingresso delle Reti Neurali

Convoluzionali nel compito di object detection, superando i precedenti approcci basati su feature manuali. L'architettura proposta, illustrata in Figura 2.1, opera attraverso una pipeline sequenziale articolata in tre fasi distinte:

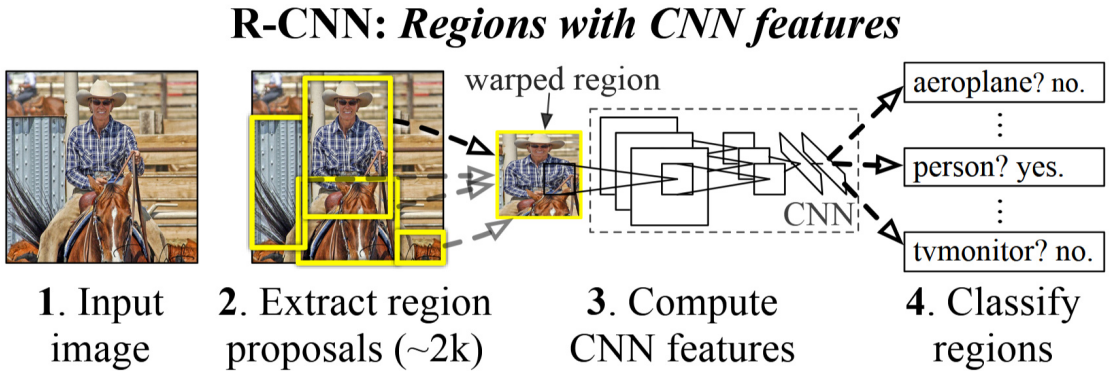



**Figura 2.1:** Pipeline dell'architettura R-CNN. Il sistema estrae circa 2000 region proposals, le ridimensiona e le classifica tramite una CNN. Fonte: Girshick et al. [1].

- **Region Proposals:** A differenza di un approccio a finestra scorrevole (*sliding window*), l'algoritmo genera circa 2000 regioni di interesse (RoI) per immagine utilizzando un algoritmo di *Selective Search*. Queste regioni, indipendenti dalla categoria, fungono da candidati per la localizzazione degli oggetti.
- **Feature Extraction:** Ogni regione candidata viene ridimensionata a una dimensione fissa e processata singolarmente attraverso una CNN profonda (es. AlexNet o VGG) per estrarre un vettore di feature ad alta dimensionalità.
- **Classification:** I vettori di feature estratti alimentano una serie di classificatori lineari, *Support Vector Machines* (SVM), addestrati specificamente per discriminare tra le varie classi di oggetti e affinare le coordinate del bounding box.

Nonostante R-CNN abbia migliorato drasticamente le metriche di accuratezza (mAP) rispetto allo stato dell'arte precedente, l'architettura presentava notevoli colli di bottiglia computazionali. La necessità di eseguire la forward-pass della CNN per ciascuna delle 2000 regioni rendeva il sistema estremamente lento e inadatto ad applicazioni real-time.

Questa limitazione è stata superata dalle evoluzioni successive, culminate in **Faster R-CNN** [7]. Quest'ultimo ha introdotto la *Region Proposal Network* (RPN), una rete neurale che condivide le feature con il detector, eliminando il costo della Selective Search e rendendo l'intera pipeline addestrabile *end-to-end*.

Tuttavia, un concetto fondamentale introdotto da R-CNN rimane tuttora una pietra miliare, specialmente nel contesto dei dataset sintetici: il paradigma del **Transfer Learning**. Girshick ha dimostrato che una rete pre-addestrata su dataset generici vasti (come ImageNet) e successivamente affinata (*fine-tuned*) su dati specifici ottiene performance superiori, suggerendo che le feature di basso livello sono trasferibili tra domini diversi (es. dal sintetico al reale).

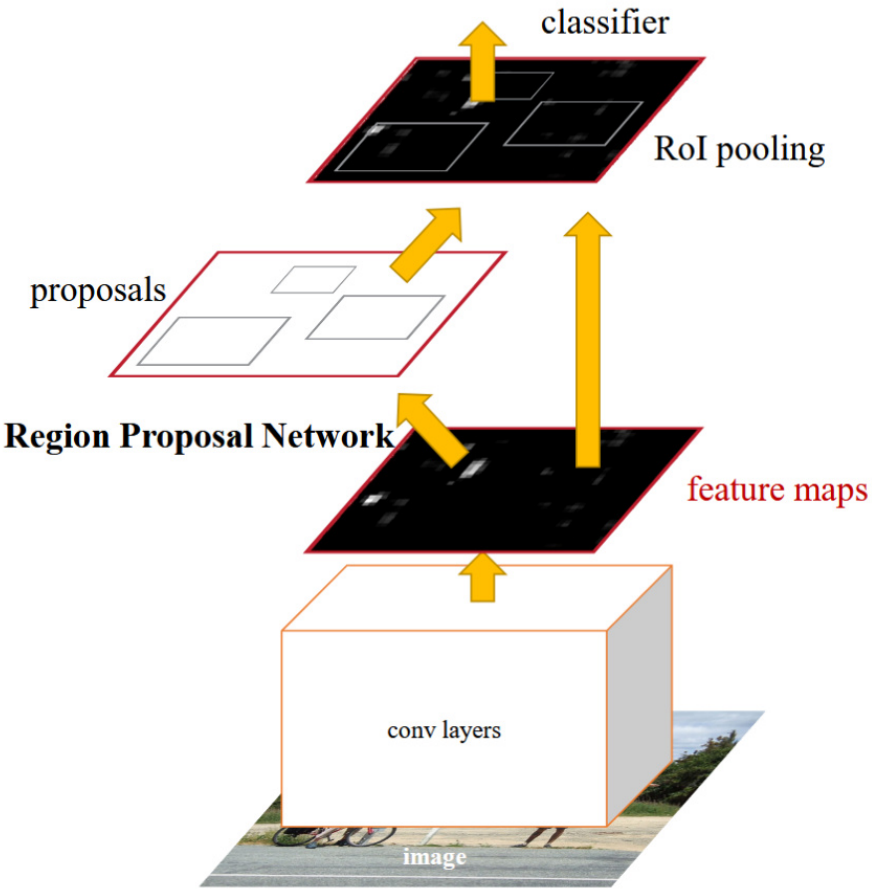



**Figura 2.2:** Architettura di Faster R-CNN. L'innovazione principale è la Region Proposal Network (RPN) che sostituisce la ricerca selettiva lenta, permettendo alla rete di imparare dove guardare direttamente dalle feature map. Fonte: Ren et al. [7].

### 2.1.2 L'approccio One-Stage: YOLO (You Only Look Once)

Per applicazioni critiche come la guida autonoma o il monitoraggio del traffico urbano, la latenza introdotta dai metodi Two-Stage (come R-CNN) risulta spesso inaccettabile. **Redmon et al.** [8] hanno rivoluzionato il campo con **YOLO**,

proponendo un cambio di paradigma radicale: il rilevamento degli oggetti viene inquadrato non più come una classificazione su regioni, ma come un singolo problema di *regressione*, mappando direttamente i pixel dell'immagine alle coordinate dei bounding box e alle probabilità di classe.

L'architettura di YOLO divide l'immagine in una griglia $S \times S$ (vedi Figura 2.3). Se il centro di un oggetto cade in una cella della griglia, quella specifica cella diventa responsabile della sua rilevazione. Ogni cella predice $B$ bounding box e i relativi punteggi di confidenza.

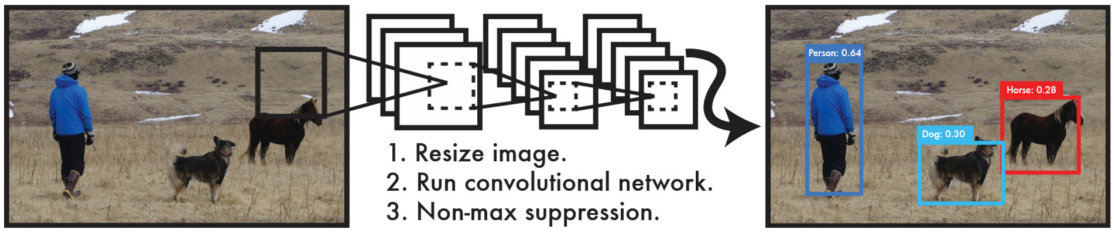



**Figura 2.3:** Funzionamento di YOLO. L'immagine viene ridimensionata e processata da una singola rete convoluzionale che restituisce simultaneamente le coordinate dei box e le probabilità di classe. Fonte: Redmon et al. [8].

La *Loss Function* utilizzata ottimizza simultaneamente la localizzazione (coordinate $x, y, w, h$) e la classificazione.

I vantaggi di questo approccio unificato sono molteplici:

- **Velocità:** YOLOv1 raggiungeva già 45 frame per secondo (fps) su GPU dell'epoca, rendendolo idoneo per l'analisi video *real-time*.
- **Contesto Globale:** A differenza delle tecniche a finestra scorrevole o basate su regioni, YOLO "vede" l'intera immagine durante l'addestramento. Questo permette alla rete di codificare informazioni contestuali, riducendo drasticamente gli errori di falsi positivi sullo sfondo rispetto a modelli come Fast R-CNN.

### 2.1.3 Il Benchmark standard: Microsoft COCO

Per valutare e comparare oggettivamente le performance di questi algoritmi, la comunità scientifica si affida a benchmark standardizzati. Attualmente, lo standard *de facto* è rappresentato da **Microsoft COCO** (Common Objects in Context) [2]. Introdotto da Lin et al., COCO si distingue dai predecessori (come PASCAL VOC) per la complessità semantica e la scala: contiene oltre 300.000 immagini e 2.5 milioni di istanze etichettate, ponendo l'enfasi su due aspetti critici:

- **Viste non iconiche:** Il dataset privilegia immagini dove gli oggetti sono parzialmente occlusi, in ombra o ripresi da angolazioni insolite, riflettendo la complessità del mondo reale rispetto alle classiche foto "da studio".
- **Contesto:** Gli oggetti sono immersi nel loro ambiente naturale e circondati da disordine visivo, costringendo la rete a utilizzare il contesto spaziale per l'inferenza.

**Figura 2.4:** Esempi dal dataset Microsoft COCO. Si noti la complessità delle scene, le occlusioni e la varietà degli oggetti annotati con maschere di segmentazione. Fonte: Lin et al. [2].

Tuttavia, l'analisi di COCO evidenzia implicitamente il limite strutturale dei dataset reali: l'elevata onerosità della creazione di annotazioni. Ottenere maschere di segmentazione *pixel-perfect* richiede uno sforzo umano enorme; è stato stimato che l'annotazione accurata di una singola immagine complessa possa richiedere più di 90 minuti di lavoro manuale. [3]

Inoltre, i dataset fotografici soffrono di un *bias* di selezione intrinseco: le immagini tendono a rappresentare condizioni meteorologiche favorevoli e buona illuminazione.

Questo rende difficile per le reti generalizzare su casi limite, come nebbia fitta, neve intensa o scenari di incidenti stradali. Tali situazioni, pur essendo rare nei dataset reali, sono critiche per la sicurezza e rappresentano proprio gli scenari che possono essere simulati digitalmente con costi marginali.

## 2.2 Soluzione dei dataset sintetici: L'approccio Sim-to-Real

A fronte delle limitazioni intrinseche dei dataset reali - in primis la scarsità di annotazioni "pixel-perfect" e la difficoltà nel reperire casi rari - la comunità scientifica ha rivolto l'attenzione verso la generazione di dati sintetici. Questo paradigma, spesso definito *Sim-to-Real*, si basa sull'utilizzo di motori di gioco o renderizzatori fisici per simulare ambienti virtuali dai quali estrarre automaticamente sia le immagini che le relative annotazioni (Ground Truth).

Il vantaggio principale di questo approccio risiede nell'azzeramento del costo di annotazione umana: poiché il motore grafico genera la scena, esso possiede una conoscenza perfetta della geometria, della classe e della posizione di ogni oggetto visibile.

### 2.2.1 Dai videogiochi ai simulatori dedicati

Uno dei lavori pionieristici in questo ambito è *Playing for Data* di **Richter et al.** [9]. Gli autori hanno intuito che i moderni videogiochi open-world, come *Grand Theft Auto V* (GTA V), offrivano già un livello di realismo visivo sufficiente per addestrare reti neurali. Intercettando le chiamate alle API grafiche tra il gioco e la GPU, sono riusciti a estrarre maschere di segmentazione semantica precise senza accedere al codice sorgente del gioco.

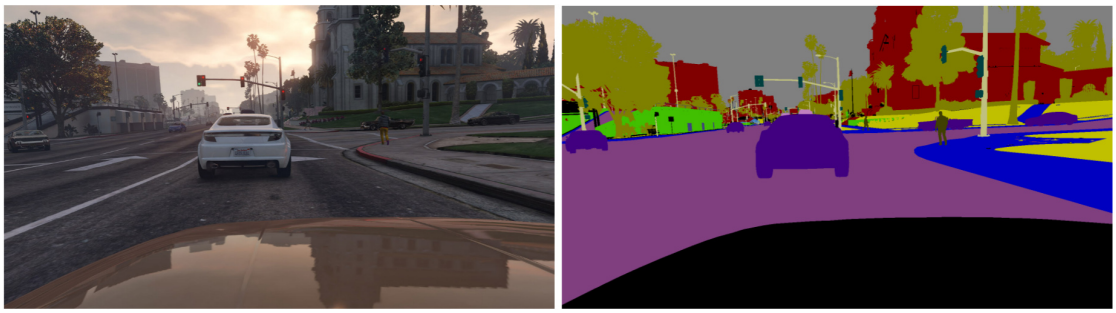


**Figura 2.5:** Esempio di estrazione automatica delle annotazioni da un motore grafico (GTA V). A sinistra l'immagine RGB, a destra la Ground Truth di segmentazione semantica. Fonte: Richter et al. [9].

Questo metodo ha dimostrato che i dati sintetici, seppur provenienti da motori di rasterizzazione pensati per l'intrattenimento, potevano arricchire i dataset reali migliorando le performance di generalizzazione.

Tuttavia, l'uso di videogiochi commerciali presenta limiti di flessibilità e licenza. Per superare questi ostacoli, la ricerca si è spostata verso simulatori proprietari o framework dedicati. Un esempio significativo è **Virtual KITTI 2** [5]. Cabon et al. hanno proposto un clone virtuale del celebre dataset KITTI, come illustrato in Figura 2.6, offrendo però ciò che il dato reale non può dare: il controllo totale sui parametri ambientali. Con Virtual KITTI 2 è possibile modificare a piacimento le condizioni meteorologiche (pioggia, nebbia), l'illuminazione e la posizione della camera, permettendo di studiare l'impatto di singole variabili sulle performance degli algoritmi di Object Detection, una pratica impossibile con i soli dati reali.

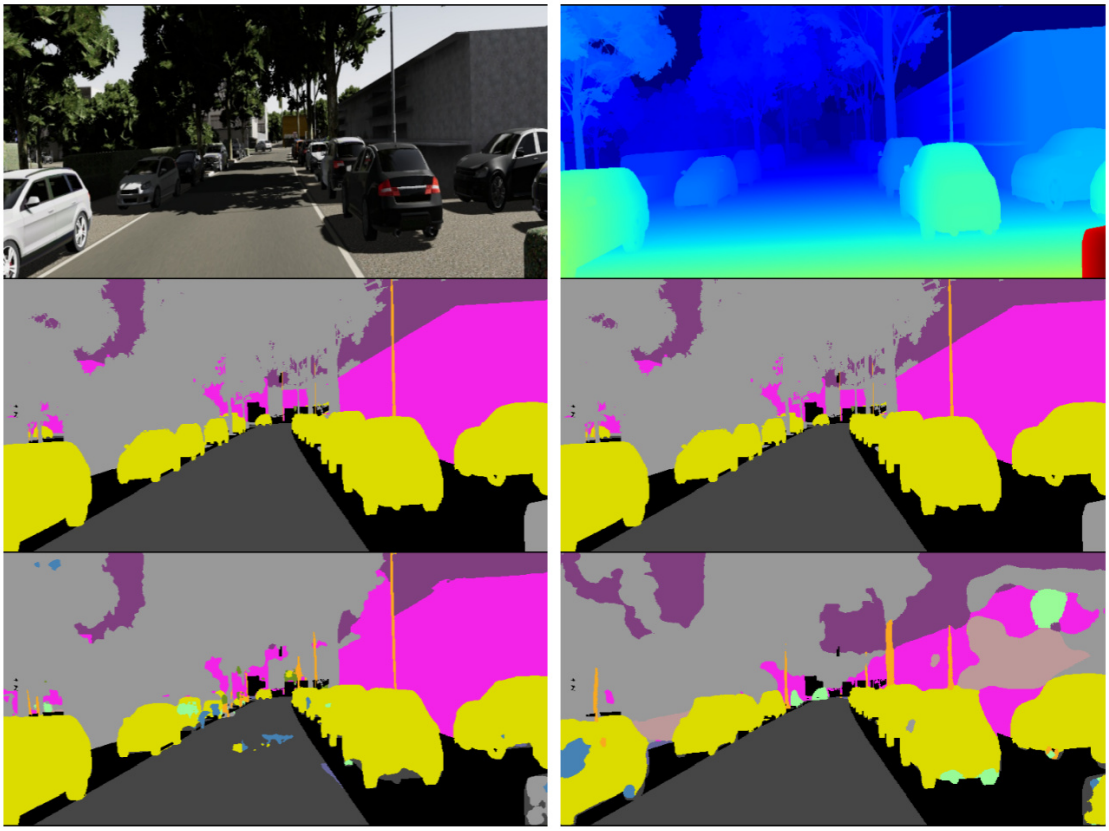


**Figura 2.6:** Esempio di dati generati da Virtual KITTI 2. Il simulatore produce automaticamente dati multimodali perfettamente allineati: immagine RGB (in alto a sx), mappa di profondità (in alto a dx) e maschere di segmentazione semantica (in basso), permettendo di testare diverse condizioni ambientali in modo controllato. Fonte: Cabon et al. [5].

## 2.3 Colmare il gap tra sintetico e reale

Nonostante i vantaggi in termini di scalabilità e ricchezza delle annotazioni, l'utilizzo esclusivo di dati sintetici introduce una nuova criticità: il *Domain Gap* (o *Domain Shift*). Le reti neurali sono estremamente sensibili alle discrepanze statistiche tra il dominio di addestramento (sintetico) e quello di test (reale). Tali discrepanze si manifestano a diversi livelli:

- **Gap di apparenza:** Differenze nelle texture, nell'illuminazione e nella resa dei materiali, spesso causate dai limiti degli algoritmi di rendering (es. approssimazioni nella rasterizzazione).
- **Gap di contenuto:** Differenze nella disposizione degli oggetti, nella varietà dei modelli 3D e nella fisica della scena rispetto al mondo reale.

Se non mitigato, questo divario porta a un drastico calo delle prestazioni quando un modello addestrato sinteticamente viene testato su immagini reali.

### 2.3.1 Realismo visivo e Domain Randomization

Per ridurre il domain gap, la letteratura propone due strategie principali, spesso complementari: il miglioramento del fotorealismo e la *Domain Randomization*.

Il lavoro di **Wrenninge et al.** con il dataset **Synscapes** [10] dimostra l'importanza della fedeltà fisica. Utilizzando tecniche di rendering basate sul *Path Tracing* (invece della più semplice rasterizzazione) e materiali definiti fisicamente (Physically Based Rendering - PBR), hanno creato un dataset che si avvicina alla qualità fotorealistica. Lo studio suggerisce che all'aumentare del realismo visivo diminuisce la distanza distributiva tra le feature apprese nel dominio sintetico e quelle necessarie per il dominio reale.

Parallelamente, un approccio volto alla generazione procedurale scalabile è presentato in **BlenderProc** [6]. Denninger et al. hanno sviluppato una pipeline modulare basata su Blender che combina il realismo del motore di rendering *Cycles* (ray-tracing) con la flessibilità procedurale. BlenderProc facilita l'implementazione della *Domain Randomization*, una tecnica che varia casualmente texture, luci, posizioni e angolazioni della camera in modo estremo. L'intuizione alla base è che, esponendo la rete a una variabilità sintetica sufficientemente ampia, il dominio reale venga percepito dalla rete stessa come una semplice variazione del dominio di training, costringendola a imparare le forme essenziali degli oggetti piuttosto che a memorizzare texture specifiche.

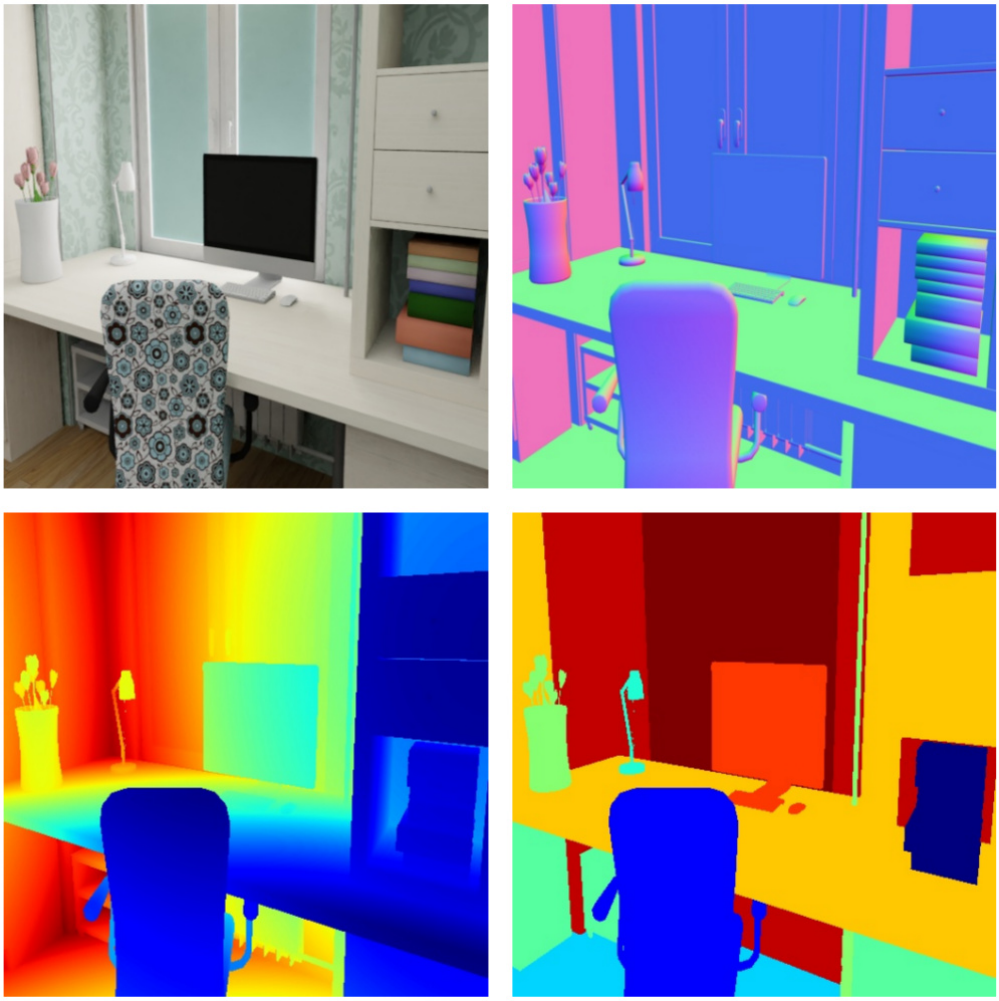


**Figura 2.7:** Output multimodale generato tramite pipeline sintetica (BlenderProc). Oltre all'immagine RGB (in alto a sx), è possibile ottenere "gratuitamente" le normali alle superfici (in alto a dx), la mappa di profondità (in basso a sx) e la segmentazione semantica (in basso a dx). Fonte: Denninger et al. [6].

Questa dicotomia tra la ricerca del fotorealismo assoluto (tipica del path tracing offline) e la necessità di generare grandi moli di dati in tempi rapidi (tipica dei motori real-time) costituisce il nucleo del confronto tecnologico che verrà affrontato nei capitoli successivi di questa tesi.

## 2.4 Aspetti etici e legali

L'adozione di dataset sintetici non è motivata esclusivamente da esigenze tecniche o prestazionali, ma risponde anche a pressanti necessità di conformità normativa che stanno investendo il settore dell'Intelligenza Artificiale. La transizione verso dati generati artificialmente rappresenta una strategia proattiva per mitigare rischi legali legati al copyright e per garantire il rispetto della privacy.

### 2.4.1 Licensing e la crisi della Data Provenance

La pratica comune di addestrare modelli di Deep Learning su enormi dataset ottenuti tramite *web scraping* indiscriminato (es. LAION, ImageNet) è oggi oggetto di forte scrutinio. Un recente audit su larga scala condotto dalla *Data Provenance Initiative*, pubblicato da **Longpre et al.** [11], ha evidenziato una crisi sistemica di trasparenza nella filiera dei dati.

Lo studio rivela che molti dataset open source, ampiamente utilizzati sia in ambito accademico che industriale, contengono materiale protetto da diritto d'autore prelevato senza un esplicito consenso. L'analisi ha mostrato che oltre il 70% dei dataset esaminati presenta licenze "non specificate", errate o incompatibili con l'uso commerciale, esponendo ricercatori e aziende a rischi legali significativi e potenziali cause per violazione di copyright.

In questo scenario, la generazione sintetica procedurale offre una soluzione strutturale, garantendo una *Data Provenance* pulita. A differenza dello *scraping*, dove l'origine del dato è incerta, in un ambiente sintetico ogni pixel generato è il risultato del rendering di asset 3D (modelli, texture) di cui l'utente possiede legittimamente la licenza (sia essa Creative Commons CC0 o una licenza commerciale acquistata). Questo approccio sposta la responsabilità legale dal controllo di milioni di immagini scaricate dal web alla gestione di un numero finito di asset digitali controllati, assicurando la piena titolarità dei dataset prodotti.

### 2.4.2 Privacy, GDPR e anonimizzazione

Nel contesto specifico della guida autonoma e della sorveglianza urbana, la raccolta di dati reali comporta inevitabilmente l'acquisizione di informazioni personali sensibili, quali i volti dei pedoni e le targhe dei veicoli. In Europa, il Regolamento Generale sulla Protezione dei Dati (GDPR) [12] impone restrizioni rigorose sul trattamento di tali dati, richiedendo il consenso esplicito degli interessati o la loro anonimizzazione irreversibile.

Le tecniche tradizionali di anonimizzazione, come la sfocatura o la pixelatura dei volti, introducono tuttavia un degrado informativo che può compromettere l'efficacia dell'addestramento, come dimostrato dallo studio di Hukkelås et al. [13]

Ad esempio, secondo Rasouli et al. [14], oscurare il volto di un pedone impedisce alla rete neurale di inferire la direzione dello sguardo (*gaze estimation*), un indizio fondamentale per prevedere se la persona sta per attraversare la strada.

Avvalorato da quanto scritto da Nikolenko in **Synthetic Data for Deep Learning** [15], i dataset sintetici risolvono questo trade-off implementando il principio della *Privacy by Design*. I pedoni inseriti nelle scene virtuali sono avatar digitali privi di corrispondenza con persone reali, e le targhe dei veicoli sono generate proceduralmente con stringhe alfanumeriche casuali. Ciò elimina alla radice qualsiasi rischio di violazione della privacy, rendendo i dati intrinsecamente conformi alle normative vigenti senza la necessità di post-elaborazioni distruttive per la qualità del segnale visivo.

# Capitolo 3

# Analisi delle tecnologie

La realizzazione di una pipeline per la generazione di dati sintetici richiede l'integrazione di strumenti eterogenei che spaziano dalla modellazione 3D procedurale al rendering fotorealistico. In questo capitolo verranno analizzate le tecnologie software selezionate per questo lavoro di tesi.

La scelta di mettere a confronto **Blender** e **Unreal Engine 5** non è casuale, ma riflette la dicotomia attuale nella computer grafica tra *Offline Rendering* e *Real-Time Rendering*. Mentre il primo rappresenta storicamente lo standard per la qualità visiva assoluta a discapito dei tempi di calcolo, il secondo sta rapidamente colmando il divario qualitativo grazie a nuove tecnologie di illuminazione globale, mantenendo velocità di generazione superiori. Verrà inoltre descritto l'addon **The City Generator** utilizzato per la generazione procedurale della città e **BlenderProc**, il framework necessario per la pipeline automatica della generazione dei dataset e delle annotazioni.

## 3.1 Blender

Blender è una suite di creazione 3D open-source che integra l'intera pipeline di produzione grafica, che include modellazione, rendering, animazione, shading, video editing, VFX, compositing e simulazioni fisiche, tutto in un unico ambiente coerente [16]. A differenza di strumenti frammentati che richiedono software specifici per ogni fase (modellazione, texturing, rendering), Blender centralizza queste operazioni su un unico *Dependency Graph*, accessibile interamente tramite API Python.

Questa trasparenza architetturale, unita alla licenza open-source, rende Blender uno strumento privilegiato per la ricerca, garantendo quella riproducibilità e quel controllo granulare sui dati necessari per l'addestramento di reti neurali, come confermato dall'adozione in framework quali BlenderProc [6]. Prima di confrontare le prestazioni dei motori di rendering, è opportuno delineare l'infrastruttura tecnica

del software, analizzando come i diversi moduli - dalla gestione della memoria alla simulazione fisica - concorrano alla creazione di un ambiente virtuale coerente.

**Architettura (Data-Blocks)** Il core di Blender gestisce la memoria tramite un sistema di **Data-Blocks**. Ogni entità (Mesh, Materiale, Scena) è un blocco indipendente che può essere istanziato e condiviso [16]. Questo permette di gestire scene complesse tramite il meccanismo di *Instancing*: migliaia di oggetti possono condividere la stessa geometria in memoria, variando solo le trasformazioni spaziali, ottimizzando drasticamente l'uso della VRAM.

**Modellazione e Sculpting** Blender offre diversi paradigmi per la definizione della geometria:

- **Poligonale:** Manipolazione diretta di vertici, spigoli e facce (supporto a N-gons) per definire superfici *Hard Surface*.
- **Sculpting:** Manipolazione di mesh ad alta densità tramite pennelli digitali per definire forme organiche o dettagli superficiali complessi.
- **Modificatori:** Uno stack di operazioni non distruttive (es. Subdivision Surface, Boolean, Decimate, Remesh) che processano la geometria in tempo reale senza alterare i dati originali.

**Geometry Nodes e proceduralità** Il modulo *Geometry Nodes* introduce un approccio parametrico basato su nodi. Invece di modellare manualmente, l'utente definisce un grafo di operazioni logico-matematiche che generano o trasformano la geometria. Questo sistema è essenziale per creare asset adattivi e distribuire elementi (scattering) su grandi superfici basandosi su regole topologiche o mappe di densità, permettendo variazioni infinite a partire da un set limitato di input.

**Materiali e Shading** La definizione delle proprietà ottiche delle superfici avviene tramite un sistema a nodi. Blender adotta lo standard PBR (Physically Based Rendering) attraverso lo shader **Principled BSDF**. Questo nodo unifica in un unico modello matematico parametri fisici come metallicità, rugosità, trasmissione e indice di rifrazione (IOR), garantendo la conservazione dell'energia e la coerenza fisica dei materiali.

**Animazione e simulazione** Il sistema di animazione gestisce l'interpolazione di qualsiasi proprietà nel tempo tramite F-Curves. Oltre al classico Keyframing e al Rigging scheletrico, l'animazione può essere guidata proceduralmente o tramite simulazioni fisiche integrate (Rigid Body per collisioni, Cloth per tessuti, Fluid

dynamics), permettendo di generare scenari dinamici complessi con interazioni fisicamente plausibili [16].

**Python scripting** L'integrazione di Python è pervasiva: ogni comando dell'interfaccia grafica corrisponde a una chiamata API (`bpy`). Ciò consente l'esecuzione in modalità *Headless* (senza GUI) su server di calcolo, facilitando il rendering batch e la manipolazione programmatica della scena per scopi di Data Augmentation.

### 3.1.1 Cycles

Cycles è il motore di rendering *production-ready* di Blender, basato sull'algoritmo di **Path Tracing** [17]. È progettato per simulare il comportamento fisico della luce senza compromessi qualitativi.

Cycles risolve l'*Equazione del Trasporto della Luce* (LTE) utilizzando metodi Monte Carlo [18]. Per ogni pixel del piano immagine, il motore lancia una serie di raggi primari (samples) nella scena. Quando un raggio interseca una superficie, viene calcolata la risposta del materiale (BSDF) e generati raggi secondari (di riflessione, rifrazione o ombra) in direzioni stocastiche. Il processo continua ricorsivamente fino a raggiungere una sorgente luminosa o il limite massimo di rimbalzi (bounces). Il colore finale del pixel è la media pesata dei contributi di tutti i percorsi campionati. Le caratteristiche tecniche principali che garantiscono il realismo includono:

- **Global Illumination (GI):** Calcola accuratamente l'illuminazione indiretta, inclusa la *Color Bleeding* (trasferimento di colore tra superfici vicine).
- **Strutture di Accelerazione:** Utilizza strutture BVH (Bounding Volume Hierarchy) per ottimizzare il calcolo delle intersezioni raggio-triangolo, sfruttando l'accelerazione hardware delle moderne GPU (es. RT Core di NVIDIA tramite API OptiX) [16].
- **Denoising:** Poiché il Path Tracing è intrinsecamente rumoroso a bassi campionamenti, Cycles integra denoiser basati su AI (come OpenImageDenoise) per ricostruire un'immagine pulita riducendo i tempi di rendering necessari.

### 3.1.2 EEVEE

EEVEE (Extra Easy Virtual Environment Engine) è un motore di rendering basato sulla **Rasterizzazione**, progettato per privilegiare la velocità e l'interattività, adottando tecniche simili ai moderni motori di gioco.

A differenza del Ray Tracing, EEVEE non simula il percorso dei fotoni. Il processo proietta direttamente la geometria 3D (triangoli) sul piano 2D dello schermo (Raster). Durante questo passaggio, le proprietà della superficie (colore,

profondità, normali) vengono scritte in buffer specializzati (*G-Buffer*). Successivamente, una serie di passaggi di post-elaborazione combina questi buffer per calcolare l'illuminazione finale.

Non potendo tracciare raggi fuori dal campo visivo, EEVEE utilizza diverse tecniche di approssimazione per simulare fenomeni complessi [16]:

- **Ombre:** Utilizza *Cascaded Shadow Maps*, ovvero mappe di profondità precalcolate dal punto di vista della luce.
- **Illuminazione Indiretta:** Simulata tramite *Irradiance Volumes* (griglie di sonde statiche che catturano la luce diffusa) e *Reflection Cubemaps*.
- **Screen Space Effects:** Riflessi (SSR) e occlusione ambientale (GTAO) sono calcolati utilizzando solo le informazioni visibili nel buffer di profondità corrente. Questo introduce limitazioni fisiche: se un oggetto che dovrebbe riflettersi è fuori dall'inquadratura, il riflesso non verrà renderizzato.

## 3.2 The City Generator

La generazione di dataset per la guida autonoma richiede scenari urbani vasti, diversificati e semanticamente coerenti. La modellazione manuale di intere città è un processo inefficace e non scalabile, che rischia inoltre di introdurre bias dovuti alla ripetitività delle scelte umane nella disposizione degli oggetti. Per superare questo limite, è stato integrato nella pipeline lo strumento **The City Generator** [19], un add-on per Blender basato interamente sul sistema dei *Geometry Nodes*.

Questo strumento permette di automatizzare la creazione di layout urbani complessi partendo da semplici primitive geometriche, garantendo al contempo il controllo granulare sui parametri di generazione.

### 3.2.1 Generazione procedurale via Geometry Nodes

A differenza di generatori basati su script distruttivi, questo add-on opera in modo non distruttivo sfruttando il grafo dei nodi di Blender. Il flusso di lavoro si articola in tre fasi principali [20]:

1. **Mesh di controllo:** L'utente definisce la pianta della città manipolando una semplice mesh piana. I bordi (edges) di questa mesh vengono interpretati dall'algoritmo come assi stradali, mentre le facce (faces) definiscono le aree edificabili (city blocks).
2. **Interpretazione parametrica:** Il modificatore *Geometry Nodes* legge gli attributi della mesh e istanzia proceduralmente gli asset. È possibile controllare

densità, altezza media degli edifici, varianza architettonica e tipologia di quartiere (es. residenziale vs commerciale) agendo su slider nell'interfaccia, senza dover rimodellare la geometria [21].

3. **Scattering degli asset:** Il sistema distribuisce automaticamente elementi di arredo urbano come lampioni, alberi e bidoni. La distribuzione non è puramente casuale ma segue regole logiche (es. i lampioni sono posizionati a intervalli regolari lungo i bordi strada, gli alberi nelle zone definite come parchi).

### 3.2.2 Simulazione del traffico e coerenza semantica

Un aspetto critico per l'addestramento di reti neurali è la presenza di veicoli dinamici che rispettino il codice della strada. L'add-on integra un modulo di simulazione del traffico che genera automaticamente le traiettorie dei veicoli:

- **Path Generation:** Le strade generate contengono curve (Splines) invisibili che fungono da binari per i veicoli. Questo assicura che le auto rimangano al centro della corsia e rispettino la curvatura degli incroci [21].
- **Logica di intersezione:** Il sistema gestisce semafori e precedenze di base, permettendo di popolare la città con traffico in movimento coerente, essenziale per evitare collisioni irreali che degraderebbero la qualità del dataset sintetico.

Infine, l'uso di materiali procedurali con supporto per shader di parallasse (per simulare gli interni delle finestre) permette di generare scene notturne realistiche, aumentando ulteriormente la varianza del dataset [19].

### 3.2.3 Interoperabilità ed esportazione

Un fattore determinante nella scelta di questo specifico add-on è la sua capacità di integrarsi in pipeline ibride. Come evidenziato nella documentazione tecnica e nei tutorial dimostrativi, lo strumento supporta l'esportazione ottimizzata dell'intera scena verso motori real-time esterni [21]. Questa funzionalità permette di trasferire la geometria procedurale, le istanze e i materiali direttamente in **Unreal Engine**, mantenendo la coerenza visiva e strutturale della città. Ciò abilita il confronto diretto tra i motori di rendering (oggetto di questa tesi) utilizzando la medesima base dati di partenza.

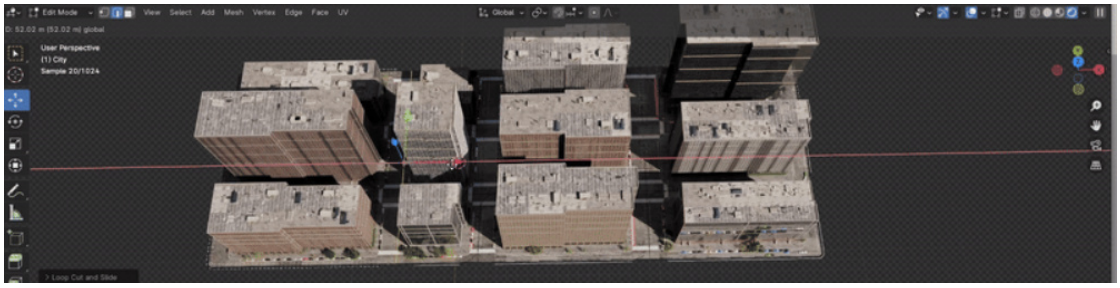


**Figura 3.1:** Visualizzazione della viewport durante la fase di editing. Fonte: The City Generator - Super Hive Market. [19].

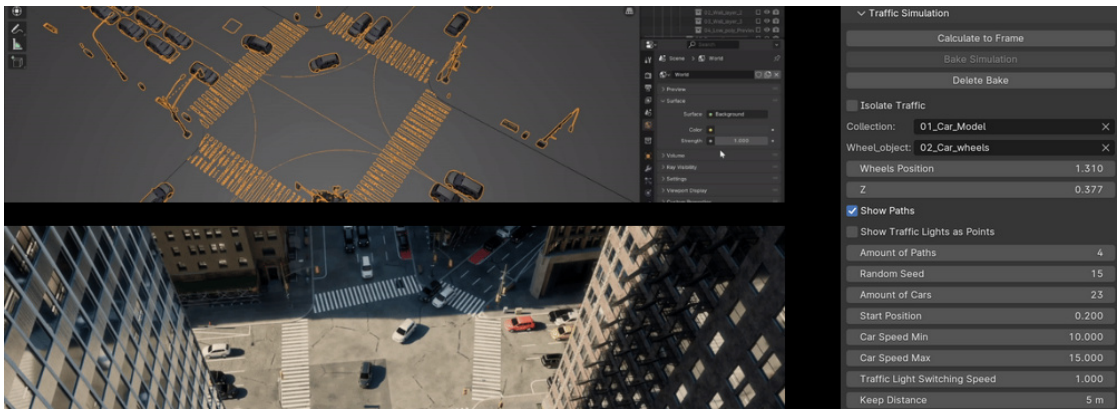



**Figura 3.2:** Il pannello (a destra) con parametri di controllo. A sinistra, il confronto tra traiettorie procedurali (in alto) e il rendering finale (in basso). Fonte: The City Generator - Super Hive Market. [19].

## 3.3 BlenderProc: Framework per la generazione di dati sintetici

La produzione di dataset sintetici su larga scala richiede un livello di automazione che trascende le funzionalità standard di modellazione 3D. A tale scopo, la letteratura scientifica ha converso verso soluzioni di generazione procedurale, tra le quali emerge **BlenderProc** [6, 22]. Si tratta di un framework open-source modulare, sviluppato in Python, progettato specificamente per generare dati fotorealistici annotati per l'addestramento di reti neurali.

Strutturalmente, BlenderProc non opera come un'applicazione standalone, bensì come un *wrapper* di alto livello costruito sopra le API native di Blender (`bpy`). La sua funzione primaria è l'astrazione della complessità: il framework maschera le chiamate di basso livello al motore grafico, permettendo all'utente di definire il

processo di generazione dati attraverso una pipeline configurabile e riproducibile, disaccoppiando la logica di simulazione dalla gestione manuale della scena 3D.

### 3.3.1 Architettura modulare

L'architettura del software si basa sull'esecuzione sequenziale di moduli funzionali indipendenti. Ogni modulo agisce come una funzione di trasformazione sullo stato globale della scena (il *Dependency Graph* di Blender). I principali blocchi logici includono:

- **Advanced loading e semantica:** I moduli *Loader* gestiscono l'importazione di asset eterogenei (file .obj, .ply, scene .blend, formati BOP)[22]. Una caratteristica distintiva è la capacità di assegnare automaticamente metadati semantici (Custom Properties) agli oggetti in fase di caricamento. Tramite filtri basati su espressioni regolari (RegEx) o gerarchie di nomi, il sistema assegna ID di classe e categorie semantiche a migliaia di istanze, predisponendo la scena per la segmentazione automatica senza intervento umano.
- **Camera sampling spaziale:** Per garantire la varianza statistica del dataset, BlenderProc sostituisce il posizionamento manuale con algoritmi di *Sampling* stocastico. È possibile definire volumi di interesse o vincoli geometrici entro cui campionare la posizione (Location) e l'orientamento (Rotation) della telecamera. Il sistema integra controlli di validità geometrica (*Constraint Checks*) per scartare automaticamente pose che generano collisioni con la geometria, occlusioni totali o inquadrature non informative (es. rivolte verso lo skybox).
- **Domain Randomization:** Per aumentare la capacità di generalizzazione dei modelli AI e mitigare l'overfitting, il framework offre moduli per la randomizzazione procedurale dei materiali (es. variazione casuale di rugosità, colore e texture PBR) e dell'illuminazione (es. campionamento di diverse mappe HDRI o variazione dell'intensità luminosa), simulando una vasta gamma di condizioni ambientali a partire da una singola geometria statica.

### 3.3.2 Generazione automatica delle annotazioni (Ground Truth)

Il core tecnologico di BlenderProc risiede nella capacità di sfruttare i buffer interni del motore di rendering per generare annotazioni *pixel-perfect* a costo zero, eliminando gli errori tipici dell'annotazione manuale. Attivando specifici passaggi di rendering (*Render Passes*), il sistema estrae dati ausiliari perfettamente allineati con l'immagine RGB:

- **Segmentazione semantica e di istanza:** Interrogando l'indice univoco degli oggetti o il loro materiale durante il processo di rasterizzazione o ray-tracing, il framework genera mappe dense in cui il valore di ogni pixel codifica la classe o l'identificativo univoco dell'oggetto visibile.
- **Bounding Box 2D/3D:** Il calcolo dei box non avviene tramite semplice proiezione geometrica dei vertici, ma include logiche di visibilità avanzate. Il sistema calcola la percentuale di superficie visibile per ogni istanza, permettendo di filtrare automaticamente oggetti pesantemente occlusi (es. $< 10\%$ di visibilità) o troncati ai bordi dell'immagine, che potrebbero introdurre rumore durante il training.
- **Dati ausiliari:** Oltre alle annotazioni standard, è possibile estrarre mappe di profondità metriche (Distance/Depth Maps) e mappe delle normali alle superfici (Surface Normals), essenziali per task di visione geometrica multimodale.

Infine, moduli di *Writing* dedicati si occupano di serializzare questi dati in formati standardizzati per la Computer Vision (come il formato COCO, Pascal VOC o HDF5)[22], rendendo l'output immediatamente ingeribile dalle moderne pipeline di Deep Learning.

## 3.4 Unreal Engine 5

Unreal Engine (UE), sviluppato da Epic Games, è una piattaforma di creazione 3D real-time che trascende il tradizionale utilizzo videoludico, trovando applicazione nella produzione virtuale, nell'automotive e nella simulazione scientifica. L'avvento della versione 5 ha introdotto cambiamenti architetturali significativi, come il sistema di geometria virtualizzata *Nanite* e l'illuminazione globale dinamica *Lumen*, progettati per gestire una complessità geometrica e luminosa senza precedenti senza la necessità di pesanti ottimizzazioni manuali (es. creazione di LODs o baking delle luci).

Tuttavia, nel contesto della generazione di dati sintetici controllati, l'aspetto più rilevante risiede nella flessibilità della pipeline di rendering, che permette di alternare tecniche di rasterizzazione real-time a soluzioni di ray tracing offline, e nella gestione avanzata dell'output tramite code di rendering.

### 3.4.1 Deferred Rendering

Il *Deferred Shading* è la pipeline di rendering predefinita di Unreal Engine, ottimizzata per le prestazioni real-time. Il termine "Differito" indica che il calcolo

dell'illuminazione viene posticipato rispetto alla rasterizzazione della geometria. Il processo avviene in due fasi:

1. **G-Buffer Pass:** La geometria viene proiettata sullo schermo e le proprietà dei materiali (Colore Base, Normali, Rugosità, Profondità, Specularità) vengono scritte in buffer immagine chiamati **G-Buffers**.

2. **Lighting Pass:** Il motore legge i dati pixel per pixel dai G-Buffers e calcola l'illuminazione finale.

Questo approccio disaccoppia la complessità geometrica da quella dell'illuminazione, permettendo di gestire centinaia di luci dinamiche. Tuttavia, il realismo è limitato dall'uso di approssimazioni in *Screen Space* (come SSR e SSAO), dove i riflessi e l'occlusione dipendono esclusivamente da ciò che è visibile nell'inquadratura, introducendo potenziali artefatti nel dataset.

**Figura 3.3:** Cinematic Viewport con visualizzazione in modalità Lit. Fonte: Rendering High Quality Frames with Movie Render Queue [23].

### 3.4.2 Path Tracing

Per superare i limiti della rasterizzazione, Unreal Engine implementa un *Path Tracer* integrato accelerato via hardware (DXR) [24]. Analogamente a Blender Cycles, questa modalità utilizza Monte Carlo [18] per risolvere l'equazione del rendering. Invece di approssimare la luce, il motore lancia raggi stocastici nella scena, accumulando campioni nel tempo per convergere verso un risultato fisicamente corretto. A differenza del Deferred Rendering, il Path Tracer gestisce nativamente:

- **Global Illumination e Riflessi:** Calcola rimbalzi multipli della luce e riflessi reali anche di oggetti fuori campo, garantendo la correttezza fisica delle ombre e della rifrazione.

- **Convergenza Progressiva:** L'immagine viene generata accumulando campioni nel tempo fino a eliminare il rumore, rendendo questa modalità idonea alla generazione di *Ground Truth* visive paragonabili ai renderizzatori offline tradizionali.

**Figura 3.4:** "Virtual tour in Unreal Engine" by ARCHVYZ. Design by Toledano Architects. Fonte: Path Tracer[24].

### 3.4.3 Movie Render Queue (MRQ)

L'esportazione del dataset non avviene tramite la cattura dello schermo in tempo reale, ma attraverso il modulo **Movie Render Queue** (MRQ). Questo strumento permette di generare frame di qualità cinematografica utilizzando tecniche di sovracampionamento (*Supersampling*) che non sarebbero sostenibili durante l'esecuzione [23].

Da questo modulo è possibile selezionare le impostazioni desiderate per il rendering finale, come formato di output, tecnica di rendering (Path tracing, Deferred, ecc.), Antialiasing e variabili di console (*Console Variables*). La qualità finale dell'immagine è determinata dal numero totale di campioni per pixel, definito dal prodotto tra campioni spaziali e temporali:

$$\text{Total Samples} = \text{Spatial Samples} \times \text{Temporal Samples}$$

La distribuzione di questi campioni influenza drasticamente il risultato:

- **Spatial Sampling:** Il motore renderizza lo stesso istante temporale più volte applicando un leggero offset (jitter) alla camera. È la tecnica raccomandata per le immagini statiche (*Stills*)[24], in quanto massimizza l'anti-aliasing

geometrico e la nitidezza dei bordi (es. usando 16 campioni spaziali e 1 temporale).

- **Temporal Sampling:** Il motore suddivide il tempo di esposizione (shutter time) in sotto-intervalli interpolando il movimento degli oggetti. È la tecnica raccomandata per le animazioni, in quanto necessaria per produrre un *Motion Blur* di alta qualità (es. usando 1 campione spaziale e multipli campioni temporali), fondamentale per garantire la coerenza temporale e produrre un Motion Blur accurato, mitigando gli artefatti di aliasing temporale.

## 3.5 Ultralytics YOLO26

Il riconoscimento degli oggetti all'interno del dataset sintetico è affidato a **YOLO26** (You Only Look Once, versione 2026)[4], l'ultima evoluzione della famiglia di modelli sviluppata da Ultralytics. Questa iterazione è stata progettata ex-novo per massimizzare l'efficienza su dispositivi periferici, garantendo elevate prestazioni anche su sistemi a bassa potenza. Come illustrato nella Figura 3.5, YOLO26 segna un netto distacco rispetto ai predecessori (YOLOv8, v10, v11) e a modelli concorrenti come RTMDet. La curva di trade-off evidenzia come, a parità di latenza su hardware TensorRT, YOLO26 garantisca un valore di mAP significativamente superiore, confermandosi come la soluzione ottimale per applicazioni di visione *real-time* in contesti con risorse computazionali limitate.

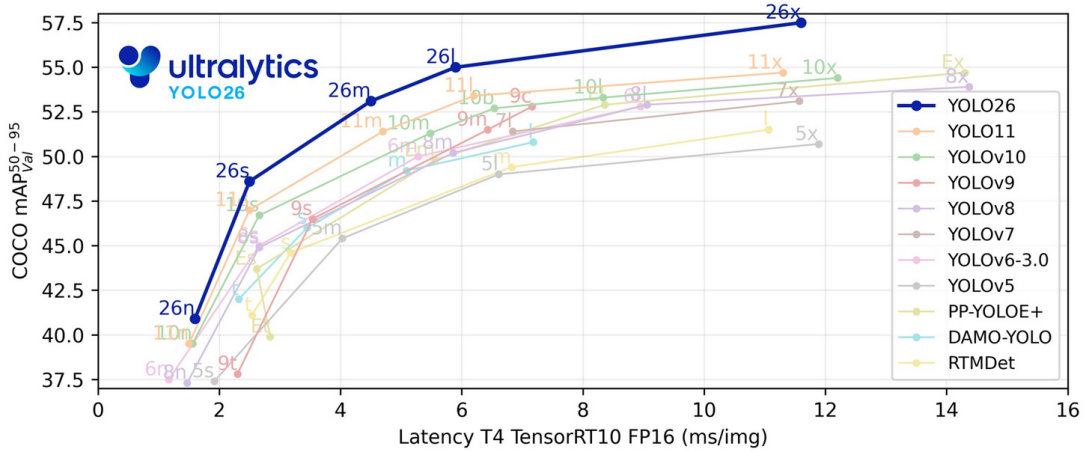



**Figura 3.5:** Confronto delle prestazioni tra YOLO26 e i principali modelli di object detection della famiglia YOLO e concorrenti. Si nota la superiorità di YOLO26 in termini di mAP COCO rispetto alla latenza (ms/img). Fonte: Ultralytics Documentation [4].

A differenza delle pipeline di visione tradizionali, YOLO26 introduce un'architettura semplificata che mira a eliminare i colli di bottiglia computazionali tipici della post-elaborazione, rendendo il modello particolarmente idoneo all'integrazione in sistemi real-time critici come la guida autonoma o la robotica mobile.

### 3.5.1 Innovazioni architetturali e principi di design

L'architettura di YOLO26 è governata da tre pilastri tecnologici che ne definiscono l'efficienza operativa:

- **Inferenza End-to-End NMS-Free:** Tradizionalmente, i modelli YOLO richiedono una fase di *soppressione non massima* (NMS) per eliminare le predizioni ridondanti. YOLO26 è un modello nativamente end-to-end che produce previsioni dirette, eliminando la necessità di NMS. [4] Questo approccio, ereditato dalle innovazioni di YOLOv10, riduce drasticamente la latenza di inferenza e semplifica l'esportazione verso hardware specializzato.
- **Rimozione della Distribution Focal Loss (DFL):** Il modulo DFL, sebbene efficace per la precisione dei bounding box, introduce complessità nel processo di esportazione e limita la compatibilità hardware. YOLO26 rimuove completamente questo modulo, garantendo un'inferenza fino al 43% più veloce su CPU rispetto alle iterazioni precedenti.[4]
- **Ottimizzatore MuSGD:** Il modello introduce un nuovo ottimizzatore ibrido denominato *MuSGD*, che combina lo *Stochastic Gradient Descent* (SGD) con tecniche di ottimizzazione *Muon*. Ispirato ai recenti progressi nell'addestramento dei Large Language Models (LLM), MuSGD offre una convergenza più rapida e una maggiore stabilità durante il processo di training, trasferendo metodologie di ottimizzazione avanzate dal deep learning testuale alla computer vision.[4]

### 3.5.2 Ottimizzazioni specifiche per i task di visione

YOLO26 non si limita all'*Object Detection*, ma implementa miglioramenti mirati per una suite completa di task di visione artificiale, molti dei quali sono oggetto di analisi in questo lavoro di tesi:

- **Object Detection e ProgLoss**: Per migliorare il riconoscimento di elementi critici nel traffico urbano, come veicoli distanti o piccoli ostacoli, YOLO26 adotta le funzioni di perdita **ProgLoss** e **STAL** [4]. Queste innovazioni aumentano la sensibilità del modello verso gli oggetti di piccole dimensioni, un requisito fondamentale per le applicazioni di sicurezza stradale e IoT.

- **Instance Segmentation multiscala**: Il modello introduce miglioramenti significativi nella segmentazione di istanza attraverso l'implementazione di una specifica *semantic segmentation loss* e di informazioni *multiscala* [4]. Queste aggiunte garantiscono una qualità superiore delle maschere di segmentazione, facilitando la distinzione precisa tra veicoli sovrapposti o parzialmente occlusi.
- **Rilevamento di oggetti orientati (OBB) e Posa**: Per task specializzati, YOLO26 integra una *angle loss* ottimizzata per risolvere le discontinuità dei bordi nei bounding box orientati (OBB) e la *Residual Log-Likelihood Estimation* (RLE) per una localizzazione millimetrica dei punti chiave nella stima della posa [4].

In sintesi, YOLO26 si configura come un framework unificato e versatile, capace di processare le diverse varianti di dataset prodotte, garantendo un'implementazione robusta e scalabile su diverse architetture hardware.

# Capitolo 4

# Sviluppo

In questo capitolo viene descritto nel dettaglio il flusso di lavoro sviluppato per la creazione del dataset sintetico. A partire dall'analisi tecnologica svolta nel capitolo precedente, si illustreranno le fasi operative che hanno portato alla costruzione dell'ambiente virtuale in Blender, alla sua replica fedele in Unreal Engine 5 e alla configurazione delle rispettive pipeline di rendering. L'obiettivo è documentare le scelte tecniche adottate per garantire la coerenza visiva e semantica tra i due motori, condizione necessaria per validare il confronto qualitativo dei dati prodotti.

## 4.1 Setup sperimentale e hardware

Prima di dettagliare le fasi di sviluppo, è necessario definire l'ambiente hardware utilizzato. La generazione di dataset sintetici tramite *Path Tracing* e la gestione di scene urbane massive richiedono una potenza di calcolo elevata, specialmente per quanto riguarda la memoria video e la velocità di processamento degli script.

Le simulazioni e i rendering sono stati eseguiti su una workstation dotata delle seguenti specifiche:

- **GPU:** NVIDIA GeForce RTX 5090 (32 GB VRAM), fondamentale per gestire la complessità della scena in Blender e Unreal Engine e, soprattutto, per la fase di rendering;
- **CPU:** Intel Core Ultra 9 285K;
- **RAM:** 64 GB DDR5, necessaria per la gestione simultanea dei software di modellazione e dei database di asset;
- **Storage:** SSD da 2 TB per garantire un'elevata velocità di scrittura durante il salvataggio dei batch di immagini.

## 4.2 Creazione della scena in Blender

La prima fase del lavoro ha riguardato la modellazione dell'ambiente urbano e la definizione delle dinamiche di scena all'interno di Blender. Poiché Blender rappresenta il punto di ingresso della pipeline, la struttura dei dati qui definita determina la qualità di tutto il processo successivo, inclusa l'esportazione verso Unreal Engine.

### 4.2.1 Modellazione procedurale della città

Per la generazione dell'ambiente urbano è stato utilizzato l'add-on *The City Generator*, sfruttando i parametri procedurali per ottenere un layout urbano completo e funzionale.

Il processo di costruzione si è concentrato sulla definizione strutturale della scena e sulla sua ottimizzazione.

**Definizione del layout urbano** Partendo da una geometria di riferimento, sono state generate contestualmente sia le strade che gli edifici. Una scelta di sviluppo fondamentale è stata quella di separare la mesh di controllo degli edifici da quella stradale. Questa separazione ha permesso di intervenire indipendentemente sui palazzi senza interferire con la topologia delle strade. Ad esempio, assegnare diverse altezze ai palazzi facenti parte dello stesso blocco richiedeva dei *loop cut* che, se applicati a una mesh di controllo stradale, potevano essere interpretati erroneamente come nuove strade.

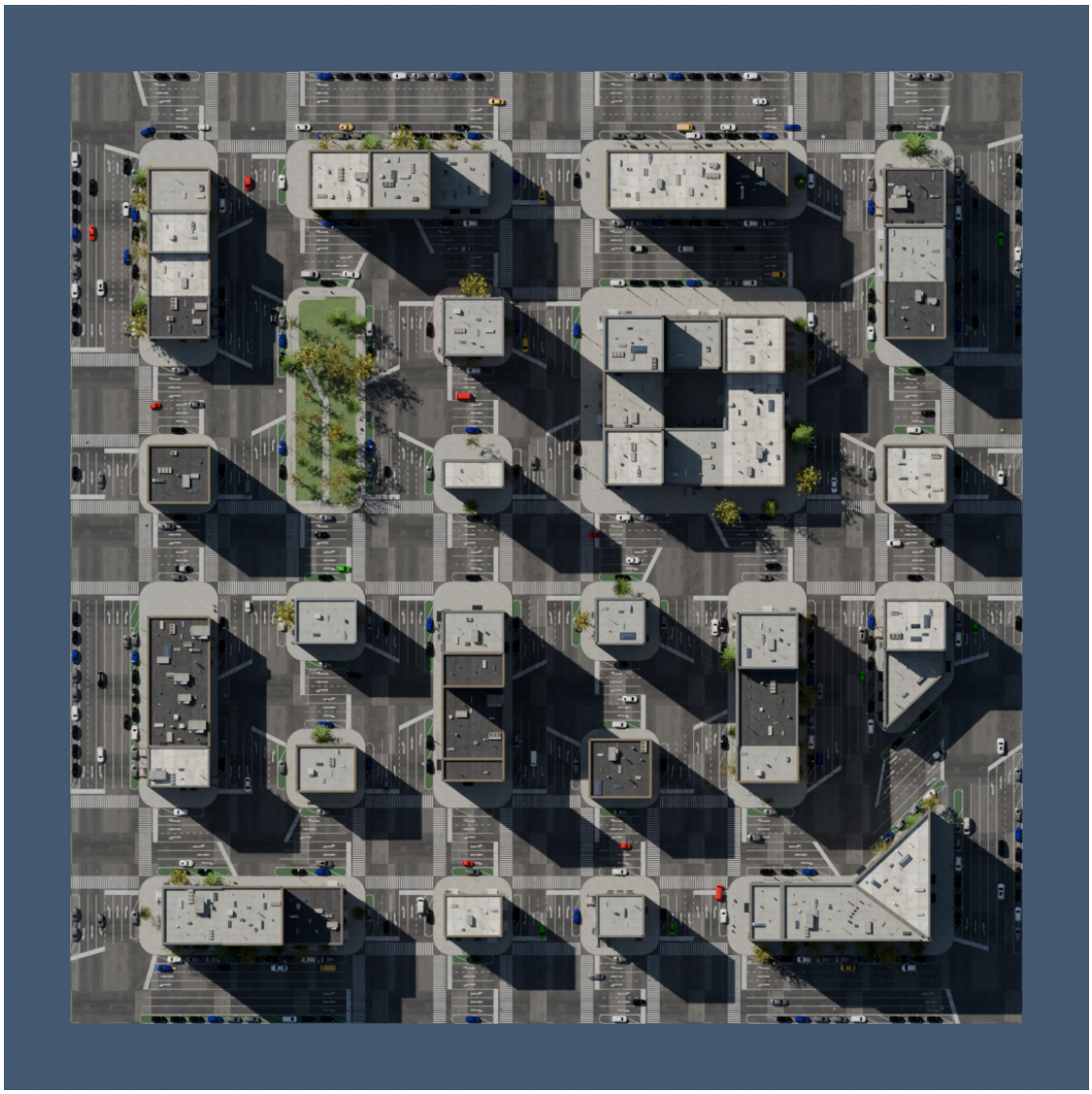


**Figura 4.1:** Vista dall'alto del layout urbano.

**Figura 4.2:** Vista isometrica della città.

**Parametrizzazione della geometria stradale e vegetazione** Per garantire uno scenario urbano denso e adatto alla simulazione del traffico, i parametri di generazione sono stati calibrati per massimizzare la complessità della scena. La sezione stradale è stata impostata con una larghezza (*Street Width*) di 22 metri, suddivisa in 4 corsie di marcia. Questa ampiezza è funzionale alla creazione di situazioni di traffico multi-livello, in cui i veicoli in primo piano possono occludere parzialmente quelli sulle corsie opposte, arricchendo la casistica per l'addestramento della rete.

Particolare attenzione è stata posta alla generazione della vegetazione e delle aree verdi (*Parks*), assegnate proceduralmente a specifiche facce della geometria di base. Attraverso i parametri di *Tree Density* e distribuzione casuale (*Seed*), sono stati istanziati alberi lungo i marciapiedi e nelle aiuole spartitraffico. La presenza di fogliame e tronchi a bordo strada non è puramente estetica, ma introduce nel dataset il fattore di *occlusione ambientale*: la rete neurale dovrà imparare a riconoscere i veicoli anche quando la loro sagoma è parzialmente interrotta da elementi naturali, aumentando la robustezza del modello finale.

**Ottimizzazione e ambiente di sfondo** Per bilanciare la complessità geometrica con le risorse di calcolo, la scena è stata strutturata su due livelli di dettaglio. La zona centrale, dove avvengono le riprese e la simulazione del traffico, è stata modellata ad alta densità poligonale e ricca di dettagli. Per evitare che l'orizzonte

risultasse vuoto durante i movimenti di camera, è stata generata una seconda città di contorno che circonda l'area principale. Questa sezione è composta da geometrie a bassa risoluzione e priva di logiche di traffico, fungendo esclusivamente da riempimento visivo per garantire la coerenza delle informazioni di sfondo nelle immagini del dataset.

**Figura 4.3:** Vista dall'alto del layout urbano completo.

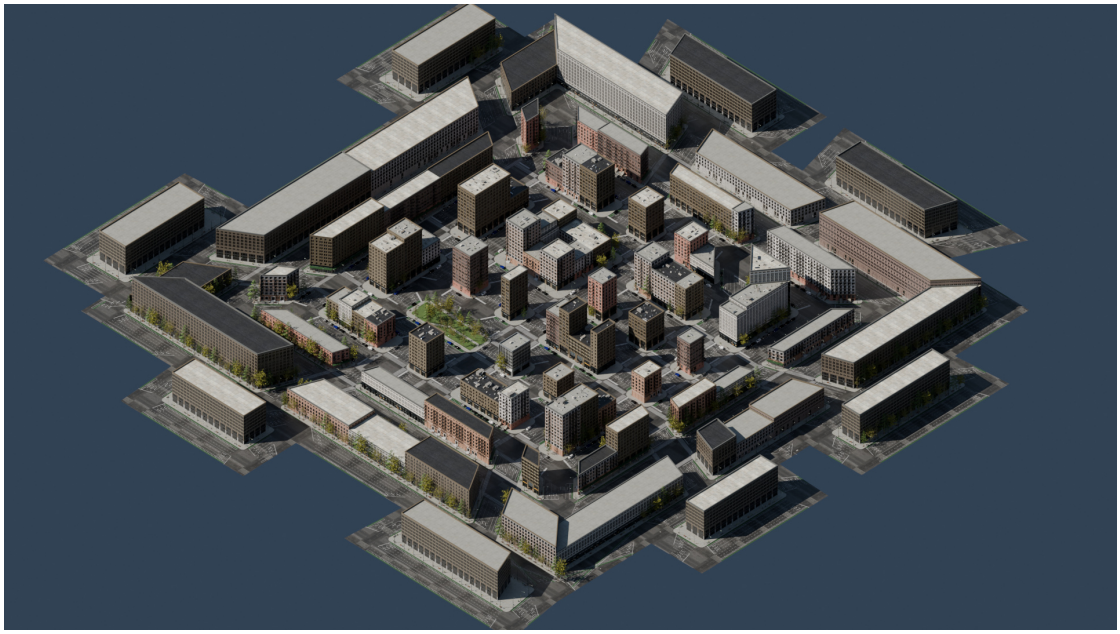


**Figura 4.4:** Vista isometrica della città completa.

### 4.2.2 Estensione e ottimizzazione del parco veicoli

Per garantire una sufficiente varianza semantica nel dataset e prevenire fenomeni di *overfitting* durante l'addestramento della rete neurale, è stato necessario espandere significativamente il parco veicoli rispetto agli asset predefiniti dell'add-on. La selezione dei modelli e la loro integrazione nella pipeline procedurale hanno richiesto una serie di interventi mirati di ottimizzazione geometrica e standardizzazione strutturale.

**Acquisizione e selezione dei modelli** Sono stati integrati nella scena numerosi modelli tridimensionali di terze parti (provenienti da repository come CGTrader e Sketchfab), selezionati per coprire diverse categorie di veicoli e garantire eterogeneità visiva. Il parco auto finale comprende:

- **Vetture standard:** utilitarie e berline (es. Alfa Romeo Giulietta, Fiat 127, Ford Focus, Ford Sedan, Mercedes-Benz 560 SEL, Fiat Multipla, Nissan 350Z);
- **Veicoli di fascia alta:** SUV e auto di lusso (es. Audi A7, Audi Q8, Mercedes-Benz Classe A, BYD SUV);
- **Veicoli commerciali e speciali:** furgoni e mezzi di soccorso (Van Volkswagen, Van generico, Ambulanza, Auto della Polizia).

**Ottimizzazione geometrica** I modelli grezzi importati presentavano spesso una topologia non ottimizzata per il rendering in tempo reale o per la simulazione massiva. Sono state quindi applicate strategie di riduzione poligonale per alleggerire il carico computazionale senza compromettere la qualità visiva:

- **Conversione topologica:** Utilizzo dell'operazione *Tris to Quads* per pulire la geometria e migliorare il flusso dei loop poligonali;
- **Decimazione selettiva:** Applicazione del modificatore *Decimate* sugli interni dei veicoli, che risultano poco visibili nelle riprese esterne, preservando invece il dettaglio della carrozzeria;
- **Rimozione geometria nascosta:** Eliminazione manuale delle facce interne e delle parti meccaniche non visibili (motore, sottoscocca dettagliato).

**Figura 4.5:** Visualizzazione wireframe di una Nissan 350Z (a sinistra) e di un'Audi A7 (a destra) ottimizzate.

**Standardizzazione per la simulazione procedurale** Per consentire all'add-on *The City Generator* di controllare il traffico e l'animazione delle ruote, ogni veicolo necessitava di essere rinominato, adattandosi allo standard gerarchico richiesto dallo script:

- **Separazione delle mesh:** Ogni auto è stata suddivisa in tre oggetti distinti: carrozzeria (*Body*), ruote anteriori (*Front Wheels*) e ruote posteriori (*Back Wheels*).
- **Nomenclatura:** È stata adottata la convenzione di naming tassativa richiesta dall' add-on (es. *01 car body*, *01 car front wheels*, *01 car back wheels*).
- **Configurazione dei pivot:** I punti di origine (*Origin Points*) delle ruote sono stati riposizionati esattamente al centro dell'asse di rotazione tra la ruota destra e sinistra. Questo passaggio è fondamentale per garantire che l'animazione procedurale della rotazione e della sterzata avvenga correttamente lungo l'asse locale, evitando artefatti visivi durante il movimento.

**Unificazione dei materiali e variazioni cromatiche** Al fine di facilitare la successiva esportazione verso Unreal Engine 5 e ridurre le *draw calls* in fase di rendering, è stata attuata una strategia di razionalizzazione dei materiali. Invece di mantenere shader unici per ogni veicolo importato, sono stati creati dei *Master Materials* condivisi per l'intero parco auto:

- **Materiali condivisi:** Vetri, pneumatici, plastiche nere, parti metalliche e cromature utilizzano lo stesso shader su tutte le auto. Sono stati preservati materiali unici solo dove strettamente necessario (es. texture specifiche per loghi, targhe o livree della polizia).
- **Varianza cromatica:** Per ogni modello sono state create 5 varianti di colore per la carrozzeria. Questo approccio, noto come *Domain Randomization*, è cruciale per insegnare alla rete neurale a riconoscere la forma del veicolo indipendentemente dal suo colore.

**Estensione del sistema di parcheggio** Infine, la logica di generazione delle auto parcheggiate a bordo strada è stata modificata. Originariamente, il sistema prevedeva l'istanziamento di un singolo modello statico. Attraverso la modifica del grafo dei *Geometry Nodes* responsabile dello scattering, il sistema è stato aggiornato per pescare casualmente da una *Collection* contenente veicoli statici di diverso colore. Contestualmente, è stato necessario intervenire sulle proprietà dei materiali dei veicoli destinati alla sosta. I modelli originali, infatti, presentavano shader emissivi attivi per i fari anteriori e posteriori. Per garantire il realismo della scena, sono state create varianti specifiche di questi materiali con emissività nulla, simulando correttamente lo stato tipico del veicolo spento delle auto in sosta.

### 4.2.3 Configurazione dell'illuminazione ambientale

L'illuminazione gioca un ruolo cruciale nel ridurre il divario tra simulazione e realtà (*sim-to-real gap*). Per garantire un fotorealismo accurato, specialmente nella resa dei materiali riflettenti come le carrozzerie dei veicoli e i vetri, si è optato per una tecnica di *Image-Based Lighting* (IBL) basata sull'utilizzo di mappe HDRI (*High Dynamic Range Imaging*).

**Strategia HDRI** A differenza dell'utilizzo di luci sintetiche (come *Sun, Point Lights* o *Area Lights*), le mappe HDRI permettono di illuminare la scena proiettando informazioni di luminanza catturate dal mondo reale a 360 gradi. Questo approccio offre due vantaggi tecnici determinanti per il training della rete neurale:

- **Fedeltà cromatica:** La luce ambientale possiede temperature colore naturali e variazioni di intensità coerenti con il cielo reale.

- **Riflessioni complesse:** Poiché le auto sono oggetti altamente speculari, le HDRI forniscono un ambiente di riflessione ricco e realistico, evitando l'aspetto "piatto" o artificiale tipico delle luci standard.

**Varianza delle condizioni atmosferiche** Per aumentare la capacità di generalizzazione del modello YOLO e prevenire l'*overfitting* su una singola condizione di luce, sono state selezionate quattro diverse mappe ambientali (provenienti dalla libreria *Poly Haven*), corrispondenti ad altrettanti scenari atmosferici:

1. **Sereno mattutino:** Luce netta, ombre marcate e temperatura colore neutra/fredda;
2. **Cielo coperto (Overcast):** Luce totalmente diffusa, ombre morbide quasi assenti e basso contrasto (scenario critico per il rilevamento dei bordi);
3. **Parzialmente nuvoloso:** Illuminazione diffusa ma con direzionalità solare ancora presente;
4. **Tramonto:** Illuminazione calda, ombre lunghe e basso angolo di incidenza solare.

**Figura 4.6:** Confronto dei quattro scenari di illuminazione ambientale.

### 4.2.4 Simulazione del traffico

Una volta definito l'ambiente statico e il parco veicoli, è stata configurata la logica di movimento per popolare la scena urbana. La simulazione non utilizza un motore fisico in tempo reale, che risulterebbe eccessivamente oneroso per il calcolo, ma si basa su un sistema di inseguimento di percorsi (*Path Following*) gestito dall'add-on.

**Configurazione dei percorsi** La rete stradale è stata coperta attraverso la generazione procedurale di curve spline. È stato impostato un parametro di *Path Amount* pari a 50 per garantire una saturazione uniforme dell'area urbana: un numero inferiore di percorsi avrebbe lasciato alcune sezioni della città vuote, mentre un numero eccessivo avrebbe creato ingorghi irrealistici e sovrapposizioni ingestibili. Le curve generate coprono tutte le corsie disponibili, inclusi gli incroci e le svolte, garantendo che i veicoli appaiano in ogni porzione del campo visivo della telecamera virtuale.

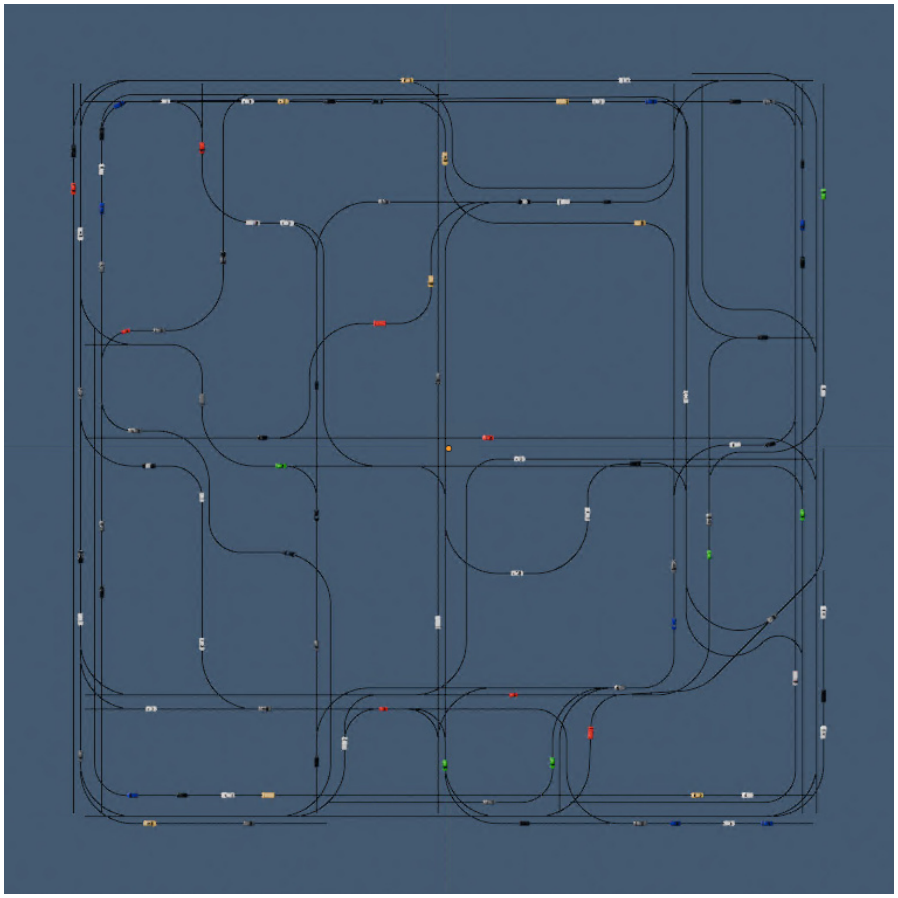


**Figura 4.7:** Visualizzazione dall'alto della rete di nodi e percorsi spline per la simulazione del traffico procedurale.

**Gestione delle collisioni e distanziamento** Per mitigare la compenetrazione delle mesh, ovvero quando due veicoli si sovrappongono fisicamente, generando dati di training corrotti o irrealistici, sono stati calibrati due parametri fondamentali:

- **Car distance Min (15 m):** Imposta la distanza iniziale minima di sicurezza tra le origini dei veicoli. Questo "cuscinetto" invisibile impedisce le collisioni a catena durante i rallentamenti.
- **Intersection distance (16 m):** Regola la logica di precedenza agli incroci, evitando che i veicoli su percorsi perpendicolari si scontrino al centro dell'intersezione.

**Dinamica e varianza** Per evitare che il traffico sembrasse robotico o eccessivamente sincronizzato, è stata introdotta una componente stocastica nella velocità di percorrenza. I parametri di *Min Speed* (0.400) e *Max Speed* (0.600) definiscono un

intervallo entro il quale ogni istanza veicolare varia la propria andatura. Inoltre, l'utilizzo di un *Instance Seed* (impostato a 7) e di un *Simulation Seed* assicura che, sebbene la generazione sia casuale, essa rimanga deterministica e riproducibile se necessario.

Infine, il sistema pesca casualmente i modelli 3D dalla *Collection* ottimizzata descritta nella sezione precedente, assegnando automaticamente le ruote corrette ai vincoli di rotazione per simulare il rotolamento.

### 4.2.5 Esportazione della scena verso Unreal Engine

Il trasferimento dell'ambiente da Blender a Unreal Engine 5 ha rappresentato una delle sfide tecniche più critiche dell'intero flusso di lavoro. La necessità di mantenere intatte le geometrie, i materiali PBR e, soprattutto, le animazioni procedurali del traffico, ha richiesto l'abbandono dei formati di interscambio standard a favore di soluzioni più robuste e lo sviluppo di script Python personalizzati.

**Scelta del formato: limiti di FBX e adozione di glTF** Inizialmente è stato testato il formato FBX, standard industriale per lo scambio di asset 3D. Tuttavia, i test preliminari hanno evidenziato criticità insormontabili: tempi di importazione in Unreal eccessivamente lunghi per la geometria urbana e, soprattutto, una mappatura errata dei materiali, che richiedeva una ricostruzione manuale degli shader. Seguendo il tutorial **How to Export The City Generator | Blender to Unreal Engine** [25], si è optato per il formato glTF 2.0 (*GL Transmission Format*). Questo formato ha garantito una corrispondenza quasi perfetta dei materiali e una gestione più efficiente delle gerarchie di oggetti.

**Esportazione dell'ambiente statico** La città generata proceduralmente è composta da migliaia di istanze. Per esportare correttamente questa geometria mantenendo un frame rate accettabile, è stato necessario convertire le istanze procedurali in mesh reali, pur mantenendo una struttura gerarchica. La procedura operativa ha previsto:

1. La conversione delle istanze in oggetti reali (*Make Instances Real*);
2. L'applicazione delle trasformazioni visive (*Visual Transform to Mesh*) sull'oggetto generatore principale;
3. L'esportazione in formato *.gltf* separando le texture in una cartella esterna per facilitarne la gestione in Unreal Engine.

**Il problema dell'animazione del traffico** La sfida maggiore ha riguardato l'esportazione dei veicoli. Poiché il movimento delle auto in Blender era gestito tramite nodi di simulazione (*Simulation Nodes*) e curve, non esistevano keyframe tradizionali sulla timeline. Di conseguenza, Unreal Engine interpreta le auto come mesh statiche immobili. Per risolvere il problema, è stata sviluppata una pipeline di automazione basata su tre script Python distinti.

**1. Definizione keyframe** Il primo script è stato progettato per impostare un keyframe ad ogni istante per ogni veicolo. Lo script itera su ogni frame della timeline, legge la posizione e la rotazione calcolata dai nodi per ogni istanza di auto e inserisce un keyframe (LocRotScale) su un nuovo oggetto fisico. Di seguito si riporta la logica principale dell'algoritmo di campionamento:

```
for f in range(START_FRAME, END_FRAME + 1):
scene.frame_set(f)
instances_f = get_instances_data(f)

for i in range(len(instances_f)):
    obj = baked_objs[i]
    obj.matrix_world = instances_f[i].matrix_world
    obj.keyframe_insert(data_path="location", frame=f)
    obj.keyframe_insert(data_path="rotation_euler", frame=f)
```

**Listing 4.1:** Logica di inserimento Keyframe

**2. Conversione in NLA Tracks** Una volta ottenuti i keyframe, è emersa una seconda limitazione: l'exporter glTF gestisce le animazioni in modo ottimale solo se organizzate in tracce NLA (*Non-Linear Animation*). Un secondo script ha automatizzato il processo di "Push Down", convertendo l'azione attiva di ogni veicolo in una striscia NLA indipendente, rendendola compatibile con lo standard di esportazione.

**3. Esportazione sequenziale dei modelli animati** Infine, si è riscontrato che l'esportatore glTF di Blender non supportava correttamente l'export simultaneo di centinaia di oggetti animati con tracce NLA separate in un unico file (o in batch automatico). Per ovviare a ciò, è stato scritto un terzo script di *Batch Export*. L'algoritmo seleziona un veicolo alla volta, lo isola, e avvia l'export glTF con parametri specifici, salvando ogni auto come file asset individuale.

```
def export_one_object(obj):
    bpy.ops.export_scene.gltf(
        filepath=os.path.join(export_folder, f"{obj.name}.gltf"),
        export_format='GLTF_SEPARATE',
        use_selection=True,
        export_animations=True,
        export_nla_strips=True
)

for m in selected_meshes:
    export_one_object(m)
```

**Listing 4.2:** Snippet dello script di Batch Export

Questa procedura ha permesso di ottenere in Unreal Engine una cartella di asset pronti all'uso, ciascuno con la propria animazione perfettamente sincronizzata con la scena urbana.

## 4.3 Replica della scena in Unreal Engine 5

Una volta completata la generazione degli asset in Blender, la fase successiva prevedeva la ricostruzione fedele dell'ambiente all'interno di Unreal Engine 5. L'obiettivo primario di questa fase non è stato il semplice import dei modelli, ma la garanzia della *cross-engine consistency*: ovvero assicurare che la resa visiva, l'illuminazione e il comportamento dinamico della scena fossero percettivamente identici tra i due motori di rendering, condizione necessaria per la validità del confronto finale.

### 4.3.1 Importazione degli asset

L'importazione è avvenuta sfruttando i file in formato glTF generati dagli script di automazione descritti nel paragrafo precedente. Durante questa procedura, è stata prestata particolare attenzione alla gestione della geometria. Sebbene Unreal Engine 5 introduca la tecnologia *Nanite* per la virtualizzazione della geometria, in questo specifico caso d'uso si è scelto di disabilitare Nanite per tutti gli asset importati. I test preliminari, infatti, avevano evidenziato la comparsa di artefatti visivi su alcune mesh. L'utilizzo della pipeline di rendering classica per le mesh statiche e scheletriche ha garantito una rappresentazione priva di difetti e geometricamente accurata.

### 4.3.2 Mantenimento della coerenza visiva

Per replicare le condizioni di illuminazione definite in Blender, non è stato sufficiente importare le mappe ambientali, ma è stato necessario configurare una pipeline di illuminazione ibrida e calibrare rigorosamente il post-processing.

**Configurazione HDRI Backdrop** Per la gestione dell'illuminazione basata su immagini (IBL), è stato attivato il plugin nativo *HDRI Backdrop*. A differenza della semplice *Sky Light*, questo strumento proietta la mappa ambientale su una geometria sferica (*Enviro Dome*) che avvolge la scena, fornendo non solo luce ma anche un contesto visivo corretto per le riflessioni a terra. Un passaggio critico ha riguardato l'allineamento dell'orientamento solare. Poiché i due software utilizzano sistemi di coordinate differenti, è stato necessario ricalcolare l'angolo di rotazione della *Source Cubemap*. Dato un angolo di rotazione ZBlender impostato a 90° nel software di origine, la corretta rotazione in Unreal ZUnreal è stata determinata secondo la relazione:

$$Z_{\text{Unreal}} = -90^\circ - Z_{\text{Blender}} = -90^\circ - 90^\circ = -180^\circ \equiv 180^\circ \tag{4.1}$$

**Figura 4.8:** Visualizzazione prospettica della città in Unreal Engine.

**Illuminazione e correzione materiali** Sebbene l'HDRI fornisca un'illuminazione ambientale eccellente, in alcuni scenari (specialmente quelli soleggiati) la sola mappa non era sufficiente a generare ombre nette e definite come quelle del motore

Cycles. Per ovviare a ciò, è stata introdotta una *Directional Light* supplementare, allineata manualmente alla posizione del sole nella texture HDRI.

**Post-process volume e esposizione** Per garantire che i dati generati fossero deterministici e non influenzati da adattamenti automatici della luminosità, è stato inserito un *Post Process Volume* globale (impostato su *Infinite Extent*). I parametri chiave del Post-Process Volume sono stati calibrati per allineare la resa visiva a quella di Blender. L'esposizione è stata bloccata impostando i valori di compensazione (*Min* e *Max EV100*) a 1, eliminando così l'adattamento automatico della luminosità. Contestualmente, è stato attivato il metodo *Bloom Standard* per gestire correttamente la dispersione della luce sulle superfici riflettenti.

### 4.3.3 Project settings e gestione della skin cache

La gestione del traffico in Unreal Engine si basa sull'utilizzo di *Skeletal Meshes* animate tramite *Animation Sequences*. Dato l'elevato numero di veicoli presenti simultaneamente in scena (288 agenti), la gestione delle deformazioni delle mesh ha richiesto un'ottimizzazione specifica a livello di motore.

Per garantire la corretta riproduzione delle animazioni e prevenire artefatti visivi durante il movimento delle ruote, è stato necessario abilitare il sistema di **Compute Skin Cache**. Questa tecnologia sposta il calcolo della deformazione dei vertici (skinning) dalla CPU alla GPU (tramite Compute Shaders), permettendo inoltre il ricalcolo delle tangenti per frame, essenziale per una corretta illuminazione dei modelli in movimento.

Lo sviluppo ha richiesto due interventi:

1. **Configurazione globale:** Nei *Project Settings*, il limite di memoria per la Skin Cache è stato innalzato a 1024 MB (Il valore di default da 128 MB risulta spesso insufficiente per scene urbane affollate);

2. **Configurazione locale:** Per ogni istanza di *Animation Sequence* posizionata nel livello, è stata forzata l'abilitazione dell'opzione *Skin Cache Usage* dal pannello dettagli, assicurando che ogni veicolo beneficiasse dell'accelerazione hardware per la deformazione geometrica.

## 4.4 Pipeline di rendering Cycles - BlenderProc

Sulla base del framework BlenderProc (3.3), lo sviluppo della pipeline di generazione dati è avvenuta attraverso lo sviluppo di uno script Python dedicato, progettato per orchestrare le API di BlenderProc all'interno dell'ambiente Cycles. L'obiettivo di questa fase è trasformare la scena statica e ottimizzata (descritta nella Sezione 4.2) in un dataset dinamico e annotato, automatizzando le operazioni di variazione ambientale e campionamento. Il codice sviluppato gestisce l'intero processo in un flusso sequenziale, comprendendo il caricamento della scena, il posizionamento della camera, le impostazioni di rendering e la serializzazione delle annotazioni YOLO, come dettagliato nelle sezioni seguenti.

### 4.4.1 Caricamento scena e assegnazione semantica

La fase di inizializzazione dello script gestisce l'importazione dell'asset urbano principale (*City2500.blend*). A differenza di un caricamento standard, questa procedura include passaggi critici per la corretta interpretazione della geometria da parte del motore di segmentazione:

- **Filtraggio e configurazione ottica:** Tramite il modulo *Loader*, vengono importate esclusivamente le mesh e le camere. Per queste ultime, il framework preserva fedelmente i parametri ottici impostati in Blender. Nello specifico, è stata utilizzata una lunghezza focale di 50mm e una dimensione del sensore (*Sensor Size*) di 35mm. Inoltre, la funzione di *Depth of Field* (profondità di campo) è stata disattivata per garantire che ogni elemento della scena, indipendentemente dalla distanza focale, risultasse perfettamente nitido per facilitare la successiva fase di etichettatura.

- **Mapping delle classi:** Lo script itera su tutti gli oggetti caricati applicando una logica di classificazione basata sul nome. Agli oggetti contenenti le stringhe identificative *"body"* o *"wheels"* viene assegnato il *category_id = 2* (classe "auto"), mentre a tutti gli altri elementi (strade, edifici, props) viene assegnato il *category_id = 1* (classe "background").

```
1     for obj in objs:
2         name = obj.get_name().lower()
3         if "body" in name or "wheels" in name:
4             obj.set_cp("category_id", 2)
5         else:
6             obj.set_cp("category_id", 1)
```

**Listing 4.3:** Logica di mapping delle classi

- **Gestione delle istanze:** Per consentire la segmentazione delle singole auto (e non solo della categoria), è stato sviluppato un algoritmo che raggruppa le parti disgiunte dello stesso veicolo. Poiché ogni auto è composta da tre mesh separate (corpo, ruote anteriori, ruote posteriori) che condividono un suffisso numerico nel nome (es. *.001*), lo script assegna lo stesso *Instance ID* univoco a tutte le parti che condividono tale suffisso. Questo garantisce che, nella maschera di segmentazione finale, l'auto venga percepita come un'entità singola e coerente.

```
instance_counter = 1
suffix_to_instance = {}
for obj in objs:
    name = obj.get_name().lower()
    if any(k in name for k in ["body", "front wheels", "back
   wheels"]):
        if "." in name:
            suffix = name.split(".")[-1]
        else:
            continue
        if suffix not in suffix_to_instance:
            suffix_to_instance[suffix] = instance_counter
            instance_counter += 1
        obj.set_cp("instance_id", suffix_to_instance[suffix])
```

**Listing 4.4:** Assegnazione di Istance ID unico per auto (body + ruote)

### 4.4.2 Applicazione delle mappe HDRI

Per gestire la mole di dati prevista (10.000 immagini totali) e per poter far fronte all'uso di 4 HDRI diverse, l'esecuzione è stata suddivisa in 4 processi *batch* sequenziali da 2.500 frame ciascuno. Quindi lo script accetta un argomento da riga di comando (indice di run) che determina l'intervallo della simulazione (0-2499, 2500-4999, 5000-7499 e 7500-9999) e quale condizione di illuminazione applicare. Utilizzando le funzioni del modulo *bproc.world*, viene caricata la mappa HDRI corrispondente al batch corrente (*Sereno mattutino* per il batch 0, *Cielo coperto* per il batch 1, *Parzialmente nuvoloso* per il batch 2, *Tramonto* per il batch 3). Con tale modulo è stato possibile decidere la forza (*Strength*) della mappa HDRI e la rotazione di quest'ultima, in questo caso impostata a 90 gradi sull'asse Z.

### 4.4.3 Posizionamento stocastico della camera

La generazione delle viste è gestita dalla funzione *add_random_street_camera*, che sviluppa un algoritmo di *Raycasting* a due stadi per identificare pose valide

e semanticamente rilevanti. L'algoritmo opera secondo i seguenti step per ogni frame:

1. **Campionamento posizione (XY):** Viene estratta una coordinata planare casuale $(x, y)$ all'interno dei confini (*Bounding Box*) della città (solo la parte in cui circolano le auto).

```
city_bbox = get_world_bbox(city_plane)
min_x, min_y = city_bbox[:, :2].min(axis=0)
max_x, max_y = city_bbox[:, :2].max(axis=0)
x = random.uniform(min_x, max_x)
y = random.uniform(min_y, max_y)
```

**Listing 4.5:** Posizionamento casuale della camera nei confini della città

2. **Raycast verticale (Ground Detection):** Un raggio viene lanciato dall'alto verso il basso ($Z \rightarrow -Z$) per intercettare la geometria sottostante. Se il punto di impatto appartiene a oggetti non calpestabili (definiti nella lista *FORBIDDEN_PREFIXES*, come tetti o cime di alberi), la posizione viene scartata per evitare compenetrazioni o punti di vista irrealistici.

```
ray_origin = mathutils.Vector((x, y, city_bbox[:, 2].max() +
    100.0))
ray_dir = mathutils.Vector((0, 0, -1))
hit, loc, normal, fi, hit_obj, _ = scene.ray_cast(depsgraph,
    ray_origin, ray_dir)
```

**Listing 4.6:** Raycast verticale

3. **Setup camera:** Se il punto a terra è valido, la camera viene posizionata a un'altezza variabile ($h \in [1.6\text{m}, 22.5\text{m}]$) e orientata con angoli di *Yaw* casuali $(0 - 2\pi)$ e *Pitch* leggermente negativo ($-0.15$ / $-0.45$ rad) per inquadrare la strada.

```
    if not hit:
        continue
    if is_forbidden_object(hit_obj):
        continue
    cam_height = random.uniform(*cam_height_range)
    cam_location = loc + mathutils.Vector((0, 0, cam_height))

    yaw = random.uniform(0, 2 * math.pi)
    pitch = random.uniform(-0.45, -0.15)
```

**Listing 4.7:** Setup della camera

4. **Validazione semantica:** Per evitare immagini prive di significato ai fini dell'addestramento (es. strade vuote o muri), viene eseguita una scansione frontale. L'algoritmo proietta un ventaglio di 10 raggi lungo il frustum visivo della telecamera; la posa viene accettata solo se almeno uno di questi raggi intercetta un oggetto classificato come "auto".

```
num_rays = 10

# Genera un ventaglio di raggi con offset orizzontale
for offset in np.linspace(-0.4, 0.4, num_rays):
    dir = mathutils.Vector((
        forward_dir.x + offset,
        forward_dir.y + offset,
        forward_dir.z
    )).normalized()

    # Lancia il raggio nella scena valutata (max 120m)
    hit, _, _, _, hit_obj, _ = scene.ray_cast(
        depsgraph, cam_location, dir, distance=120.0
    )

    # Se colpisce un'auto, la posa è valida
    if hit and is_car_object(hit_obj):
        found_car = True
        break

if not found_car:
    continue
```

**Listing 4.8:** Algoritmo di ricerca attiva dei veicoli tramite Raycast multipli

5. **Esportazione e conversione delle pose:** Una volta validata la posa per il frame corrente, i parametri di posizione e rotazione vengono salvati simultaneamente in due file di testo distinti. Il primo file mantiene le coordinate native di Blender, mentre il secondo esegue una conversione in tempo reale per il sistema di riferimento di Unreal Engine 5. Questa doppia registrazione è fondamentale per la *cross-engine consistency*: Unreal Engine utilizza un sistema di coordinate *left-handed* (Z-up) e scale differenti (centimetri invece di metri) rispetto a Blender.

Le formule di conversione applicate per mappare la trasformazione della camera da Blender verso Unreal sono le seguenti:

**Conversione Posizione (cm)**

- $X_U = X_B \cdot 100$
- $Y_U = -Y_B \cdot 100$
- $Z_U = Z_B \cdot 100$

**Conversione Rotazione (deg)**

- $Roll_U = RotY_B$
- $Pitch_U = RotX_B - 90°$
- $Yaw_U = -90° - RotZ_B$

### 4.4.4 Rendering e generazione output

Una volta definita la posa della camera, la pipeline procede con la fase di rendering. L'obiettivo primario è ottenere immagini fotorealistiche che massimizzino la qualità dei riflessi sulle superfici dei veicoli.

**Configurazione del motore Cycles** I parametri di rendering sono stati impostati per privilegiare la fedeltà fisica rispetto alla velocità di esecuzione. In particolare, è stato definito un numero elevato di campioni (*Samples*) e, fattore cruciale per le superfici vetrate delle auto, è stato aumentato il numero di rimbalzi della luce (*Light Bounces*) per la trasmissione e la trasparenza.

```
bproc.renderer.set_max_amount_of_samples(4096)
bproc.camera.set_resolution(image_width=1920, image_height=1080)
bproc.renderer.set_noise_threshold(0.01)

bproc.renderer.set_light_bounces(
    diffuse_bounces=8,
    glossy_bounces=8,
    transmission_bounces=12,
    volume_bounces=0,
    max_bounces=12,
    transparent_max_bounces=32
)
```

**Listing 4.9:** Configurazione parametri Cycles

**Configurazione della segmentazione** Parallelamente al rendering RGB, BlenderProc viene istruito per generare le mappe di *Ground Truth* tramite la funzione *enable_segmentation_output* la quale consente l'uso di più chiavi per produrre immagini sia semantiche sia con istanze coerenti tra i frame: *category_id* (classe), *instance_id* (per distinguere singole istanze di auto) e *instance* (per distinguere ogni mesh).

```
bproc.renderer.enable_segmentation_output(
    map_by=["category_id", "instance_id", "instance"],
    default_values={"category_id": 0, "instance_id": 0},
    output_dir=SEG_OUTPUT_DIR,
```

```
5       file_prefix=SEG_PREFIX
6   )
```

**Listing 4.10:** Abilitazione output segmentazione

**Post-Processing in-memory e output** A differenza della modalità standard di BlenderProc, che prevede la scrittura diretta dei file su disco, si è optato per il recupero dei dati di rendering in memoria (*return_data=True*). Questa scelta architetturale è fondamentale per la pipeline di annotazione: permette allo script di accedere immediatamente alle matrici grezze di segmentazione (*instance_id_segmaps* e *category_id_segmaps*) sotto forma di array NumPy. Disponendo dei dati in memoria, è possibile inviarli alla funzione di post-processing personalizzata che calcola le coordinate dei Bounding Box e genera i file di annotazione YOLO contestualmente al salvataggio delle immagini RGB. L'indice finale dei file viene calcolato sommando l'indice del frame corrente all'offset del batch ($l \times$ frame_end), dove $l$ è l'argomento passato da riga di comando.

```
1   data = bproc.renderer.render(return_data=True)
2
3   instance_segs = data["instance_id_segmaps"]
4   category_segs = data["category_id_segmaps"]
5
6   # Salvataggio Immagini RGB
7   for i, img in enumerate(data["colors"]):
8       index = l * bpy.context.scene.frame_end + i
9       imageio.imwrite(
10          os.path.join(rgb_out, f"{FILE_PREFIX}{index:04d}.png"),
        img
11      )
```

**Listing 4.11:** Gestione in-memory per generazione annotazioni

**Verifica visiva della segmentazione** Per finalità esclusivamente di debug, lo script include una routine di post-processing che genera mappe di segmentazione a colori. In queste immagini, a ogni istanza identificata come veicolo viene assegnata una cromia casuale, mentre lo sfondo e gli oggetti non rilevanti vengono renderizzati in nero. Questo output visivo permette di validare immediatamente la qualità dei dati, assicurando che le diverse parti dell'auto (carrozzeria e ruote) siano state correttamente aggregate e che le maschere di segmentazione aderiscano perfettamente alla geometria del modello 3D.

### 4.4.5 Generazione delle annotazioni Bounding Box

Un altro passaggio della pipeline riguarda la traduzione delle mappe di segmentazione in annotazioni geometriche utilizzabili per l'addestramento di reti di *Object Detection* (formato YOLO). A differenza degli approcci manuali o basati su stime, in questo contesto i rettangoli di delimitazione (*Bounding Box*) vengono derivati analiticamente dalle maschere di segmentazione, garantendo una precisione assoluta rispetto alla geometria visibile.

Il modulo di post-processing analizza le matrici di segmentazione conservate in memoria. Per ogni istanza identificata come veicolo, l'algoritmo estrae le coordinate di tutti i pixel attivi e ne calcola gli estremi spaziali (minimi e massimi) sugli assi orizzontale e verticale. Tali valori vengono successivamente trasformati per soddisfare le specifiche del formato YOLO, normalizzando le coordinate nell'intervallo $[0, 1]$ rispetto alla risoluzione dell'immagine. Infine, viene applicato un filtro euristico per scartare annotazioni relative a oggetti troppo piccoli (rumore o veicoli in lontananza) che non fornirebbero caratteristiche utili all'apprendimento della rete.

Di seguito viene mostrato il nucleo logico della funzione, evidenziando la trasformazione matematica da coordinate pixel a formato YOLO normalizzato:

```
# Estrazione delle coordinate dei pixel appartenenti all'auto
ys, xs = np.where(mask)

if len(xs) > 0 and len(ys) > 0:
    #Identificazione degli estremi del rettangolo (in pixel)
    x_min, x_max = xs.min(), xs.max()
    y_min, y_max = ys.min(), ys.max()

    # Conversione in formato YOLO (Centro, Larghezza, Altezza)
    # e normalizzazione rispetto alle dimensioni (img_w, img_h)
    cx = ((x_min + x_max) / 2) / img_w
    cy = ((y_min + y_max) / 2) / img_h
    w = (x_max - x_min + 1) / img_w
    h = (y_max - y_min + 1) / img_h
```

**Listing 4.12:** Logica di calcolo e normalizzazione del Bounding Box

**Figura 4.9:** Rappresentazione visiva dei bounding box generati automaticamente in BlenderProc, sovrapposti a render sintetici in Cycles.

### 4.4.6 Generazione delle annotazioni per la segmentazione di istanza

Per estendere l'utilità del dataset oltre la semplice (*Object Detection*), la pipeline è stata configurata per produrre anche annotazioni compatibili con task di *Instance Segmentation* (es. per modelli YOLO26m-seg). In questo formato, l'oggetto non è descritto da un rettangolo, ma da una poligonale chiusa che segue fedelmente il profilo della sagoma del veicolo.

La funzione dedicata elabora le stesse maschere di segmentazione utilizzate in precedenza, ma ne estrae dei contorni vettoriali:

- **Estrazione dei contorni:** Utilizzando l'algoritmo di analisi topologica sviluppato nella libreria OpenCV (*cv2.findContours*), la maschera raster binaria viene convertita in una serie di coordinate vettoriali che descrivono il perimetro dell'oggetto.
- **Serializzazione:** I punti del poligono risultante vengono normalizzati rispetto alle dimensioni dell'immagine e trascritti in sequenza lineare, preceduti dall'ID di classe.

Di seguito si riporta lo snippet relativo alla logica di estrazione e normalizzazione del contorno:

```
contours, _ = cv2.findContours(
    mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if cv2.contourArea(cnt) >= 3:
        seg_points = []
        for p in cnt.squeeze():
            x = p[0] / img_w
            y = p[1] / img_h
            seg_points.append(f"{x:.6f} {y:.6f}")
```

**Listing 4.13:** Estrazione del poligono di segmentazione tramite OpenCV

**Figura 4.10:** Rappresentazione visiva della maschera di segmentazione generata automaticamente in BlenderProc rispetto al render Cycles.

### 4.4.7 Analisi delle performance computazionali (Cycles)

A conclusione del processo di generazione, si riportano i dati relativi ai tempi di calcolo necessari per la produzione dell'intero dataset di 10.000 immagini. I rendering sono stati eseguiti sulla workstation di riferimento descritta in precedenza 4.1. Il tempo totale di elaborazione è stato di circa **146 ore** (circa 6 giorni di calcolo continuativo).

È interessante notare come, a parità di complessità geometrica e risoluzione ($1920 \times 1080$), i tempi di rendering abbiano mostrato una varianza significativa in funzione delle condizioni di illuminazione (HDRI) utilizzate. La Tabella 4.1 riassume i risultati per i quattro batch di esecuzione.

**Tabella 4.1:** Tempi di rendering per batch (Cycles)

| Batch | Scenario | Tempo tot. (s) | Tempo tot. (h) | Media (s) |
|---|---|---|---|---|
| 0-2499 | Sereno | 167 962 | $\approx$ 46,6 | 67,2 |
| 2500-4999 | Nuvoloso | 114 185 | $\approx$ 31,7 | 45,7 |
| 5000-7499 | Parz. nuvoloso | 121 676 | $\approx$ 33,8 | 48,7 |
| 7500-9999 | Tramonto | 122 439 | $\approx$ 34,0 | 49,0 |

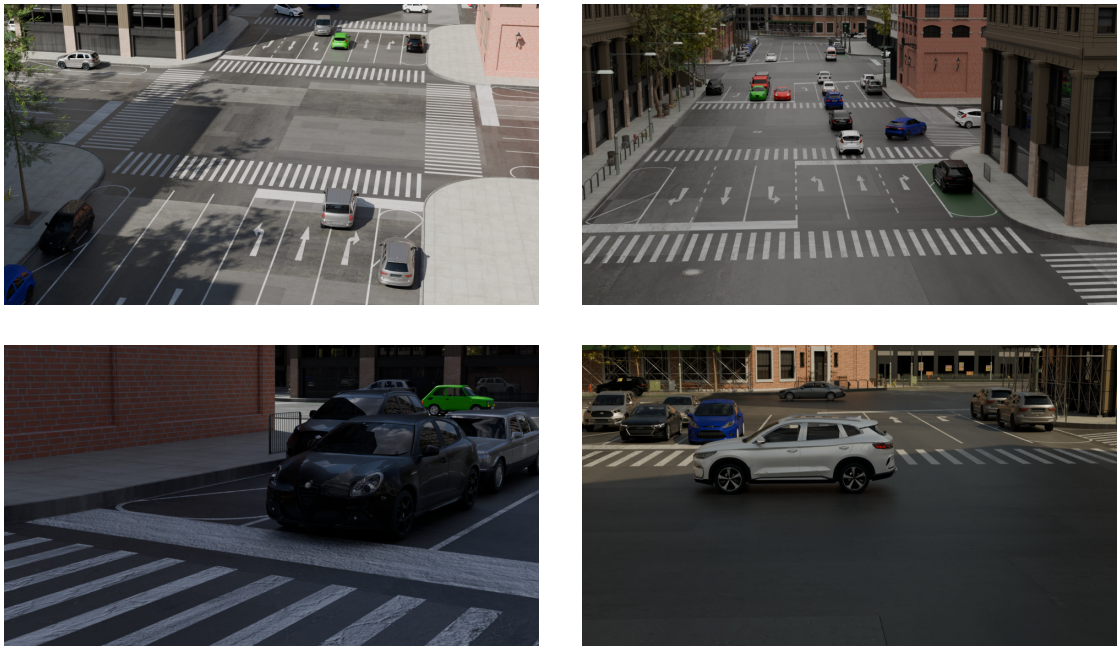


**Figura 4.11:** Rappresentazione visiva di 4 render in Cycles. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto.

**Analisi della varianza** Il divario prestazionale osservato, con il batch "Sereno mattutino" che ha richiesto circa il 47% del tempo in più rispetto allo scenario "Cielo coperto", è attribuibile al meccanismo di *Adaptive Sampling* del motore Cycles.

- **Luce diretta (Sereno):** La presenza di una fonte luminosa intensa (il sole) genera ombre nette e contrasti elevati. Queste condizioni introducono una maggiore quantità di rumore stocastico nel path tracing, costringendo l'algoritmo a utilizzare un numero maggiore di campioni per pixel per scendere sotto la soglia di rumore impostata (0.01).
- **Luce diffusa (Coperto):** L'illuminazione omogenea tipica del cielo nuvoloso genera ombre morbide e basso contrasto. Il rumore converge molto più rapidamente, permettendo al motore di rendering di terminare il calcolo del frame prima di raggiungere il limite massimo di 4096 samples.

## 4.5 Pipeline di rendering EEVEE

Al fine di valutare l'efficienza computazionale del motore *real-time* rispetto al *Path Tracing*, è stata sviluppata una pipeline di rendering parallela basata su **EEVEE-Next**. L'obiettivo di questa fase è replicare fedelmente il dataset originale, mantenendo l'identica coerenza geometrica e dinamica, ma riducendo drasticamente i tempi di calcolo.

### 4.5.1 Allineamento dinamico e lettura pose della camera

Per garantire che le inquadrature fossero perfettamente sovrapponibili a quelle generate in Cycles, è stato sviluppato uno script Python per la sincronizzazione della telecamera. Lo script legge il file generato durante la simulazione in BlenderProc (4.4.3), estraendo per ogni frame le coordinate spaziali e le rotazioni.

Attraverso l'uso delle API di Blender (*bpy*), lo script associa i dati ai corrispondenti numeri di frame nella timeline, inserendo automaticamente dei *keyframe* di tipo *Location* e *Rotation Euler*:

```
for line in lines[1:]:
    t = [x.strip() for x in line.split(',')]
    frame = int(t[0])

    loc = (float(t[1]), float(t[2]), float(t[3]))
    rot = (math.radians(float(t[4])),
           math.radians(float(t[5])),
           math.radians(float(t[6])))

    # Posizionamento e inserimento keyframe
    scene.frame_set(frame)
    cam.location = loc
    cam.rotation_euler = rot
    cam.keyframe_insert("location", frame=frame)
    cam.keyframe_insert("rotation_euler", frame=frame)
```

**Listing 4.14:** Script di sincronizzazione e keyframing della camera

### 4.5.2 Configurazioni ottimali e gestione delle trasparenze

Poiché EEVEE non utilizza il calcolo fisico dei raggi, è stata necessaria una calibrazione specifica dei parametri di rendering e dei materiali per emulare il fotorealismo di Cycles.

**Parametri di rendering** Sono stati definiti le seguenti impostazioni per massimizzare la fedeltà visiva:

- **Render Samples:** 64;
- **Illuminazione globale:** Attivazione della *Fast GI Approximation* (8 *rays*, 12 *steps*) per simulare i rimbalzi di luce indiretta;
- **Ray Tracing:** Abilitazione del *Raytracing* nativo con precisione impostata a 1.0;
- **Shadows:** Attivazione delle ombre con campionamento ottimizzato.

**Ottimizzazione dei materiali vetrati** Il materiale del vetro (*CityGen Glass*) è stato rielaborato per supportare la trasparenza in modalità EEVEE. È stato inserito un *Mix Shader* tra il materiale di base e un nodo *Transparent BSDF*, pilotato da un nodo *Fresnel* con IOR 1.2. Nelle impostazioni del materiale (*Surface Settings*), sono state attivate le opzioni *Transparent Shadows* e *Raytraced Transmission*, impostando il modello di spessore su *Slab*. Questa configurazione permette alla luce di attraversare i vetri dell'auto, rendendo visibili gli interni e migliorando drasticamente il realismo complessivo.

### 4.5.3 Rendering headless e registrazione dei tempi

Per minimizzare il consumo di risorse ed eliminare l'overhead dell'interfaccia grafica, il processo è stato eseguito in modalità **Headless** tramite terminale. Questo approccio garantisce la massima fedeltà ai tempi di calcolo puri del motore. Il comando utilizzato per il lancio del processo è: *blender.exe -b City2500.blend -P renderEEVEE.py*

Lo script di automazione integra una routine di monitoraggio che registra i timestamp di inizio e fine processo, i quali costituiscono la base per la valutazione quantitativa dell'efficienza del motore.

### 4.5.4 Analisi delle performance computazionali (EEVEE)

Sulla base dei log generati durante la fase di rendering headless, è stata condotta l'analisi dei tempi di calcolo per il motore EEVEE-Next. Per garantire la coerenza metodologica, i test sono stati eseguiti sulla medesima workstation di riferimento (4.1) utilizzata per Cycles. Il tempo totale di elaborazione per le 10.000 immagini è stato di circa **9,9 ore**, un dato che evidenzia un abbattimento dei costi computazionali di oltre il 93% rispetto alla pipeline in Cycles.

Contrariamente a quanto osservato con il *Path Tracing*, i tempi di rendering in EEVEE mostrano una stabilità quasi assoluta tra i diversi batch, con variazioni minime che non dipendono dalla natura dell'illuminazione ambientale. La Tabella 4.2 riporta i dati estratti dai log di rendering.

**Tabella 4.2:** Tempi di rendering per batch (EEVEE-Next)

| Batch | Scenario | Tempo tot. (s) | Tempo tot. (h) | Media (s) |
|---|---|---|---|---|
| 0-2499 | Sereno | 8 266,88 | ≈ 2,30 | 3,31 |
| 2500-4999 | Nuvoloso | 9 995,59 | ≈ 2,77 | 4,00 |
| 5000-7499 | Parz. nuvoloso | 8 805,18 | ≈ 2,45 | 3,52 |
| 7500-9999 | Tramonto | 8 546,94 | ≈ 2,37 | 3,42 |

**Figura 4.12:** Rappresentazione visiva di 4 render in EEVEE. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto.

**Indipendenza dalle condizioni di illuminazione** L'analisi dei dati evidenzia come il motore EEVEE non risenta del divario prestazionale tra scenari a luce diretta e luce diffusa. Questa omogeneità è dovuta all'architettura stessa del motore:

- **Carico computazionale fisso:** Essendo un motore basato su rasterizzazione, il tempo di calcolo è determinato principalmente dal numero di poligoni a schermo e dalla risoluzione dell'output, fattori rimasti invariati in tutti i batch.
- **Assenza di campionamento adattivo:** EEVEE non deve risolvere l'equazione della luce tramite la convergenza di campioni stocastici. La generazione

di ombre e riflessi avviene tramite tecniche di approssimazione (come le *Shadow Maps* e il *Screen Space Ray Tracing*) che presentano un costo computazionale costante, indipendentemente dal contrasto o dal rumore visivo della scena.

## 4.6 Pipeline di rendering Unreal Engine 5

La replica della scena in Unreal Engine 5 è stata finalizzata allo sfruttamento di due diverse tecnologie di rendering: il **Path Tracer** (PT), per la massima fedeltà fisica tramite tracciamento dei raggi, e il **Deferred Renderer**, per valutare le prestazioni di un motore ottimizzato per il tempo reale. Per garantire la perfetta corrispondenza tra i software, il progetto è stato convertito in ambiente C++.

### 4.6.1 Sviluppo del sistema di importazione pose

Al fine di garantire la *cross-engine consistency*, è stata sviluppata una classe C++ dedicata *MyActor*. Lo scopo di questo componente è automatizzare l'importazione delle pose della camera dal file generato nella pipeline BlenderProc (4.4.3), in particolare il file adattato al sistema di coordinate di Unreal Engine.

**Configurazione del modulo** Per supportare le API cinematografiche, il file di configurazione *Build.cs* è stato esteso includendo le dipendenze necessarie per la gestione dei *Sequencer* e delle camere:

```
1 PublicDependencyModuleNames.AddRange(new string[] {
2     "Core", "CoreUObject", "Engine", "InputCore",
3     "LevelSequence", "MovieScene", "MovieSceneTracks", "
    CinematicCamera"
4 });
```

**Listing 4.15:** Configurazione dipendenze nel file Build.cs

**Logica di keyframing** L'attore *MyActor* espone nell'editor un metodo (*ImportCameraPoses*) che esegue la lettura del file delle pose. L'algoritmo inserisce un **Constant Key** per ogni singolo frame; questa scelta è stata dettata dalla necessità di eliminare qualsiasi interpolazione software, garantendo che la posizione e la rotazione (Pitch, Yaw, Roll) della *CineCameraActor* corrispondano esattamente ai calcoli vettoriali eseguiti in Blender.

```
1  //Unreal rotation order: Pitch, Yaw, Roll
2  const FRotator UnrealRotation(Pitch, Yaw, Roll);
3  const FFrameNumber KeyTime(Frame);
4
5  Channels[0]->AddConstantKey(KeyTime, Location.X);
6  Channels[1]->AddConstantKey(KeyTime, Location.Y);
7  Channels[2]->AddConstantKey(KeyTime, Location.Z);
8  Channels[3]->AddConstantKey(KeyTime, UnrealRotation.Roll);
9  Channels[4]->AddConstantKey(KeyTime, UnrealRotation.Pitch);
10 Channels[5]->AddConstantKey(KeyTime, UnrealRotation.Yaw);
```

**Listing 4.16:** Core della logica di importazione pose in C++

### 4.6.2 Configurazione del Sequencer e ottimizzazione delle animazioni

L'importazione delle pose ha richiesto una calibrazione rigorosa del *Level Sequencer* per prevenire artefatti visivi e sfasamenti temporali:

- **Sincronizzazione del traffico:** Tutte le *Animation Sequences* dei veicoli sono state inserite nel Sequencer e allineate ai *Camera Cuts*. Questo ha garantito che lo stato dinamico del traffico fosse identico a quello catturato negli altri motori.
- **Gestione del tick e skin cache:** Per evitare la scomparsa improvvisa delle mesh (*culling*) o deformazioni errate durante il movimento veloce, è stata attivata l'opzione *Always tick pose and refresh bones*. Quest'impostazione, combinata con l'incremento del buffer a 1024 MB del sistema **Compute Skin Cache** nei *Project Settings*, come descritto in precedenza 4.3.3, risulta nel comportamento desiderato in fase di rendering.
- **Risoluzione temporale:** La *Tick Resolution* e il frame rate del Sequencer sono stati bloccati a 24 fps. L'utilizzo della risoluzione standard di Unreal (24000 fps) è stato evitato poiché introduceva micro-sfasamenti sub-frame durante il posizionamento delle tracce animate.

### 4.6.3 Configurazione della Movie Render Queue

Il processo di rendering è stato gestito tramite il plugin nativo **Movie Render Queue** (MRQ). Sono state configurate diverse pipeline di generazione per bilanciare la fedeltà visiva e l'efficienza computazionale:

- **Anti-Aliasing** Sia per la pipeline *Path Tracer* che per quella *Deferred*, è stato impostato uno *Spatial Sample Count* di 64 e un *Temporal Sample*

*Count* di 1. Questa configurazione è stata identificata come il compromesso ottimale per la produzione massiva di dataset statici (*stills*), garantendo una pulizia dell'immagine adeguata per addestramento della rete neurale senza tempi di calcolo proibitivi. [24] Al fine di garantire un confronto equo con il motore *Cycles* di Blender, è stata condotta una serie di test dedicata utilizzando il *Path Tracer* con un campionamento massivo di 4096 *sample*. Questa pipeline ha permesso di eliminare quasi totalmente il rumore stocastico residuo, consentendo di valutare se una maggiore pulizia del dato sintetico influisca effettivamente sulle capacità di generalizzazione del modello YOLO26 rispetto alla versione ottimizzata da 64 *sample*.

- **Setup camera:** La *CineCameraActor* è stata configurata con una focale di 50mm e un sensore con larghezza 35mm e altezza 19.6875mm, disabilitando *Depth of Field* e **Motion Blur**. L'assenza di sfocatura di movimento è risultata essenziale per mantenere la nitidezza dei veicoli durante i rapidi spostamenti della camera tra i batch.
- **Console Variables:** È stato necessario incrementare la variabile di console *r.GpuProfilerMaxEventBufferSizeKB* a 1024 per supportare la densità poligonale della scena durante il rendering.

### 4.6.4 Automazione del rendering e registrazione dei tempi

Per quantificare le prestazioni computazionali, è stato sviluppato uno script Python eseguito direttamente nell'ambiente Unreal. Lo script utilizza un *MoviePipelinePIEExecutor* per avviare la coda di rendering e aggancia una funzione di *callback* (*on_finished*) per calcolare la durata esatta di ogni batch.

### 4.6.5 Analisi delle performance computazionali (Unreal Path Tracer)

Sulla base dei log generati tramite lo script Python dedicato, è stata condotta l'analisi dei tempi di calcolo per il motore **Unreal Path Tracer**, confrontando la configurazione ottimizzata a 64 sample con quella ad alta fedeltà a 4096 sample. Per garantire la coerenza metodologica, i test sono stati eseguiti sulla medesima workstation di riferimento (4.1).

Il tempo totale di elaborazione per le 10.000 immagini con la configurazione a **64 sample** è stato di circa **5,14 ore**. Questo risultato conferma l'estrema efficienza della pipeline rispetto a Cycles, risultando circa 28 volte più veloce a parità di campionamento.

Al contrario, l'utilizzo di **4096 sample** ha comportato un incremento massiccio dell'onere computazionale, richiedendo un tempo totale di circa **323,7 ore** (oltre

13 giorni di calcolo ininterrotto). Questo incremento di circa 63 volte nel tempo di rendering è stato necessario per eguagliare il setup di riferimento di Cycles e valutare l'impatto della rimozione del rumore residuo sulle performance della rete neurale.

La Tabella 4.3 confronta i dati estratti per entrambe le configurazioni, suddivisi per scenario meteorologico.

**Tabella 4.3:** Confronto tempi di rendering: 64 vs 4096 Sample (Unreal Path Tracer)

| **Config.** | **Batch** | **Scenario** | **Tempo tot. (h)** | **Media (s)** |
|---|---|---|---|---|
| **64 Sample** | 0-2499 | Sereno | $\approx$ 1,26 | 1,81 |
| | 2500-4999 | Nuvoloso | $\approx$ 1,30 | 1,88 |
| | 5000-7499 | Parz. nuvoloso | $\approx$ 1,28 | 1,84 |
| | 7500-9999 | Tramonto | $\approx$ 1,29 | 1,86 |
| **4096 Sample** | 0-2499 | Sereno | $\approx$ 80,06 | 115,30 |
| | 2500-4999 | Nuvoloso | $\approx$ 81,61 | 117,52 |
| | 5000-7499 | Parz. nuvoloso | $\approx$ 80,86 | 116,45 |
| | 7500-9999 | Tramonto | $\approx$ 81,17 | 116,89 |

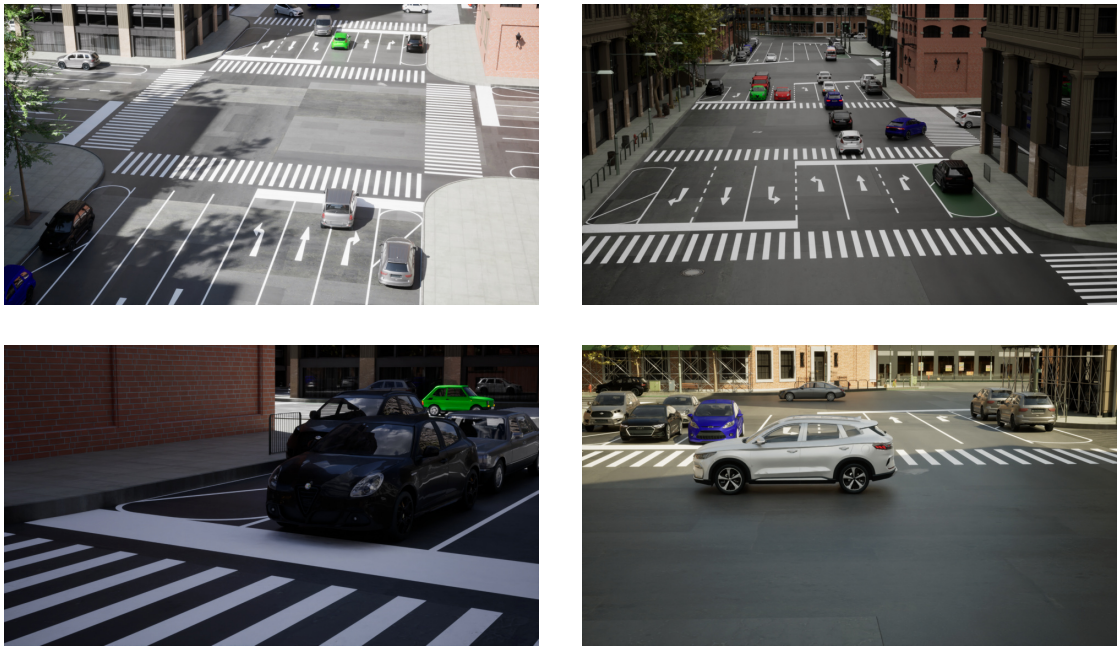


**Figura 4.13:** Rappresentazione visiva di 4 render in Unreal Path Tracing. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto.

### 4.6.6 Analisi delle performance computazionali (Unreal Deferred)

L'ultima fase dell'analisi computazionale ha riguardato il motore **Unreal Deferred Rendering**. Nonostante si tratti di una tecnica di rasterizzazione ottimizzata per il tempo reale, i parametri di alta qualità impostati nella *Movie Render Queue* hanno portato a un tempo totale di elaborazione di circa **6,22 ore**. I risultati per i singoli batch sono riportati nella Tabella 4.4.

**Tabella 4.4:** Tempi di rendering per batch (Unreal Deferred)

| Batch | Scenario | Tempo tot. (s) | Tempo tot. (h) | Media (s) |
|---|---|---|---|---|
| 0-2499 | Sereno | 5 671 | $\approx$ 1,57 | 2,26 |
| 2500-4999 | Nuvoloso | 5 284 | $\approx$ 1,47 | 2,11 |
| 5000-7499 | Parz. nuvoloso | 5 728 | $\approx$ 1,59 | 2,29 |
| 7500-9999 | Tramonto | 5 722 | $\approx$ 1,59 | 2,29 |

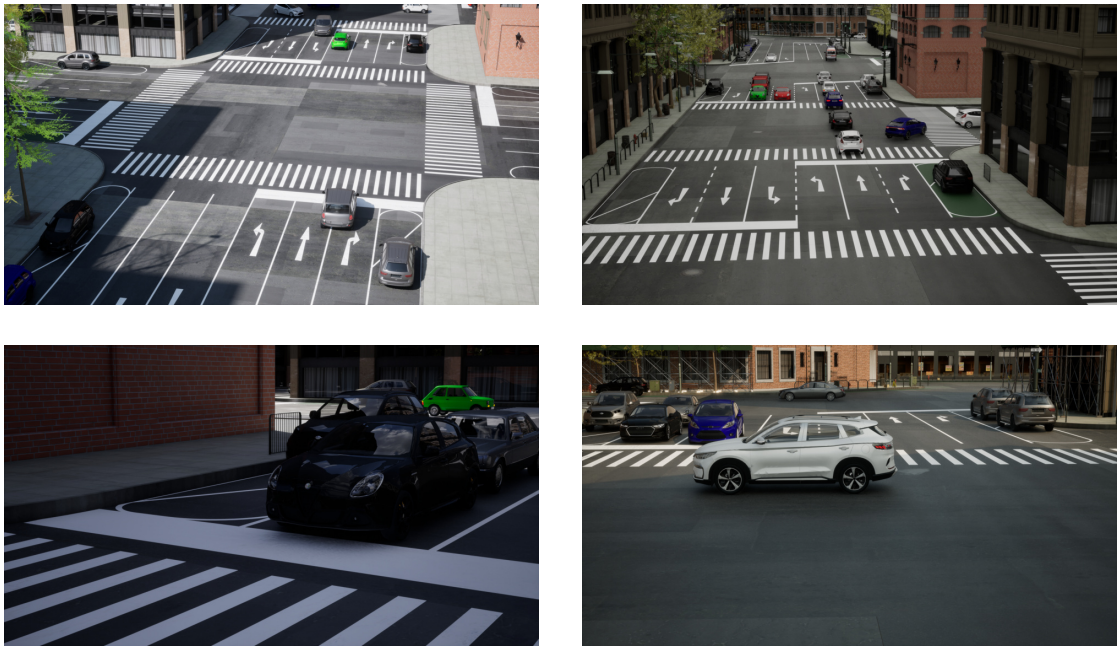


**Figura 4.14:** Rappresentazione visiva di 4 render in Unreal Deferred. In alto a sinistra con cielo sereno, in alto a destra con cielo coperto, in basso a sinistra con cielo parzialmente nuvoloso e in basso a destra al tramonto.

**Confronto con le pipeline real-time** È interessante notare come il Deferred Rendering di Unreal sia risultato leggermente più lento del Path Tracer interno. Questo fenomeno è attribuibile alle impostazioni di *Cinematic Quality*:

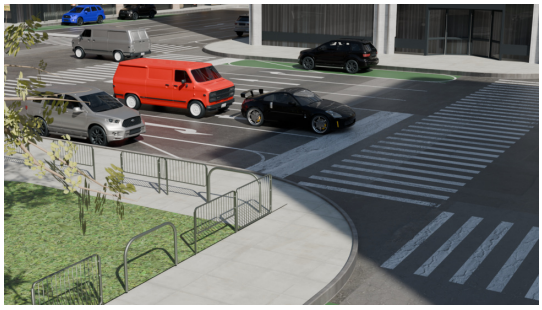
- **Overhead del Post-Processing:** La pipeline Deferred applica numerosi passaggi di post-elaborazione (Lumen, riflessi *Screen Space*, ombre virtuali) che, se impostati al massimo livello di dettaglio, possono superare il costo di un campionamento Path Tracer.
- **Stabilità computazionale:** Come per EEVEE, il tempo di calcolo rimane slegato dalle condizioni di luce HDRI, confermando la prevedibilità dei motori basati su rasterizzazione nel processo di produzione massiva di dati.

## 4.7 Confronto qualitativo tra i motori di rendering

A completamento dell'analisi tecnica, viene presentato un confronto visivo diretto della medesima inquadratura campionata dai quattro motori di rendering utilizzati.

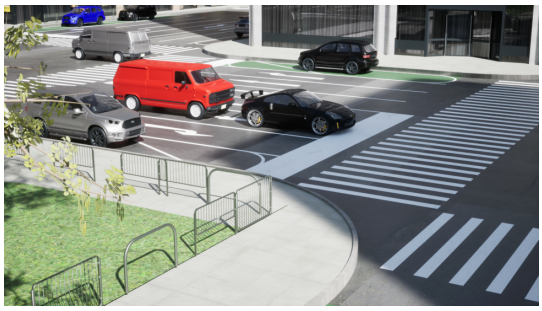
Questa comparazione permette di valutare l'efficacia della pipeline nel mantenere la coerenza geometrica e luminosa tra le diverse tecnologie.

Come si evince dalla Figura 4.15, i motori basati su *Path Tracing* (Cycles e Unreal PT) offrono la massima fedeltà nella gestione delle riflessioni speculari sulle carrozzerie e nella propagazione della luce indiretta. I motori *real-time* (EEVEE e Unreal Deferred), pur con lievi discrepanze nella resa delle ombre di contatto e nella rifrazione dei vetri, garantiscono una coerenza semantica (posizionamento degli oggetti e dei veicoli) pressoché identica, validando l'interoperabilità degli script di sincronizzazione sviluppati.



Cycles

EEVEE



UE Path Tracer

UE Deferred

Cycles

EEVEE

UE Path Tracer

UE Deferred

**Figura 4.15:** Confronto visivo della stessa istanza di simulazione renderizzata con i quattro motori della pipeline.

## 4.8 Addestramento dei modelli YOLO26

Completata la fase di generazione dei dati sintetici, la ricerca si è focalizzata sulla valutazione dell'efficacia di tali dataset nell'addestramento di architetture per la Computer Vision. L'obiettivo è duplice: determinare l'impatto della fedeltà del motore di rendering sulle capacità predittive del modello e analizzare il rapporto tra costo computazionale (tempo di rendering) e accuratezza finale.

### 4.8.1 Progettazione dei dataset per il confronto

Per garantire un'analisi rigorosa e scientifica, sono state create due diverse tipologie di campionamento per ciascuno dei cinque motori di rendering (Cycles, EEVEE, Unreal Path Tracer 64/4096 e Unreal Deferred).

**Analisi a parità di volume (Quantity-based)** È stata definita una gerarchia di dataset con dimensioni crescenti: **500, 1.000, 5.000 e 10.000 immagini**. Per ogni dataset, è stata applicata una suddivisione fissa tra *training set* (80%) e *validation set* (20%). Questa configurazione permette di isolare la variabile "qualità dell'immagine", osservando come la rete reagisce a diversi livelli di fotorealismo e rumore stocastico a parità di informazioni geometriche fornite.

**Analisi a parità di budget temporale (Time-based)** Al fine di valutare l'efficienza produttiva, sono stati generati dataset basati sul tempo di calcolo investito: **1h, 2h, 5h, 12h e 24h**.

- Per i motori *Path Tracing* intensivi (Cycles e Unreal PT 4096), i dataset coprono l'intero arco temporale fino alle 24 ore.
- Per i motori ottimizzati (EEVEE, Unreal Deferred e Unreal PT 64), l'analisi si interrompe alla soglia delle 5 ore, in quanto la loro elevata velocità di generazione ha permesso di esaurire il limite massimo di 10.000 immagini univoche previste dal framework di simulazione prima di raggiungere i target temporali superiori.

### 4.8.2 Addestramento

L'attività di addestramento ha riguardato due task distinti: **Object Detection** e **Instance Segmentation**. Sono stati impiegati i modelli allo stato dell'arte **YOLO26m** e **YOLO26m-seg**.

Per garantire la riproducibilità dell'esperimento e la validità del confronto comparativo, ogni sessione di addestramento è stata eseguita *from scratch* (senza

l'ausilio di pesi pre-addestrati su dataset reali) adottando i seguenti iperparametri fissi:

**Tabella 4.5:** Iperparametri di addestramento comuni

| **Parametro** | **Valore** |
|---|---|
| Modelli | YOLO26m, YOLO26m-seg |
| Epoche | 300 |
| Batch Size | 48 |
| Risoluzione Input | $640 \times 640$ |
| Seed | 0 (Deterministic = True) |
| Deterministic | True |

L'utilizzo di un seed deterministico e l'azzeramento dei pesi iniziali assicurano che ogni variazione osservata nelle metriche di performance ($mAP_{50}$, $mAP_{50-95}$) sia riconducibile esclusivamente alle caratteristiche intrinseche dei dati sintetici generati dai diversi motori di rendering.

# Capitolo 5

# Analisi dei risultati

In questo capitolo vengono presentati e discussi i risultati sperimentali ottenuti dall'addestramento dei modelli YOLO26m (Object Detection) e YOLO26m-seg (Instance Segmentation). L'analisi mira a determinare l'efficacia del dato sintetico nel superare il *Sim-to-Real gap*, articolandosi attraverso tre fasi metodologiche principali:

1. **Analisi Sim-to-Real a intervalli fissi:** L'addestramento dei modelli è stato condotto *from scratch* esclusivamente su dataset sintetici, campionati per volumi d'immagine e budget temporali definiti. In questa fase, la validazione durante il training è stata eseguita sul dominio sintetico, mentre il test finale è stato effettuato sul dataset reale per misurare la capacità di generalizzazione pura di ciascun motore di rendering.

2. **Cross-Engine Validation (Matrici di performance):** Per valutare la robustezza delle feature apprese, è stata condotta una validazione incrociata sui dataset massimali (10000 immagini). I modelli addestrati su uno specifico motore sono stati testati contro i set di validazione prodotti da tutti gli altri engine, generando quattro matrici di metriche che evidenziano quanto le caratteristiche di un particolare render influiscano sulla portabilità del modello verso altri domini sintetici.

3. **Ottimizzazione mediante Fine-Tuning:** Infine, è stata esplorata la strategia del *Transfer Learning*. I modelli pre-addestrati sui dataset sintetici da 10.000 immagini sono stati rifiniti (*fine-tuned*) integrando prima un dataset reale di 12000 immagini, successivamente uno da 1000. La validazione, condotta sul set reale da 535 immagini, ha permesso di quantificare il beneficio incrementale apportato dal sintetico come base di partenza per l'addestramento in contesti ad alta complessità.

Tale struttura permette di evidenziare i punti di forza e le criticità di ogni approccio, fornendo una panoramica completa sul trade-off tra fedeltà del rendering e prestazioni della rete neurale.

## 5.1 Metriche di valutazione

Per quantificare l'efficacia del modello nel task di computer vision, sono state adottate le metriche standard del benchmark Microsoft COCO:

- **Precision e Recall:** La *Precision* misura la capacità di evitare falsi positivi (predizioni errate), mentre il *Recall* indica la capacità di individuare tutti gli oggetti reali presenti nel dataset, minimizzando i falsi negativi.
- **mAP50 (Mean Average Precision):** Calcola la media della precisione su una soglia di *Intersection over Union* (IoU) del 50%. È un indicatore generale della capacità di localizzazione "grossolana".
- **mAP50-95:** Rappresenta la metrica più rigorosa, mediata su diverse soglie di IoU. Premia i modelli capaci di produrre bounding box e maschere estremamente aderenti alla sagoma reale dell'oggetto, ed è fondamentale per valutare la qualità del dato sintetico.

## 5.2 Analisi Sim-to-Real

Nelle sezioni seguenti vengono analizzati i risultati ottenuti addestrando i modelli esclusivamente su dati sintetici e testandoli sul dominio reale. Il confronto è strutturato secondo più modalità: sul dataset massimale (da 10.000 immagini), a parità di volume del dataset (Sezione 5.2.1) e a parità di budget temporale investito nella generazione (Sezione 5.2.2).

### 5.2.1 Analisi a parità di numero di immagini

Questa analisi isola la qualità del rendering, confrontando dataset di dimensioni identiche estratti dai diversi motori.

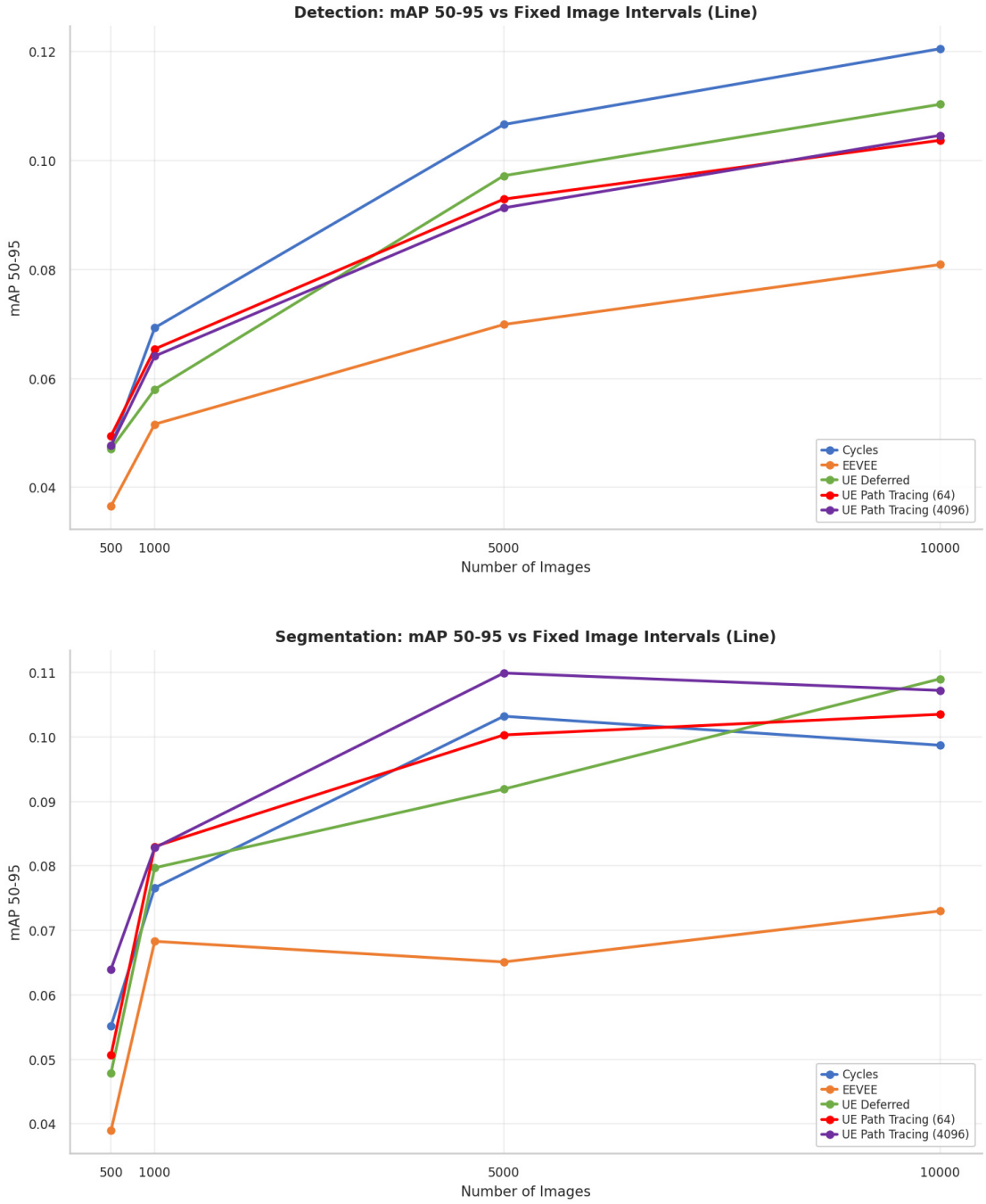



**Figura 5.1:** Andamento del mAP50-95 al variare del numero di immagini (500, 1.000, 5.000 e 10.000) per i task di Detection e Segmentation

**Analisi dei dati:** Come si osserva nella Figura 5.1, le prestazioni di tutti i motori seguono una curva di crescita logaritmica. È interessante notare come, nel task di **Segmentation**, il distacco tra i motori *Path Tracing* (Cycles e Unreal PT) e quelli *Raster* (EEVEE) sia ancora più marcato rispetto alla semplice detection.

### 5.2.2 Analisi a parità di budget temporale

Valutando il rendimento in funzione del tempo di rendering, il paradigma della "qualità assoluta" viene messo in discussione dall'efficienza produttiva.

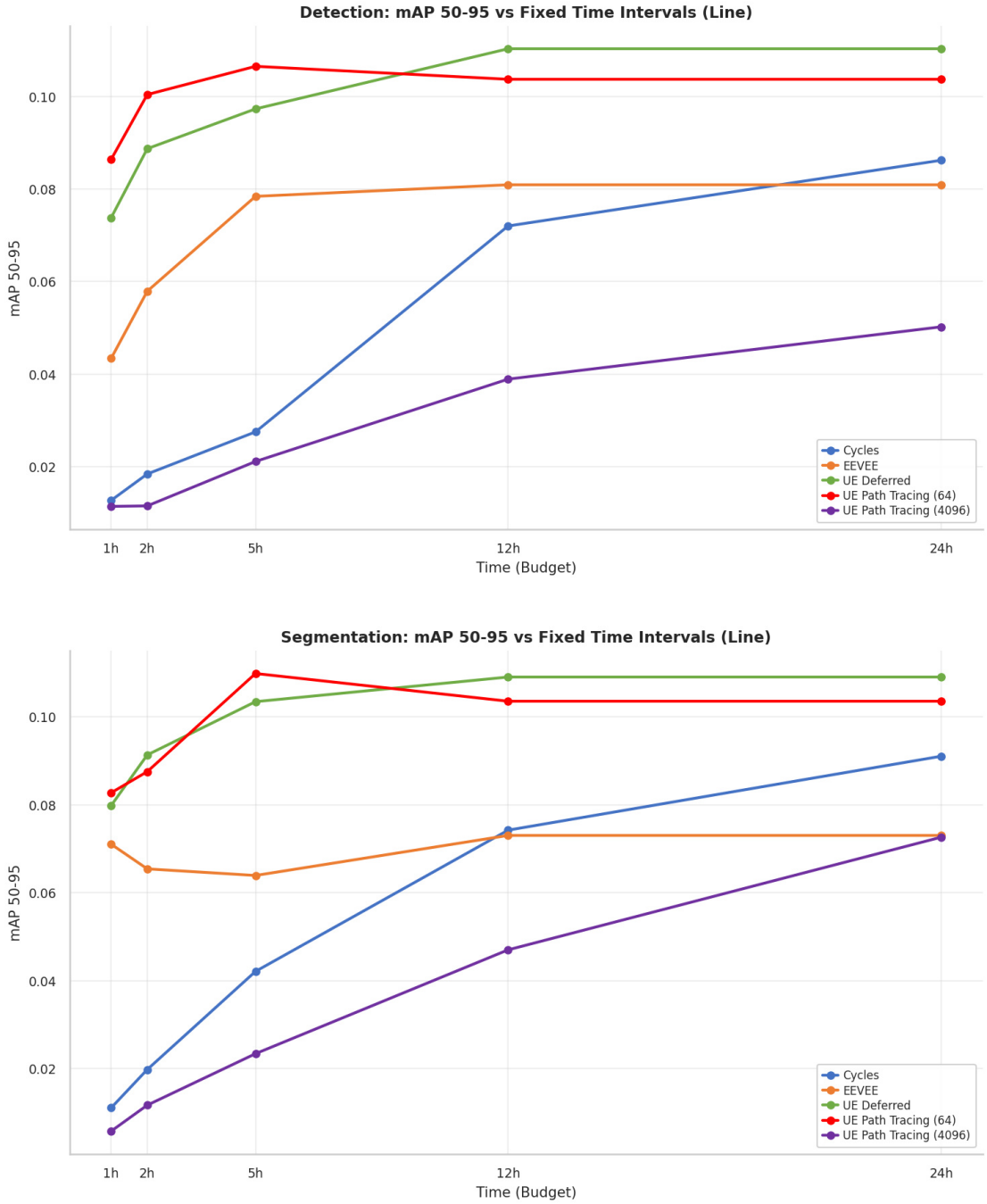



**Figura 5.2:** Andamento del mAP50-95 al variare del tempo di rendering (1h, 2h, 5h, 12h e 24h) per i task di Detection e Segmentation

**Analisi dei dati:** I risultati mostrati in Figura 5.2 evidenziano l'impatto pratico della scelta del motore di rendering. Con un budget di sole 5 ore, i motori ottimizzati (Unreal Path Tracer 64 e Unreal Deferred) hanno già addestrato il modello su quasi 10.000 immagini, raggiungendo un plateau di precisione più elevato. Nello stesso arco temporale, Cycles ha prodotto meno di 400 immagini, risultando in un modello quasi incapace di generalizzare (mAP prossimo allo zero). Emerge chiaramente che la **varietà statistica** (poter vedere 10.000 inquadrature diverse in poco tempo) è più importante per YOLO rispetto alla perfezione dei singoli frame calcolati in 146 ore di rendering.

### 5.2.3 Confronto Unreal Path Tracer: 64 vs 4096 sample

In questa sezione viene analizzato l'impatto del numero di campioni sulla qualità del dato sintetico, confrontando due configurazioni dello stesso motore: una ottimizzata a basso campionamento (64 sample) e una ad alta fedeltà (4096 sample).

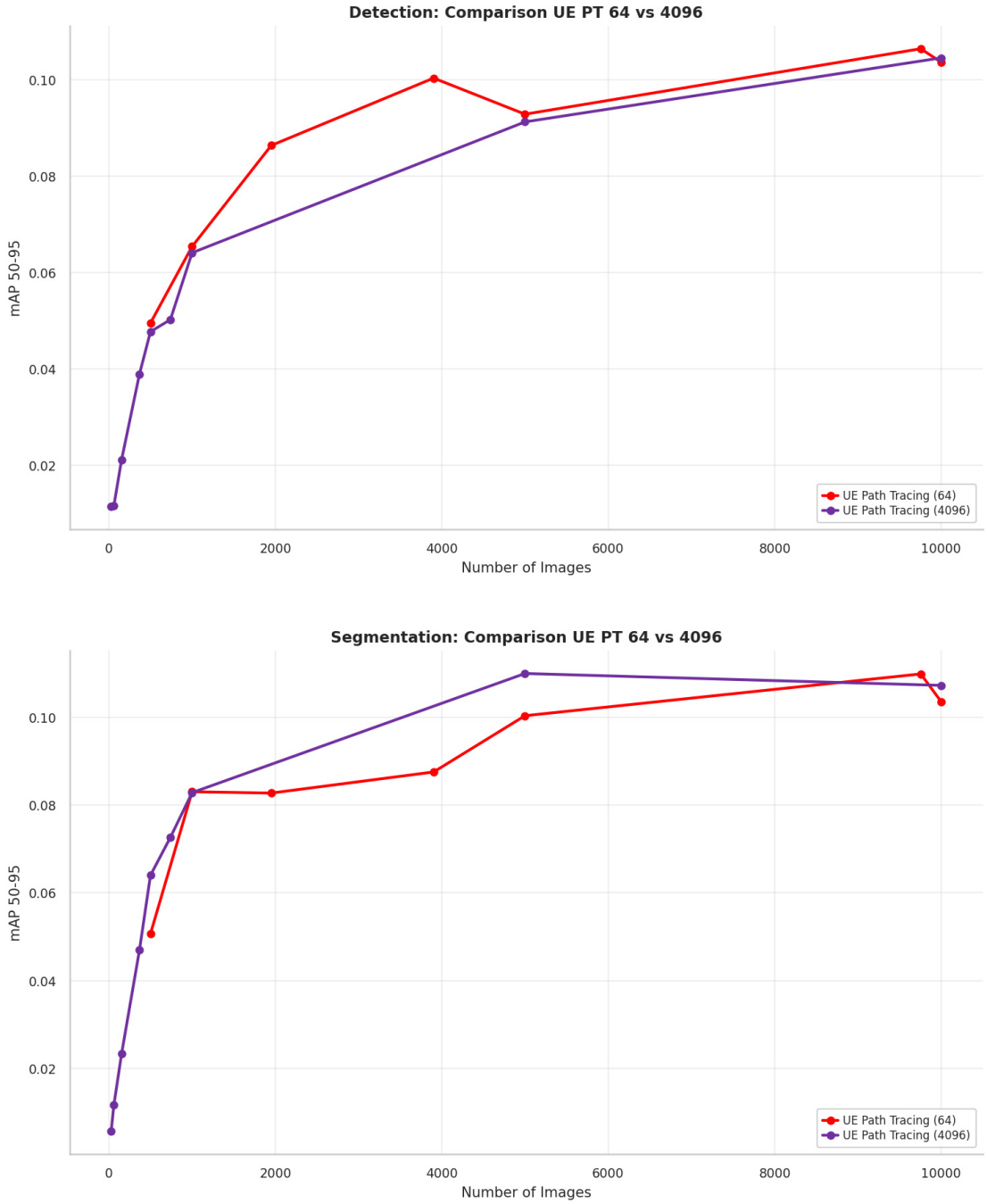



**Figura 5.3:** Andamento del mAP50-95 al variare del numero di immagini per i task di Detection e Segmentation per Unreal Path Tracing confrontando il campionamento a 64 e 4096. Il grafico include tutti gli addestramenti effettuati (1h, 2h, 5h, 12h, 24h, 500, 1.000, 5.000, 10.000)

**Analisi dei dati:** Il dato più sorprendente, ravvisabile in Figura 5.3, è l'esiguità del guadagno prestazionale a fronte di un costo computazionale immenso. Il passaggio da 64 a 4096 sample (un incremento del tempo di calcolo di circa il 6300%) produce un miglioramento del mAP50-95 inferiore al 1.5%. Questo suggerisce che YOLO26 sia robusto al rumore stocastico del Path Tracing, anche a fronte di una differenza minima tra le rispettive immagini, difficile da cogliere ad occhio nudo. Per la rete, le feature di alto livello (forma del veicolo, ruote, fanali) sono distinguibili anche in presenza di rumore, rendendo il rendering ad alta fedeltà quasi superfluo per scopi di addestramento puro.

### 5.2.4 Analisi delle metriche sul dataset massimale

La Tabella 5.1 riporta i risultati ottenuti dai modelli addestrati *from scratch* esclusivamente sui dataset sintetici da 10.000 immagini, valutati sul set reale.

**Tabella 5.1:** Metriche di performance dei modelli addestrati *from scratch* su 10.000 immagini sintetiche, valutate sul dominio reale (COCO). Precision e Recall sono indicate rispettivamente con P e R.

| **Object Detection** | | | | |
|---|---|---|---|---|
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.1205 | 0.2040 | 0.4387 | 0.2376 |
| EEVEE | 0.0809 | 0.1442 | 0.3494 | 0.1762 |
| UE Deferred | 0.1103 | 0.1770 | 0.3705 | 0.1828 |
| UE PT 64 | 0.1037 | 0.1724 | 0.4560 | 0.1886 |
| UE PT 4096 | 0.1046 | 0.1753 | 0.4066 | 0.1953 |
| **Instance Segmentation** | | | | |
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.0987 | 0.1698 | 0.4090 | 0.1919 |
| EEVEE | 0.0730 | 0.1361 | 0.3759 | 0.1708 |
| UE Deferred | 0.1090 | 0.1799 | 0.4609 | 0.1965 |
| UE PT 64 | 0.1035 | 0.1746 | 0.4183 | 0.2077 |
| UE PT 4096 | 0.1072 | 0.1835 | 0.4438 | 0.2148 |

Gli istogrammi in Figura 5.4 che seguono offrono una panoramica completa del confronto "testa a testa" tra le tecnologie.

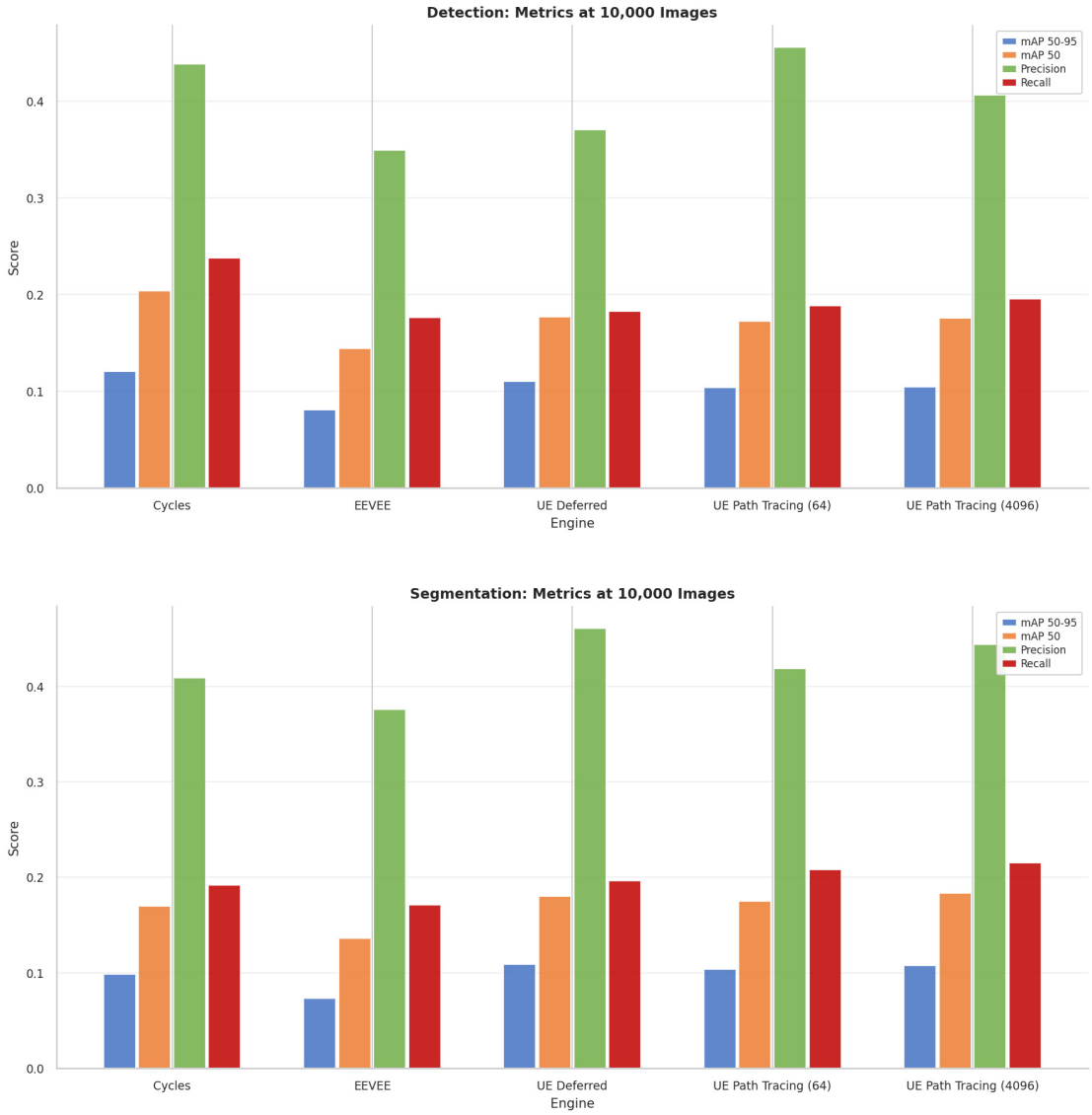



**Figura 5.4:** Confronto delle metriche mAP50-95, mAP50, Precision e Recall per tutti i motori su dataset a 10000 immagini

### 5.2.5 Precision vs Recall

L'analisi delle curve Precision-Recall permette di valutare la stabilità del modello nel discriminare i veicoli dallo sfondo complesso urbano.

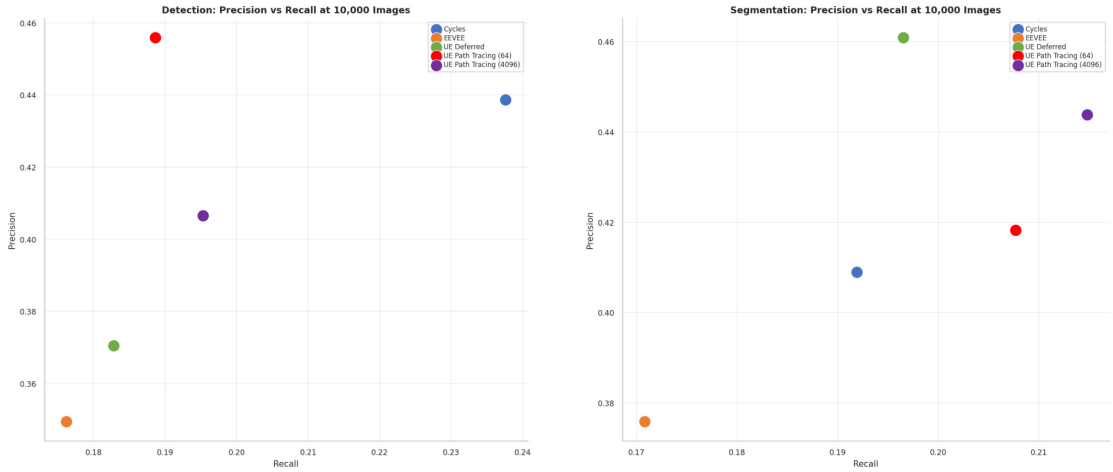



**Figura 5.5:** Confronto delle metriche di Precision e Recall per tutti i motori su dataset a 10000 immagini

**Analisi dei dati:** I motori fotorealistici garantiscono il miglior equilibrio tra Precision e Recall (Figura 5.5) in entrambi i task. Nella Detection, UE PT4096 riduce i falsi positivi grazie alla fedeltà fisica del render, mentre Cycles risulta essere il miglior compromesso tra Precision e Recall. Nella Segmentation, UE Deferred mostra la Precision più alta tramite maschere conservative, pur peccando in Recall. Il Path Tracing rimane superiore per robustezza generale, fondamentale per rilevare ostacoli in condizioni critiche. Nel contesto di questo esperimento, la qualità del simulatore sembra influenzare l'affidabilità finale del modello nel mondo reale.

### 5.2.6 Confronto qualitativo dell'inferenza su dati reali

Al fine di visualizzare l'impatto dei diversi motori di rendering sulla capacità di generalizzazione, viene presentata in questa sezione una comparazione diretta dell'inferenza effettuata dai modelli addestrati sui dataset da 10.000 immagini.

La griglia sottostante (Figura 5.6) comprende 8 campioni estratti dal dominio reale (COCO Dataset) e permette di osservare come la qualità del dato sintetico di addestramento si rifletta sulla precisione delle maschere di segmentazione e dei bounding box in ambiente reale.

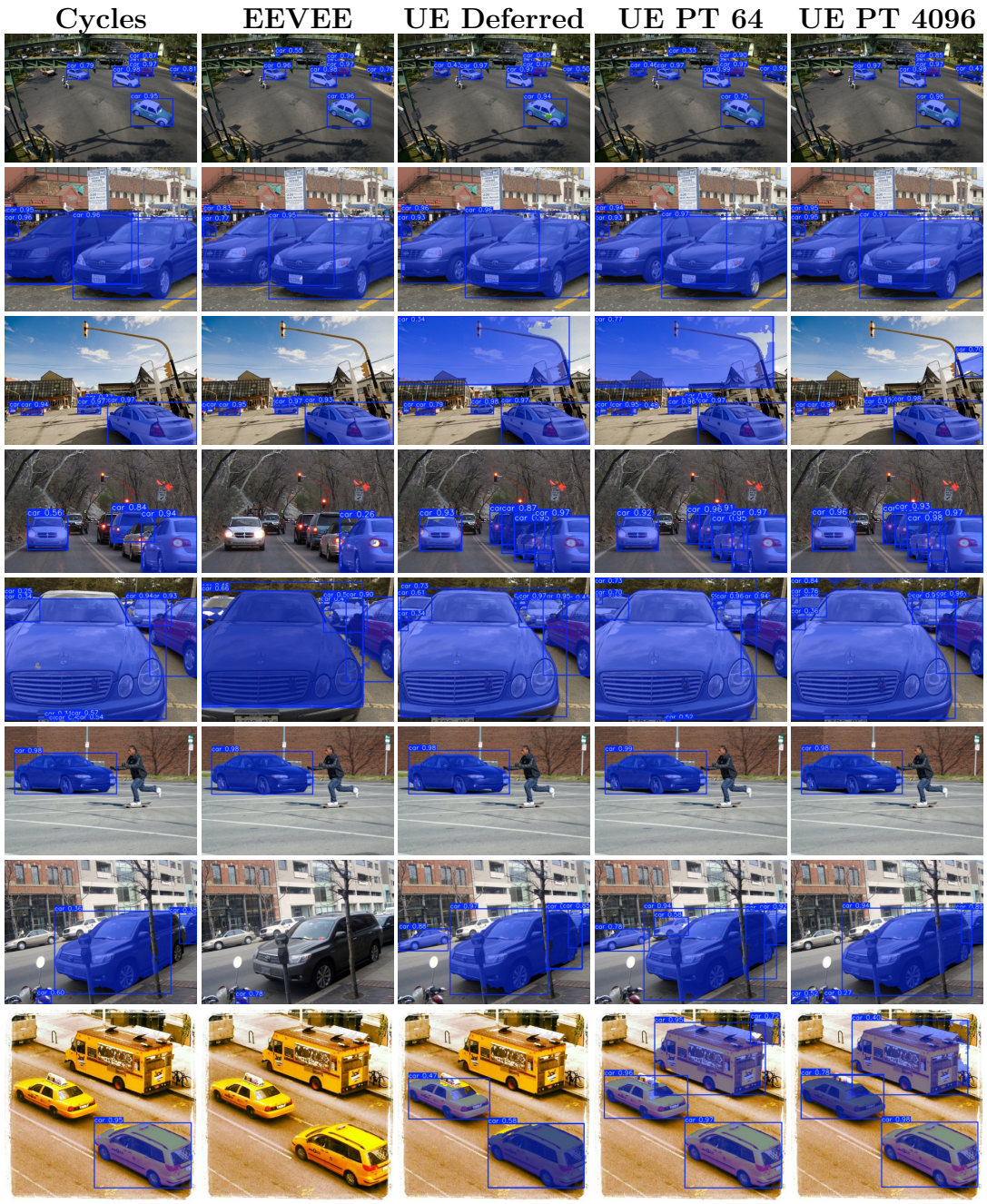



**Figura 5.6:** Matrice di confronto qualitativo: ogni riga rappresenta la medesima immagine reale processata dai cinque modelli addestrati sui diversi motori di rendering (10.000 immagini). La comparazione visiva permette di apprezzare la diversa accuratezza nelle maschere di segmentazione.

**Analisi visiva dell'inferenza:** L'ispezione della griglia (Figura 5.6) conferma i dati discussi in precedenza. Il modello addestrato su **Cycles** e **Unreal Path Tracer** mostra una maggiore precisione nel delineare le ruote e i sottoscocca, zone dove le ombre del rendering facilitano l'apprendimento dei volumi. Il modello **EEVEE** tende a produrre maschere leggermente meno aderenti ai bordi in presenza di riflessi speculari forti sulle carrozzerie. Infine, il confronto tra **UE PT 64** e **UE PT 4096** non evidenzia particolari discrepanze, validando l'ipotesi che un campionamento massivo non sia strettamente necessario per la generalizzazione sul reale.

### 5.2.7 Conclusioni sul Sim-to-Real e sul confronto tra motori di rendering

Analizzando le metriche aggregate, si osserva come Unreal Path Tracer a 64 sample rappresenti, nel contesto di questo contesto sperimentale, un buon punto di equilibrio tra qualità e velocità produttiva. Offre una precisione simile ai modelli a campionamento elevato (Cycles e PT 4096), con tempi di generazione nettamente inferiori. In particolare, nel mAP50-95 per la segmentazione, il Path Tracing a basso campionamento mantiene un vantaggio qualitativo sui bordi rispetto alla rasterizzazione pura, senza però richiedere settimane di calcolo. Questi risultati, pur promettenti, sono riferiti a una singola scena urbana e a una sola classe di oggetti, e andrebbero confermati su pipeline più diversificate.

## 5.3 Cross-Engine validation

Per valutare la capacità di generalizzazione dei modelli e l'interoperabilità tra i diversi domini sintetici, è stata condotta un'analisi di *Cross-Engine Validation*. In questa fase, i modelli addestrati sul set massimale (10.000 immagini) di un motore specifico sono stati testati sui set di validazione prodotti da tutti gli altri motori di rendering.

L'obiettivo è identificare quale motore generi le feature più "universali", ovvero quelle che permettono al modello di operare con successo anche su dati provenienti da pipeline grafiche differenti.

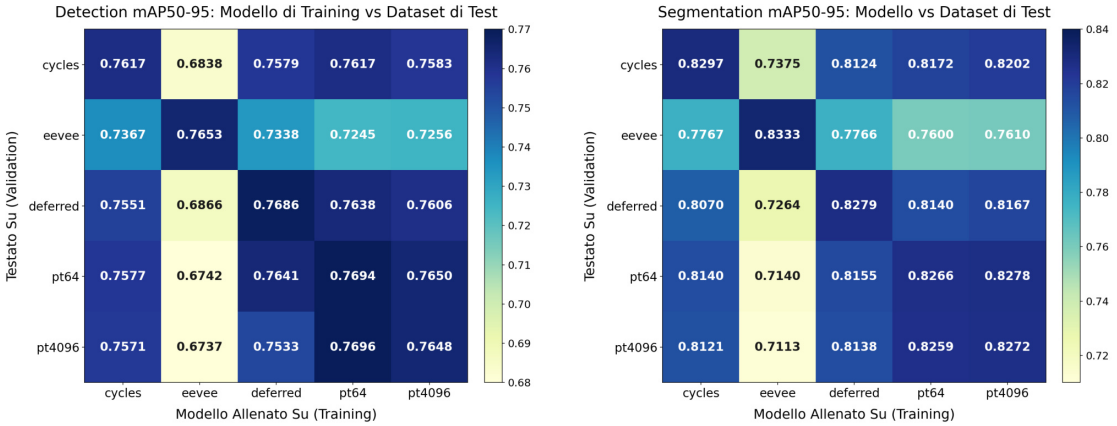



**Figura 5.7:** Matrici tra domini sintetici (mAP50-95). Le colonne rappresentano il motore di addestramento (*modello*), mentre le righe rappresentano il dataset di test.

**Analisi dei risultati:** Le matrici in Figura 5.7 mostrano una forte diagonale, confermando che ogni modello ottiene le prestazioni migliori quando testato sul proprio dominio. Tuttavia, emergono osservazioni cruciali sulla robustezza:

- **Interoperabilità Path Tracing:** I modelli addestrati su *Unreal Path Tracer* (colonne PT 64 e 4096) e *Cycles* mostrano un'altissima compatibilità reciproca. Le performance rimangono elevate anche quando testate sui rispettivi dataset (righe), suggerendo che il Path Tracing, indipendentemente dal software (Blender o Unreal), estragga feature geometriche e di illuminazione coerenti.
- **Il caso EEVEE e Deferred:** Il modello addestrato su EEVEE (colonna EEVEE) subisce un calo di performance significativo quando testato sui set Path Tracing (righe Cycles e PT), specialmente nella segmentazione. Questo indica che le approssimazioni della rasterizzazione (come le *screen-space reflections*) creano pattern che la rete fatica a traslare su modelli fisicamente corretti. Al contrario, i modelli Path Tracing si dimostrano più "resilienti", ottenendo risultati accettabili anche se testati su EEVEE.
- **Robustezza del Rumore:** La matrice conferma che la colonna *UE PT 64* (modello addestrato con rumore) produce risultati quasi identici alla colonna *PT 4096*, validando l'ipotesi che la pulizia del segnale sia secondaria rispetto alla correttezza dei volumi e delle ombre per la generalizzazione.

## 5.4 Ottimizzazione mediante Fine-Tuning

L'ultima fase della sperimentazione analizza il contributo del dato sintetico nel paradigma del *Transfer Learning*, valutando come il pre-training su 10.000 campioni sintetici influenzi le performance finali rispetto a un addestramento puramente reale. La sperimentazione è stata articolata su tre scenari principali: il fine-tuning con 12.000 immagini reali (50 epoche), il fine-tuning con un set ridotto di 1.000 immagini reali (300 epoche) e l'addestramento *from scratch* con 10.000 immagini reali utilizzato come baseline.

La Tabella 5.2 riporta i risultati dei modelli pre-addestrati sul sintetico e successivamente rifiniti con 12.000 immagini reali (COCO) per 50 epoche. La riga *COCO reale* rappresenta la baseline dell'addestramento *from scratch* su 10.000 immagini reali per 300 epoche, riportata per facilitare il confronto diretto.

**Tabella 5.2:** Metriche di performance (mAP50-95, mAP50, Precision (P), Recall (R)) dei modelli pre-addestrati su 10.000 immagini sintetiche e rifiniti con 12.000 immagini reali COCO per 50 epoche.

| **Object Detection** | | | | |
|---|---|---|---|---|
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.4411 | 0.6675 | 0.7311 | 0.5986 |
| EEVEE | 0.4426 | 0.6679 | 0.7467 | 0.5895 |
| UE Deferred | 0.4465 | 0.6734 | 0.7263 | 0.6099 |
| UE PT 64 | 0.4434 | 0.6693 | 0.7435 | 0.5879 |
| UE PT 4096 | 0.4410 | 0.6675 | 0.7435 | 0.5904 |
| COCO reale | 0.4282 | 0.6500 | 0.7639 | 0.5962 |
| **Instance Segmentation** | | | | |
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.4238 | 0.6510 | 0.7449 | 0.5654 |
| EEVEE | 0.4220 | 0.6550 | 0.7496 | 0.5717 |
| UE Deferred | 0.4169 | 0.6431 | 0.7265 | 0.5688 |
| UE PT 64 | 0.4246 | 0.6561 | 0.7429 | 0.5804 |
| UE PT 4096 | 0.4224 | 0.6549 | 0.7253 | 0.5812 |
| COCO reale | 0.4361 | 0.6611 | 0.7531 | 0.6057 |

La Tabella 5.3 riporta i risultati dei modelli pre-addestrati sul sintetico e rifiniti con un set ridotto di 1.000 immagini reali (COCO) per 300 epoche. Questo scenario simula la condizione di scarsità di dati reali annotati, valutando la capacità del pre-training sintetico di compensare la limitata disponibilità di campioni del dominio target.

**Tabella 5.3:** Metriche di performance (mAP50-95, mAP50, Precision (P), Recall (R)) dei modelli pre-addestrati su 10.000 immagini sintetiche e rifiniti con 1.000 immagini reali COCO per 300 epoche.

| **Object Detection** | | | | |
|---|---|---|---|---|
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.3272 | 0.5282 | 0.6826 | 0.4907 |
| EEVEE | 0.3325 | 0.5256 | 0.7084 | 0.4643 |
| UE Deferred | 0.3393 | 0.5451 | 0.7341 | 0.4760 |
| UE PT 64 | 0.3252 | 0.5168 | 0.7000 | 0.4624 |
| UE PT 4096 | 0.3332 | 0.5272 | 0.7054 | 0.4794 |
| COCO reale | 0.4282 | 0.6500 | 0.7639 | 0.5962 |
| **Instance Segmentation** | | | | |
| **Motore** | **mAP50-95** | **mAP50** | **P** | **R** |
| Cycles | 0.3310 | 0.5425 | 0.6864 | 0.5023 |
| EEVEE | 0.3350 | 0.5407 | 0.6845 | 0.4931 |
| UE Deferred | 0.3259 | 0.5279 | 0.7205 | 0.4591 |
| UE PT 64 | 0.3340 | 0.5466 | 0.7103 | 0.4868 |
| UE PT 4096 | 0.3283 | 0.5313 | 0.6790 | 0.4878 |
| COCO reale | 0.4361 | 0.6611 | 0.7531 | 0.6057 |

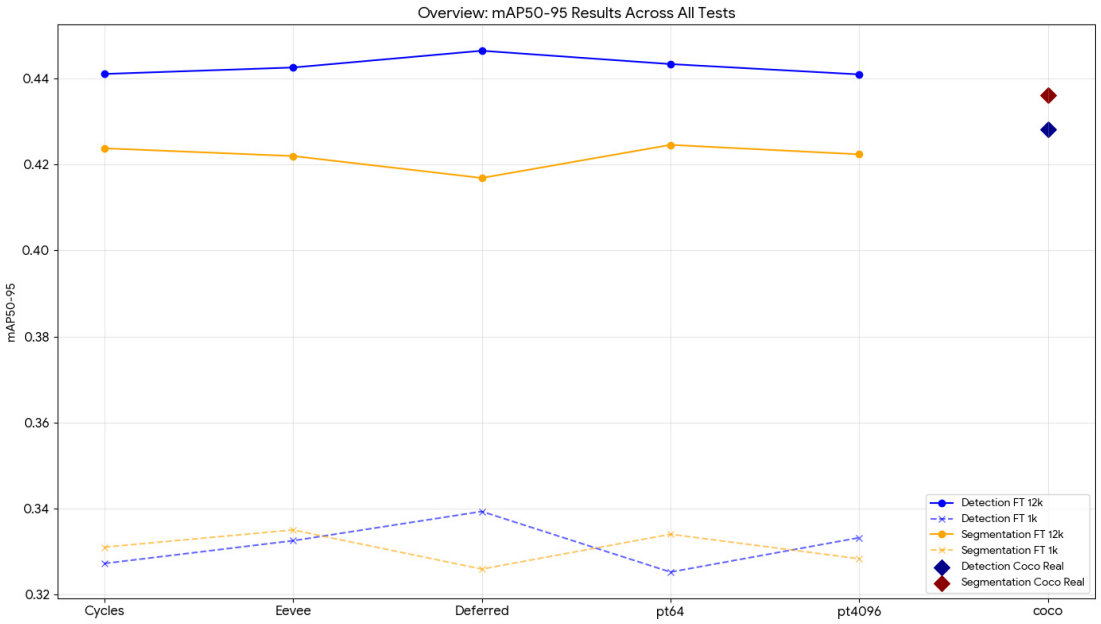



**Figura 5.8:** Confronto della metrica mAP50-95 tra i due scenari di fine-tuning rispetto al baseline reale. La linea continua blu è relativa al fine tuning da 12.000 immagini per la detection, la linea continua gialla è relativa al fine tuning da 12.000 per la segmentazione; la linea tratteggiata blu indica il fine tuning da 1.000 immagini per la detection, mentre la linea tratteggiata gialla indica il fine tuning da 1.000 immagini per la segmentazione. Il punto di dispersione blu rappresenta il risultato dell'allenamento reale from scratch per la detection, mentre quello rosso indica quello per la segmentazione

### 5.4.1 Analisi dei risultati di detection

Il confronto tra i risultati ottenuti nel task di Object Detection (Figura 5.8 - Linee blu) evidenzia l'impatto della quantità di dati reali sulla capacità del modello di adattarsi al dominio target. Come mostrato nella Figura 5.9, il passaggio da 1.000 a 12.000 immagini reali produce un incremento sostanziale della precisione.

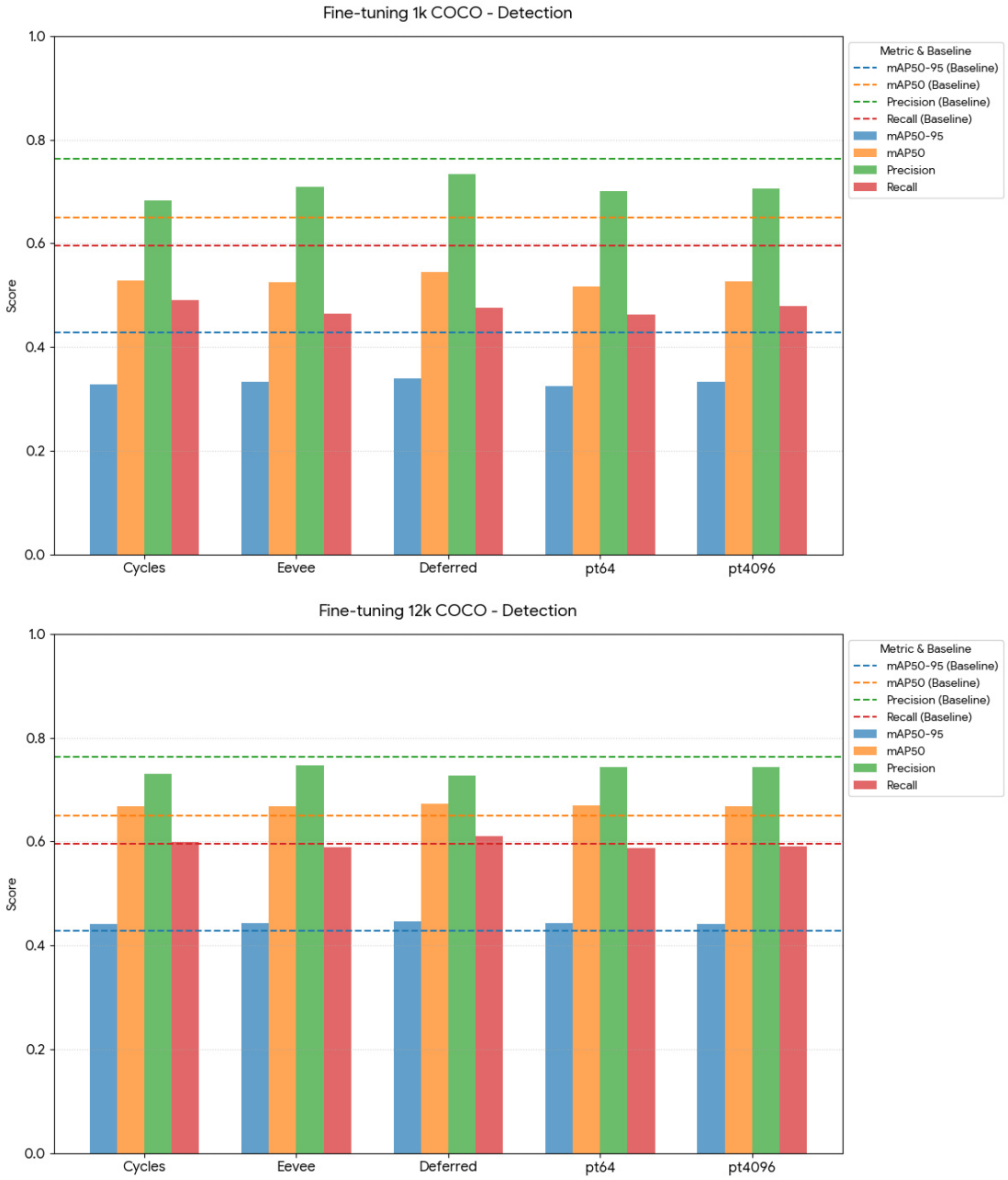



**Figura 5.9:** Confronto dei risultati di Detection tra i due set di immagini reali utilizzati per il fine-tuning. Le linee tratteggiate indicano il riferimento dell'addestramento sul dataset reale da 10.000 immagini from scratch

Un aspetto di interesse, osservabile in Figura 5.9, risiede nel fatto che, per la Detection, la mAP50-95 del modello pre-allenato sul sintetico e rifinito con 12.000 immagini reali supera quella dell'addestramento *from scratch* da 10.000 immagini, nonostante quest'ultimo abbia beneficiato di un numero di epoche sei volte superiore (300 contro 50). Questo risultato dimostra che il pre-training sintetico fornisce una base di conoscenze strutturali (bordi, geometrie, relazioni spaziali) utile per l'adattamento al dominio reale. Estendere questa conclusione a contesti più generali richiederebbe tuttavia una validazione su architetture e domini applicativi differenti.

### 5.4.2 Analisi dei risultati di segmentazione

Parallelamente, i risultati del task di Segmentation (Figura 5.10) confermano una tendenza simile per quanto riguarda il volume dei dati, pur evidenziando una differenza qualitativa nel Transfer Learning.

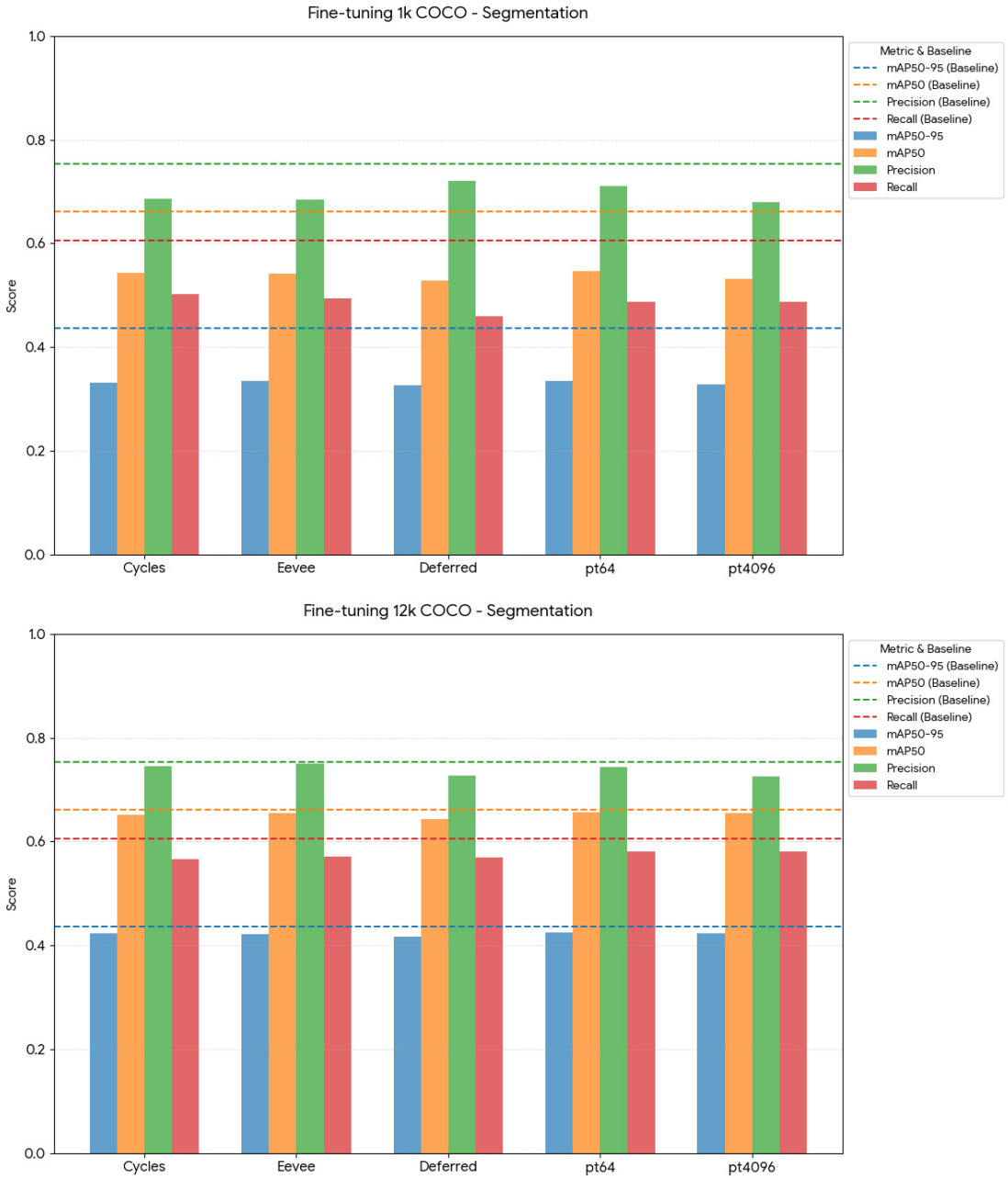



**Figura 5.10:** Confronto dei risultati di Segmentation tra i due set di immagini reali utilizzati per il fine-tuning. Le linee tratteggiate indicano il riferimento dell'addestramento sul dataset reale da 10.000 immagini from scratch

A differenza della detection, nella segmentazione il risultato del fine-tuning da

12.000 immagini rimane, in tutte le metriche, leggermente inferiore rispetto all'addestramento *from scratch* reale. Questa discrepanza suggerisce che la segmentazione a livello di pixel sia più sensibile al *domain shift*: la precisione millimetrica richiesta per le maschere risente maggiormente delle differenze di texture tra sintetico e reale, rendendo il dato reale puro leggermente più efficace per questo specifico task rispetto alla combinazione ibrida.

### 5.4.3 Discussione sul volume dei dati e salto prestazionale

Il confronto tra il fine-tuning a 1.000 immagini e quello a 12.000 immagini (Figure 5.9 e 5.10) porta a una conclusione cruciale: in questo esperimento, il valore aggiunto sembra essere dato principalmente dalla quantità dei dati reali, piuttosto che dal numero di epoche di addestramento. Anche estendendo l'addestramento del set da 1.000 immagini a 300 epoche, le prestazioni rimangono notevolmente inferiori rispetto al set da 12.000 fermato a sole 50 epoche. Questo suggerisce che, almeno nel contesto osservato, un aumento del tempo di calcolo non sia sufficiente a compensare la mancanza di varianza statistica.

È tuttavia rilevante osservare la capacità di adattamento del modello: l'introduzione di sole 1.000 immagini reali è stata sufficiente per **triplicare le performance** rispetto al solo allenamento sintetico (che presentava un $mAP$ estremamente basso, tra 0.08 e 0.12 - Vedi Tabella 5.1). Tale balzo prestazionale evidenzia come il pre-training sintetico, pur non essendo autosufficiente, crei una struttura pronta per essere "ancorata" al mondo reale con uno sforzo di campionamento minimo, validando l'approccio sintetico come strategia di *data augmentation* e *feature learning* preliminare.

## 5.5 Conclusioni sul trade-off metodologico e sviluppi futuri

L'analisi comparativa condotta in questo capitolo mette in luce, nel contesto sperimentale adottato, un equilibrio tra fedeltà visiva e scalabilità produttiva. Sebbene i motori basati su *Path Tracing* ad alto campionamento producano dati qualitativamente superiori, il costo computazionale richiesto rappresenta un fattore limitante rispetto al volume di dati generabile nello stesso arco temporale.

I risultati ottenuti suggeriscono che, per il task di detection e segmentazione di veicoli in ambiente urbano sintetico, potrebbe essere preferibile adottare una pipeline capace di generare una vasta gamma di situazioni in tempi brevi, piuttosto che concentrare il budget temporale su un numero esiguo di frame ad altissima fedeltà. In questo contesto specifico, il binomio **Unreal Engine + Path Tracer (Low Sample)** ha mostrato un buon compromesso tra correttezza fisica e rapidità

di generazione. È tuttavia opportuno sottolineare che queste indicazioni emergono da un setting ridotto (una singola scena, una sola classe di oggetti, un unico modello) e non possono essere estese acriticamente a domini applicativi differenti. Risultati analoghi sono riportati in letteratura. Nel contesto della classificazione di immagini, è stato osservato che l'integrazione di immagini sintetiche oltre una certa soglia riduce significativamente l'overfitting, permettendo di raggiungere performance competitive con una frazione dei dati reali normalmente richiesti [26]. Analogamente, nel controllo qualità industriale, un approccio ibrido sintetico-reale ha dimostrato robustezza superiore rispetto al solo dato sintetico in presenza di condizioni critiche come riflessi e sfondi complessi [27]. Questi riscontri supportano, in contesti diversi, la direzione metodologica adottata in questo lavoro.

L'efficacia di questo approccio è parzialmente supportata anche dai risultati del *fine-tuning*: l'introduzione di 1.000 immagini reali ha prodotto un incremento sostanziale delle performance rispetto al solo addestramento sintetico, suggerendo che il dato virtuale possa fornire una base strutturale utile per il successivo adattamento al dominio reale. Anche in questo caso, la generalizzazione di tale osservazione richiederebbe esperimenti su dataset reali più ampi e su altri domini applicativi.

### 5.5.1 Sviluppi futuri

Alla luce delle evidenze emerse, gli sviluppi futuri di questa ricerca dovrebbero focalizzarsi sul potenziamento della **varietà statistica** del dataset, elemento dimostratosi più critico della fedeltà visiva del singolo pixel.

- **Diversificazione degli scenari:** Data la complessità nel gestire simulazioni di traffico dinamico inter-piattaforma, una strategia promettente consiste nella creazione di ambienti vasti e diversificati con traffico statico posizionato manualmente. Questo permetterebbe di replicare in modo controllato una moltitudine di situazioni stradali critiche, massimizzando il valore informativo di ogni frame prodotto.
- **Espansione dei domini ambientali:** I risultati hanno mostrato una parziale difficoltà del modello sintetico nel generalizzare su immagini reali caratterizzate da contesti naturali. Risulta quindi necessario estendere la modellazione oltre l'ambiente prettamente urbano, includendo scenari extraurbani, autostradali e naturali (foreste, parchi), aumentando drasticamente la densità di elementi organici come gli alberi, molto frequenti nel dataset COCO.
- **Ampliamento delle condizioni atmosferiche e di luce:** Sebbene la sperimentazione attuale abbia coperto diverse configurazioni (soleggiato, nuvoloso, tramonto), è prioritario estendere il dataset a condizioni critiche come la

**pioggia** e la **notte**. La pioggia introduce sfide legate alla rifrazione sulle lenti e alla riflettenza dell'asfalto bagnato, mentre lo scenario notturno testa la capacità del modello di operare con sorgenti di luce artificiale puntiformi e forti contrasti.

- **Varietà del parco veicoli:** Un'estensione della libreria di asset 3D, includendo un numero maggiore di tipologie, modelli e varianti cromatiche di veicoli, ridurrebbe ulteriormente il *Sim-to-Real gap*, portando il modello a una comprensione più universale e meno legata alle specifiche geometrie utilizzate in questa sperimentazione.
- **Simulazione di degradazioni ottiche:** Un filone di ricerca prioritario riguarda l'implementazione sistematica di effetti post-processing volti a replicare i limiti delle camere reali. L'introduzione di *motion blur* (per simulare la velocità), sfocature di profondità (*depth of field*) e rumore del sensore renderebbe i dati sintetici meno "perfetti" e più simili alle acquisizioni rumorose del mondo reale, migliorando la robustezza del modello nelle condizioni di inferenza più difficili.

In definitiva, il futuro della generazione di dati sintetici risiede in una modellazione procedurale e strategica della scena, dove la ricchezza del contenuto e la diversità ambientale prevalgono sulla ricerca della perfezione fotografica fine a se stessa.

# Bibliografia


[1] Ross Girshick, Jeff Donahue, Trevor Darrell e Jitendra Malik. *Rich feature hierarchies for accurate object detection and semantic segmentation*. arXiv:1311.2524. 2014 (cit. alle pp. 1, 5, 6).

[2] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. arXiv:1405.0312. 2015 (cit. alle pp. 1, 5, 9).

[3] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth e Bernt Schiele. *The Cityscapes Dataset for Semantic Urban Scene Understanding*. arXiv:1604.01685. 2016 (cit. alle pp. 1, 9).

[4] Ultralytics. *Ultralytics YOLO26*. 2026. URL: `https://docs.ultralytics.com/it/models/yolo26/` (cit. alle pp. 2, 27–29).

[5] Yohann Cabon, Naila Murray e Martin Humenberger. *Virtual KITTI 2*. arXiv:2001.10773. 2020 (cit. alle pp. 2, 11).

[6] Maximilian Denninger et al. *BlenderProc: Reducing the Reality Gap with Photorealistic Rendering*. 2020 (cit. alle pp. 2, 12, 13, 17, 22).

[7] Shaoqing Ren, Kaiming He, Ross Girshick e Jian Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. arXiv:1506.01497. 2016 (cit. alle pp. 5–7).

[8] Joseph Redmon, Santosh Divvala, Ross Girshick e Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. arXiv:1506.02640. 2016 (cit. alle pp. 5, 7, 8).

[9] Stephan R. Richter, Vibhav Vineet, Stefan Roth e Vladlen Koltun. *Playing for Data: Ground Truth from Computer Games*. arXiv:1608.02192. 2016 (cit. a p. 10).

[10] Magnus Wrenninge e Jonas Unger. *Synscapes: A Photorealistic Synthetic Dataset for Street Scene Parsing*. arXiv:1810.08705. 2018 (cit. a p. 12).

[11] Shayne Longpre et al. *The Data Provenance Initiative: A Large Scale Audit of Dataset Licensing & Attribution in AI*. 2023 (cit. a p. 14).

[12] European Union. *Regulation (EU) 2016/679 of the European Parliament and of the Council on the protection of natural persons with regard to the processing of personal data and on the free movement of such data (General Data Protection Regulation)*. Official Journal of the European Union, L 119/1. Accessed: 2024-02-12. 2016 (cit. a p. 14).

[13] Erling Hukkelas e Lindigo Frank. «Does Image Anonymization Impact Computer Vision Training?» In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. IEEE. 2023, pp. 4276–4285 (cit. a p. 14).

[14] Amir Rasouli, Iuliia Kotseruba e John K. Tsotsos. «PIE: A Large-Scale Dataset and Models for Pedestrian Intention Estimation and Trajectory Prediction». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE. 2019, pp. 6262–6271 (cit. a p. 15).

[15] Sergey I. Nikolenko. *Synthetic Data for Deep Learning*. Cham: Springer International Publishing, 2021. DOI: 10.1007/978-3-030-75178-4 (cit. a p. 15).

[16] Blender Foundation. *Blender 4.2 Reference Manual*. Accessed: 2024-05-20. 2024. URL: https://docs.blender.org/manual/ (cit. alle pp. 17–20).

[17] Brecht Van Lommel. «The Cycles Shader System». In: *Proceedings of the 2013 SIGGRAPH Talks*. ACM. 2013 (cit. a p. 19).

[18] Matt Pharr, Wenzel Jakob e Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. 3rd. Morgan Kaufmann, 2016 (cit. alle pp. 19, 25).

[19] SuperHive Market. *The City Generator - Product Page*. Accessed: 2024-05-20. 2024. URL: https://superhivemarket.com/products/the-city-generator (cit. alle pp. 20–22).

[20] Blender Add-ons. «The City Generator Add-on Review». In: (2023). Accessed: 2024-05-20. URL: https://blender-addons.org/the-city-generator/ (cit. a p. 20).

[21] Cinematic Cookie. *Using The City Generator 2.0 in Blender | Tutorial*. YouTube. Accessed: 2024-05-20. 2024. URL: https://www.youtube.com/watch?v=kRHkGoTQKM8 (cit. a p. 21).

[22] DLR-RM. *BlenderProc2 Documentation*. GitHub. 2024. URL: https://github.com/DLR-RM/BlenderProc (cit. alle pp. 22–24).

[23] Epic Games. *Rendering High Quality Frames with Movie Render Queue*. Unreal Engine 5.5 Documentation, Accessed: 2025-01-22. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/rendering-high-quality-frames-with-movie-render-queue-in-unreal-engine?application_version=5.5 (cit. alle pp. 25, 26).

[24] Epic Games. *Path Tracer in Unreal Engine*. Unreal Engine 5.5 Documentation, Accessed: 2025-01-22. 2025. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/path-tracer-in-unreal-engine?application_version=5.5 (cit. alle pp. 25, 26, 62).

[25] Cinematic Cookie. *How to Export The City Generator | Blender to Unreal Engine*. YouTube. Accessed: 2025-01-05. 2025. URL: https://www.youtube.com/watch?v=lTaw9ZpWtBk (cit. a p. 42).

[26] John W. Belanger Smutny e Creed Jones. *The Effects of using created Synthetic images in Computer Vision training*. arXiv:2506.03449 (cit. a p. 91).

[27] Dmytro Bondar e Yevheniia Basova. *SYNTHETIC-TO-REAL DOMAIN ADAPTATION IN COMPUTER VISION SYSTEMS: TOWARDS HIGH-PRECISION INDUSTRIAL APPLICATIONS*. International Journal of Mechatronics and Applied Mechanics 2025, Issue 21, Vol. 1 (cit. a p. 91).