UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Applicazione del Deep Learning per l'approssimazione del Ray Tracing

Application of Deep Learning for Ray Tracing approximation

Relatore:
Prof. Primo ZINGARETTI

Candidato:
Roberto BROCCOLETTI

Anno Accademico 2020-2021

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

A mio padre, so che ci sei sempre

Sommario

In questa tesi è esposto un nuovo approccio per il miglioramento del rendering, gettando le basi per una serie di approfondimenti futuri sull'argomento. Il lavoro si fonda sull'idea che un render calcolato con una tecnica di illuminazione locale e lo stesso render calcolato con una tecnica di illuminazione globale (Ray Tracing principalmente), usando la stessa equazione di illuminazione, siano vicini, in molti casi, in termini di Image Quality. Questa quindi si può andare a migliorare utilizzando delle informazioni che descrivono la geometria della parte di scena inquadrata (G-Buffer), generando una differenza (Delta), che sommato ad un render preliminare (quello con illuminazione locale), permetta di ottenere un miglioramento visivo dell'immagine. Per realizzare tutto questo si presenta una CNN coadiuvata da un Discriminatore Percettivo, seguendo uno schema GAN. Inoltre si propone anche un set di strumenti creati appositamente per la realizzazione del progetto, che permettono di costruire un dataset che possa essere usato per fare ricerca sull'argomento toccato in questa sede.

Indice

1. Introduzione **1**
- 1.1. Contesto . . . 1
- 1.2. Motivazioni . . . 1
- 1.3. Obiettivi . . . 3
- 1.4. Struttura della Tesi . . . 3

2. Stato dell'arte **4**

3. Dati e Metodo **8**
- 3.1. Dataset . . . 8
  - 3.1.1. Dati Necessari . . . 8
  - 3.1.2. Tools . . . 9
  - 3.1.3. Framework Creazione Dataset . . . 24
- 3.2. Metodologia . . . 26
  - 3.2.1. Panoramica . . . 26
  - 3.2.2. DeltaNet . . . 28
  - 3.2.3. Impostazione della Loss . . . 30
  - 3.2.4. Discriminatore Percettivo . . . 30
  - 3.2.5. Dettagli su Implementazione e Allenamento . . . 32

4. Risultati e Commenti **33**
- 4.1. Casi Normali . . . 33
- 4.2. Caso Favorevole . . . 42
- 4.3. Casi Particolari . . . 44

5. Conclusioni e Sviluppi Futuri **50**
- 5.1. Conclusioni . . . 50
- 5.2. Sviluppi Futuri . . . 50

A. Appendice **54**
- A.1. Computer Graphics . . . 54
  - A.1.1. OpenGL . . . 54
  - A.1.2. Cubemap . . . 54
  - A.1.3. Illuminazione locale vs globale . . . 54
  - A.1.4. Physically Based Rendering . . . 55
  - A.1.5. Deferred Shading vs Forward Shading . . . 58

A.2. Blender . . . 61
    A.2.1. Python API . . . 61
    A.2.2. Eevee vs Cycles . . . 61
A.3. HSV . . . 62
A.4. Deep Learning . . . 63
    A.4.1. Squeeze and Excitation . . . 63
    A.4.2. MBConv . . . 63
A.5. Formato Wavefront .OBJ . . . 65

Elenco delle figure

3.1. Esempio di dato 9
3.2. Panoramica su Structured3D 10
3.3. Panoramica su InteriorNet 10
3.4. Codice Modificato 12
3.5. Codice Originale 12
3.6. Vertex Shader 14
3.7. Fragment Shader 15
3.8. Modalità di RandCameras 16
3.9. Esempio di Generazione e Posizionamento dei punti di controllo 17
3.10. Esempio Pre e Post Generazione 18
3.11. Dettaglio del pannello sugli angoli 18
3.12. Esempio di distribuzione controllata delle camere nello spazio 19
3.13. Esempio di file JSON contenente le camere 19
3.14. Esempio di utilizzo della Modalità rendering 20
3.15. Implementazione di getRandomPointFromSpace 21
3.16. Implementazione di createCam 22
3.17. Implementazione di exportCamerasConfig 23
3.18. Implementazione di addRandomCam 23
3.19. Workflow per la creazione del dataset 24
3.20. Esempio di file CSV ottenibile con lo script 26
3.21. Differenze tra illuminazione locale e globale 27
3.22. Schema del Generatore 29
3.23. Schema del Discriminatore 31

4.1. Scena 1 33
4.2. Scena 2 34
4.3. Scena 2 dettaglio dentro la vasca 35
4.4. Scena 2 dettaglio sul muro 35
4.5. Scena 4 36
4.6. Scena 5 37
4.7. Scena 5 dettaglio soffitto 37
4.8. Scena 6 38
4.9. Scena 7 38
4.10. Scena 7 dettaglio nuvole 39
4.11. Scena 7 dettaglio uccelli 39

4.12. Scena 8 40
4.13. Scena 9 41
4.14. Esempi dell'artefatto 41
4.15. Scena 3 42
4.16. Altri esempi della scena 3 43
4.17. Scena 9 dettaglio 44
4.18. Scena 11 dettaglio 45
4.19. Caso particolare scena 9 46
4.20. Caso particolare 1 scena 11 47
4.21. Caso particolare 2 scena 11 48
4.22. Tabella riepilogativa sui risultati 49

5.1. Funzione identità tramite Enc-Dec 51
5.2. Schema modello con Enc-Dec 52
5.3. Schema riassuntivo Stable View Synthesis. 52
5.4. Allenamento Ciclico per la funzione identità Enc-Dec 53

A.1. Modello Microfacet. 55
A.2. Esempio dell'influenza della Roughness sul materiale. 56
A.3. Esempio di BRDF. 58
A.4. Schema esemplificativo di una moderna pipeline di rendering. 58
A.5. Forward Rendering. 59
A.6. Deferred Rendering. 59
A.7. Cilindro HSV. 62
A.8. Schema dello Squeeze and Excitation 63
A.9. Configurazione MBConv6. 64
A.10.Esempio di file OBJ. 65
A.11.Esempio di file MTL. 65

Capitolo 1.

Introduzione

1.1. Contesto

Negli ultimi 25 anni il mondo della **Computer Graphics** si è evoluto enormemente, spinto dai vari media che si sono avvalsi di questa tecnologia per poter creare mondi e situazioni che vanno al di fuori della realtà. Infatti come è possibile riscontrare nei film e spesso nelle serie TV, il grado di realismo delle scene calcolate al computer ha fatto un salto in avanti immenso anche in produzioni minori. In questo contesto storico-culturale è impossibile non prendere atto della sempre più diffusione dei videogiochi e della loro importanza sia a livello culturale che economico. Proprio questi ultimi hanno guidato lo sviluppo della CG, soprattutto in ambito real-time, cioè per applicazioni interattive, arrivando ad un livello di fedeltà che era impensabile fino a qualche anno fa.

1.2. Motivazioni

Con l'esplosione del mercato dei videogiochi, e in generale l'utilizzo sempre più preponderante della CG nelle applicazioni, sono nate nuove problematiche, legate al voler portare su più dispositivi possibili le applicazioni grafiche. Questo per un motivo molto semplice, maggiore è il numero di dispositivi che possono eseguire correttamente un programma, potenzialmente maggiori sono le vendite. Nonostante la grafica interattiva sia sempre più visivamente accattivante, non si sono raggiunti ancora, in termini di fedeltà grafica, i rendering effettuati con tecniche offline, quindi con un budget di tempo virtualmente illimitato, che impiegano anche minuti per essere calcolati, al contrario dei 33.3 ms al massimo, necessari per avere un'immagine fluida all'occhio umano. Quindi il primo problema consiste nel non riuscire a raggiungere ancora una fedeltà grafica che si può definire fotorealistica. Rispettando determinate condizioni si possono raggiungere risultati fotorealistici, queste sono:

- avere a disposizione un hardware performante per ottenere maggiore fluidità e resa grafica
- utilizzare delle tecniche specifiche in base al contesto che si vuole rappresentare, e quindi uno sforzo in termini di programmazione, piuttosto elevato

Questa necessità di maggiore potenza però si scontra con la volontà di cercare di coprire quanti più utenti, di conseguenza dispositivi, possibili. Perciò si stanno vagliando diverse soluzioni a questo problema. La prima è il Cloud, cioè l'applicazione viene eseguita da un computer remoto e poi il risultato dell'input dell'utente viene inviato, attraverso internet, per la visualizzazione sullo schermo. Purtroppo ancora sta faticando a imporsi, come tecnica, per via delle connessioni a Internet non ottimali in alcune aree geografiche , visto che tale soluzione richiede una velocità di trasferimento dati elevata. per trasferire le immagini dal server al client e sopratutto per via della latenza che si introduce nei comandi[1], perchè questa incide molto sull'esperienza dell'utente dato che fa sembrare il contenuto più o meno reattivo in relazione al suo input. La seconda soluzione consiste nell'utilizzo delle **Reti Neurali**,come si vedrà nel capitolo 2, con lo scopo di migliorare la resa grafica o di abbassare il tempo di calcolo dei render. Un esempio pratico di questo approccio è la tecnologia DLSS di nVidia, che sta per Deep Learning Super Sampling, che server per aumentare la dimensione di un'immagine senza perdere qualità. Quindi si può calcolare un render ad una dimensione inferiore (ad esempio 1920x1080), ed ottenere con questa tecnica una immagine che sia equivalente ad una nativa ad una dimensione maggiore (ad esempio 3840x2160). Questo perchè il tempo necessario per fare il rendering dipende da molti fattori, tra cui:

- dimensione del render in pixel
- complessità poligonale della scena
- tecniche di shading dipendenti dal numero di pixel (es. Ray-Tracing)
- effetti applicati sull'immagine

Quindi come si può intuire il tempo necessario per calcolare il render è piuttosto variabile, ciò crea dei problemi, a volte, nel riuscire ad avere un numero di frame per secondo che sia costante[2]. In questo le tecniche legate alle Reti Neurali danno una mano, perchè tali reti possono essere allenate per riuscire a realizzare dei task e una volta complete vengono eseguite in un tempo che è pressochè identico di volta in volta. Oltre a questo non si può non considerare che sempre più dispositivi stanno integrando processori in grado di accelerare i calcoli necessari per questo tipo di elaborazioni, quindi si lascia ancora più "spazio" sulle GPU per poter calcolare i render che fanno da input a queste reti. Infine l'ultima considerazione a riguardo che è possibile fare è che avere dei processi di elaborazione più leggeri complessivamente (grazie a queste tecniche), significa anche riuscire ad avere una maggiore autonomia sui dispositivi portatili e quindi avere un risparmio di energia (che fa comodo ad esempio a chi gestisce i servizi Cloud e ha tanti server che calcolano continuamente immagini). Infine si può dare una spiegazione alla preferenza di utilizzo delle reti neurali rispetto

[1] cioè il tempo che intercorre per ottenere una risposta a seguito di un input dell'utente

[2] è importante perchè dona una migliore percezione di fluidità all'utente.

alle altre tecnologie. Questo tipo di approcci, permettono di raggiungere risultati che superano, anche di molto, quelli ottenibili attraverso degli algoritmi classici. Inoltre permettono di approcciare anche problemi per cui non è possibile creare algoritmi direttamente. Hanno una grande capacità di generalizzazione anche per istanze del problema inusuali.

1.3. Obiettivi

A fronte di quello detto subito sopra, il lavoro di questa tesi si pone come un lavoro di ricerca che mira a trovare l'architettura di una rete neurale che possa realizzare il task di migliorare la qualità dei render, cercando di colmare il gap tra immagini semplici, calcolate con tecniche real-time, e immagini complesse, calcolate con tecniche offline. Senza però avere la pretesa di realizzare un sistema completo e pronto per la produzione, quanto più un proof of concept per dimostrare l'efficacia e la realizzabilità di un determinato metodo, individuando punti di forza e limiti di tale approccio.

1.4. Struttura della Tesi

Questa tesi esporrà il lavoro fatto nel seguente modo:

- 2 si concentrerà su quello che riguarda lo stato dell'arte relativo al task che è stato affrontato
- 3 riguarderà tutto quello che è stato effettivamente realizzato
- 4 esaminerà i risultati ottenuti
- il capitolo 5 darà una panoramica finale sul lavoro e quali sono le possibili direzioni sulle quali concentrarsi per migliorare i risultati

Capitolo 2.

Stato dell'arte

Lo scopo di migliorare un il risultato del rendering può avere molteplici forme. La più immediata è l'ottenimento di una immagine più fotorealistica, rispetto all'immagine che si genererebbe altrimenti. Questo molto spesso è lo scopo nei media più commerciali e destinati alla fruizione finale da parte di una persona qualsiasi. Mentre non sempre si lavora con scene 3D destinate all'intrattenimento, spesso si tratta anche di altri tipi di forme tridimensionali, la cui visualizzazione su uno schermo è destinata a specialisti (es. medici). In questo caso i miglioramenti sono legati a delle difficoltà nel calcolo delle immagini in real-time, come algoritmi troppo onerosi o immagini troppo grandi. Partendo dal fotorealismo, per ottenerlo si può andare ad utilizzare una simulazione più o meno approssimativa di quello che è il reale processo fisico nella generazione di un'immagine. Fare ciò non è sempre possibile a causa della complessità di tali calcoli e farlo in tempo reale diventa difficile, per non contare lo sforzo nel dover replicare correttamente oggetti e i relativi materiali. Alcuni lavori hanno mirato ad apprendere direttamente il processo di sintesi delle immagini partendo da informazioni semantiche riguardanti le varie parti dell'immagine[1][2][3][4][5][6][7]. Queste informazioni semantiche riguardano principalmente la forma e il tipo di oggetto da generare nell'immagine. Così fatta però questa generazione risulta piuttosto ambigua nelle sue parti, e complessivamente porta alla comparsa di visibili artefatti, inconsistenza temporale. Inoltre per realizzare queste informazioni semantiche è necessario un processo di etichettatura dei dati reali, di difficile realizzazione se si considera la mole di dati necessaria. Per questo lavoro, l'approccio consiste nell'andare a migliorare un render già esistente, attraverso alcune informazioni geometriche aggiuntive, senza la necessità di avere delle informazioni semantiche che richiedono l'annotazione manuale. Poi ci sono le tecniche Image-based, che generano immagini di una scena da nuovi punti di vista partendo da un set di immagini da alcuni punti di vista prefissati della stessa scena [8][9][10][11][12][13][14][15][16]. Anche in questo caso, il lavoro da fare risulta dispendioso per la realizzazione, perchè richiede di scattare una serie di foto della scena, che poi in seguito non si riesce a fare, anche a causa di possibili cambiamenti nell'illuminazione della scena e nella composizione della stessa. Inoltre bisogna evitare di calcolare immagini da punti di vista troppo differenti dal set a disposizione, perchè ciò causa artefatti. Altri approcci hanno proposto la combinazione di tecniche tradizionali di rendering con tecniche basate

sui dati (come quelle spiegate subito sopra).Johnson et al. hanno proposto di migliorare il realismo delle immagini renderizzate attraverso il trasferimento di patch sovrapponibili, da fotografie strutturate in maniera simile[17]. Liao et al. vanno a recuperare le patch tramite il nearest neighbor in uno spazio di feature che viene appreso [18]. Reinhard et al. crea un match sulla distribuzione dei colori su di una immagine sorgente per farla corrispondere a quella di una immagine di riferimento, [19] e ha mostrato che questo può aumentare il realismo delle scene renderizzate. Esempi di questi approcci sono tecniche sempre più sofisticate di trasferimento dello stile da un'immagine ad un'altra (ad esempio una foto che viene trasformata come se fosse stata dipinta da uno specifico pittore) [20] [21] [22] [23][24][25][26]. I lavori più recenti si sono concentrati su strutture encoder/decoder, andando a fondere il contenuto dell'immagine di input con lo stile di un'immagine obiettivo [27] [28] . Spesso questi approcci sono molto dipendenti dall'utilizzo di una immagine obiettivo che abbia una composizione molto simile all'immagine di input, questo perchè le differenze di contenuto o composizione tra le due immagini peggiorano la qualità del risultato finale. Grazie all'approccio proposto in questa tesi, il problema non si pone, dato che l'immagine di input viene accoppiata con un ground-truth che è identico al render di input con la sola differenza che la scena viene calcolata con tecniche differenti di rendering. In questo caso si può classificare il task come **image-to-image traslation**, in cui la rete trasforma il render da un dominio in cui le immagini sono calcolate con tecniche real-time di illuminazione locale, ad un dominio in cui i render vengono creati con tecniche offline di illuminazione globale. Una specifica declinazione di questo task, simile a quello affrontato in questa tesi, è unpaired image-to-image traslation . Questo significa che per ogni immagine di input non è possibile ottenere direttamente un'immagine che possa servire da ground truth per l'output del modello. In questo caso si è lavorato molto nel migliorare la consistenza tra le immagini di input e di output, attraverso vari approcci tutti tendenti ad introdurre una serie di vincoli aggiuntivi al fine di "indirizzare" meglio il modello in questo complesso compito [29] [30] [31] [32].Una cosa comune alla maggior parte di questi metodi è l'utilizzo di **adversarial objectives**, cioè l'utilizzo di un discriminatore. Questo discriminatore generalmente è una rete che valuta il realismo dei risultati del modello che utilizziamo come generatore, quindi si va a creare un meccanismo tale per cui queste due reti vengono allenate parallelamente. Come spiegato nel paper originale [33], il generatore mira ad ingannare il discriminatore cercando di generare immagini sempre più realistiche nel dominio di output, mentre il discriminatore si allena per riconoscere quando una immagine viene generata oppure è "vera". Quindi la logica dietro tutto questo framework è quella di riuscire ad avere una supervisione di alta qualità al modello che effettua il compito (in questo caso la traslazione da un dominio di illuminazione locale ad un dominio di illuminazione globale). Nei vari lavori esistenti è risultato anche che un discriminatore che va a classificare un'immagine come vera (non generata) o falsa (generata da un'altra rete), tende a concentrarsi su informazioni di basso livello (es. texture nell'immagine), cioè

informazioni legate molto più all'aspetto finale dell'immagine quanto al contenuto semantico della stessa. Per ovviare questa limitazione a fronte di task più ad alto livello, sono state proposte diverse soluzioni tra cui utilizzare mappe di segmentazione semantica oppure spingere il discriminatore a comportarsi come un critico, quindi non dando una classificazione binaria, bensì assegnando uno score di realismo all'immagine generata. In questo caso in fase di allenamento del discriminatore bisogna andare a separare quanto più possibile gli score assegnati a immagini "vere" dagli score assegnati alle immagini "false". Prendendo ispirazione dal lavoro di Richter et al. [34] il discriminatore non prende in input direttamente l'immagine generata, bensì le feature map di una vgg16 preallenata, creando un mini discriminatore per ogni layer di attivazione della VGG in modo tale da avere un discriminatore per ogni livello di astrazione dell'informazione all'interno dell'encoding che effettua la VGG. Tutto ciò è necessario soprattutto perchè il discriminatore deve imparare a cogliere le differenze tra i due render in termini anche di image quality. Questo concetto è estremamente difficile da catturare all'interno di una funzione obiettivo da far ottimizzare ad una rete, dato che sta ad indicare la qualità visiva che una persona percepisce in relazione ad un'immagine. Un render può essere più o meno realistico anche a seconda di chi lo guarda, quindi il discriminatore deve riuscire a cogliere questo concetto aiutandosi con il ground truth che può utilizzare per definire cosa appare realistico. L'approccio si basa su un metodo ibrido che usa come input della rete i G-Buffers, che descrivono puntualmente ogni pixel del render in termini di colore di base, geometria ed altre informazioni necessarie per effettuare lo shading della scena da un determinato punto di vista. Nalbach et al. [35] ha già dimostrato in passato come sia possibile lo shading di una pipeline di rendering convenzionale attraverso delle CNN. In seguito AlHaija et al. [36] hanno migliorato questo approccio aggiungendo l'adversarial loss spiegata sopra. Similmente a quello che è stato realizzato in questa tesi, Bi et al. [17] hanno sviluppato una pipeline che rende più realistici render di bassa qualità di scene indoor. Come in questo lavoro, il loro approccio richiede la coppia di immagini (low-quality,high-quality) render. In generale però per questo approccio non ci si è limitati ad utilizzare solo scene indoor bensì anche di altre tipologie. Inoltre lo scopo del loro metodo è di rendere più realistiche le immagini, dividendo in due stage il procedimento. Con la prima fase passano da un render di bassa qualità, non calcolato con tecniche di **Physically Based Rendering** (PBR), ad uno equivalente in PBR, dopodichè nella seconda parte lo trasformano in modo da farlo assomigliare quanto più possibile alla realtà. Le differenze cruciali con il lavoro proposto in questa tesi sono:

- Come dati di input utilizzano solamente le normali, l'albedo e uno shading preliminare non PBR.
- il loro scopo è quello di ottenere una immagine più vicina possibile alla realtà, usando immagini di scene vere per allenare il modello nella seconda fase.

- differisce completamente la struttura della rete.

Capitolo 3.

Dati e Metodo

Questo capitolo è composto da due parti. La prima è relativa al Dataset 3.1, ed illustra tutti gli strumenti creati per la sua realizzazione con un focus sulla logica di implementazione di questi ultimi. Inoltre viene mostrato un flusso di lavoro che descrive tutto il procedimento per poter costruire un set di dati conformi alle possibili caratteristiche richieste dalla ricerca nella Computer Graphics. La seconda parte relativa alla Metodologia 3.2, invece si concentra sull'idea alla base dell'approccio proposto, come questa si traduce effettivamente in un metodo per realizzare il miglioramento del rendering. Poi si parla della struttura della rete convoluzionale utilizzata, del suo allenamento e infine quali accortezze sono state necessarie per poter trovare una corretta funzione obiettivo da minimizzare.

3.1. Dataset

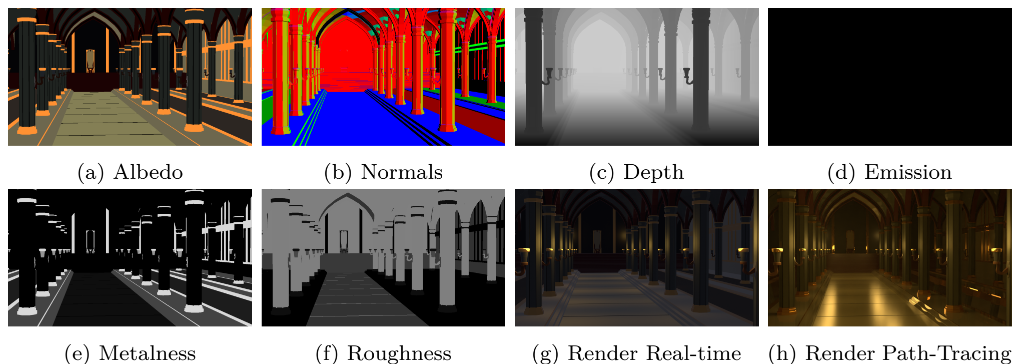
3.1.1. Dati Necessari

La prima necessità, quando si creano dei metodi Data-Driven (come nel caso delle Reti Neurali), è trovare i dati. Gli approcci basati su Deep Learning dipendono molto da essi, dalla loro qualità e quantità. Se si hanno molti dati diversi, la rete potrà essere addestrata per risultare in grado di generalizzare meglio. Inoltre dati che hanno all'interno errori o altre possibili inesattezze, rischiano di abbassare le probabilità di riuscita di un allenamento, perchè lo si fa considerando anche questo dato totalmente o parzialmente sbagliato. Detto questo si può capire come sia cruciale la fase di recupero dei dati e creazione del dataset,che in questo caso è composto dai G-Buffer, un render calcolato con un motore di rendering Real-Time e un render calcolato con un motore di rendering che fa Path-Tracing[1]. Il primo passo è stato trovare delle scene tridimensionali gratuite che fossero sufficienti in numero, complessità e diversità. È stata utilizzata una pipeline di rendering PBR, quindi le scene dovevano avere anche i materiali corretti con le caratteristiche fisiche necessarie (Roughness, Metalness ecc.). [2] Per ogni scena, ci sono una serie di punti di vista e per ognuno di questi vengono calcolati:

[1]Le informazioni di approfondimento sono presenti in Appendice.
[2]le scene sono state recuperate gratuitamente dal sito SketchFab.

- Albedo: colore di base di un materiale
- Emission: colore della luce che emette un materiale
- Roughness: il livello di "rugosità", cioè quanto un materiale diffonde la luce in ogni direzione rispetto che rifletterla in un'unica direzione
- Metalness: il grado di riflettività di un materiale, cioè se il materiale assorbe la luce o la riflette principalmente
- Depth: misura la profondità, cioè la distanza degli oggetti rispetto al piano della camera
- Normal: indica come sono orientate le normali delle superfici nello spazio tridimensionale
- Render con illuminazione locale + ombre
- Render con Path-Tracing

Maggiori dettagli sono specificati nella sezione dedicata al Physical Based Rendering in appendice A.1.4.



(a) Albedo (b) Normals (c) Depth (d) Emission

(e) Metalness (f) Roughness (g) Render Real-time (h) Render Path-Tracing

Figura 3.1.: Esempio di dato

3.1.2. Tools

Non è stato possibile trovare nessun dataset che fosse formato con precisione dai dati indicati in figura 3.1. Ad esempio Structured3D [37] (in figura 3.2) e InteriorNet [38] (in figura 3.3), offrono una gran varietà di scene indoor di qualità, nonostante ci siano molti dati a disposizione, nessuno dei due ha le mappe di Roughness, Metalness ed Emissività. Questo accade perchè generalmente i dataset che vengono creati per addestrare modelli al fine di applicarli poi alla realtà, quindi non risultano particolarmente necessari informazioni specifiche sui materiali come la Roughness e la Metalness, come anche l'Emissività. Si è reso necessario creare un set di strumenti software al fine di realizzare i dati necessari.

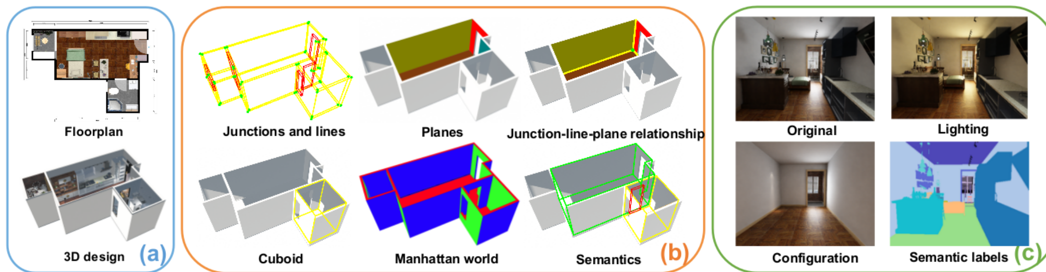


Figura 3.2.: Panoramica su Structured3D

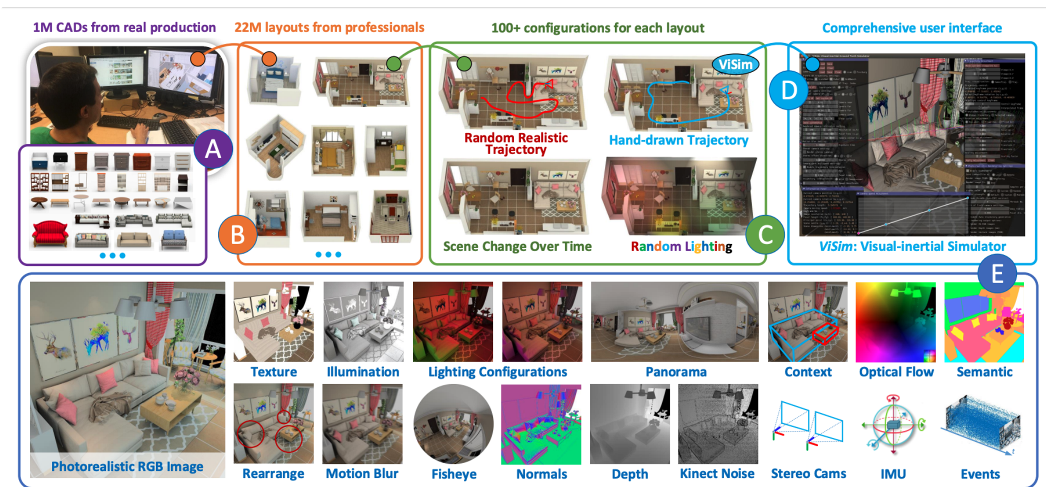


Figura 3.3.: Panoramica su InteriorNet

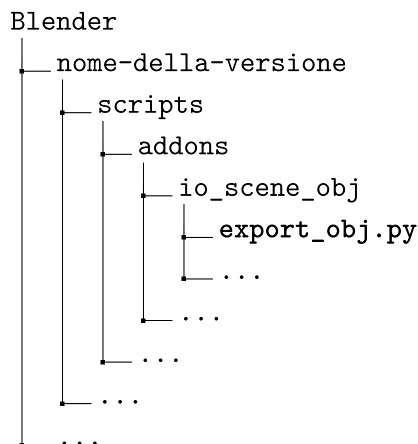
3.1.2.1. Modifiche Blender e Wavefront

Prima di spiegare i tool creati, è importante specificare che la scelta formato Wavefront OBJ, è stata dettata soprattutto dalla semplicità con cui esso rappresenta le mesh e i relativi materiali nel file .MTL. Quest'ultimo non è stato usato direttamente perchè essendo stato creato attorno al 1990, non è stato pensato per una pipeline di tipo PBR che si è diffusa dopo. Dato che non è stato mai ufficialmente aggiornato per supportare materiali per questo tipo di pipeline, sono state effettuati dei cambiamenti all'interpretazione dei valori all'interno del file .MTL. Tali modifiche si sono poi riflettute nel modo in cui è stata utilizzata la libreria[3] per l'import delle scene 3D all'interno di SimpleGbuff. Il tutto è mappato secondo la seguente tabella:

[3] Asset-ImporterLib

Principled BSDF	MTL	ASSIMP
Albedo color	Kd	AI_MATKEY_COLOR_DIFFUSE
Albedo map	map_Kd	aiTextureType_DIFFUSE
Emissive	Ke	AI_MATKEY_COLOR_EMISSIVE
Emissive map	map_emissive	aiTextureType_EMISSIVE
Roughness	Ns	AI_MATKEY_SHININESS
Roughness map	map_Ns	aiTextureType_SHININESS
Metallic	Ka	AI_MATKEY_COLOR_AMBIENT
Metallic map	map_Ka	aiTextureType_AMBIENT
Normal map	map_bump	aiTextureType_HEIGHT
Height map	disp	aiTextureType_DISPLACEMENT

Principled BSDF è lo shader[4] usato su Blender per tutti i materiali delle scene, è uno Shader PBR. Anche su Blender è stato necessario modificare l'exporter delle scene OBJ, per fare in modo che il file .mtl risultasse coerente con lo schema sopra. Il file si trova in :



È stata parzialmente cambiata la funzione **write_mtl**, nei pezzi di codice che seguono sono state riportate solo le differenze tra l'originale e la versione modificata.

[4] è un programma scritto dall'utente, che viene eseguito durante uno delle fasi della pipeline di rendering, e ne implementa il comportamento.

```python
def write_mtl(scene, filepath, path_mode, copy_set, mtl_dict):
    ...
    # Roughness
    spec = mat_wrap.roughness
    fw('Ns %.6f\n' % spec)

    # Metalness
    if use_mirror:
        fw('Ka %.6f %.6f %.6f\n' % (mat_wrap.metallic,
                                    mat_wrap.metallic,
                                    mat_wrap.metallic))
    else:
        fw('Ka %.6f %.6f %.6f\n' % (0.0, 0.0, 0.0))
    # Albedo
    fw('Kd %.6f %.6f %.6f\n' % mat_wrap.base_color[:3])
    # Emissive
    emission_strength = mat_wrap.emission_strength
    emission = [emission_strength * c for c in
                    mat_wrap.emission_color[:3]]
    fw('Ke %.6f %.6f %.6f\n' % tuple(emission))

    ...

    #### textures...
    image_map = {
            "map_Kd": "base_color_texture", # Albedo
            "map_Ka": "metallic_texture",  # Metalness
            "map_Ks": "specular_texture",
            "map_Ns": "roughness_texture", # Roughness
            "map_d": "alpha_texture",
            "map_Tr": None,
            "map_Bump": "normalmap_texture", # Normal map
            "disp": None,
            "map_emissive": "emission_color_texture" if
                        emission_strength != 0.0 else None,
            }

    ...
```

```python
def write_mtl(scene, filepath, path_mode, copy_set, mtl_dict):
    ...

    spec = (1.0 - mat_wrap.roughness) * 30
    spec *= spec
    fw('Ns %.6f\n' % spec)

    if use_mirror:
        fw('Ka %.6f %.6f %.6f\n' % (mat_wrap.metallic,
                                    mat_wrap.metallic,
                                    mat_wrap.metallic))
    else:
        fw('Ka %.6f %.6f %.6f\n' % (1.0, 1.0, 1.0))

    fw('Kd %.6f %.6f %.6f\n' % mat_wrap.base_color[:3])




    fw('Ke %.6f %.6f %.6f\n' %
                        mat_wrap.emission_color[:3])
    ...

    #### textures...
    image_map = {
            "map_Kd": "base_color_texture",
            "map_Ka": None,
            "map_Ks": "specular_texture",
            "map_Ns": "roughness_texture",
            "map_d": "alpha_texture",
            "map_Tr": None,
            "map_Bump": "normalmap_texture",
            "disp": None,
            "refl": "metallic_texture",
            "map_Ke": "emission_color_texture",
            }

    ...
```

Figura 3.4.: Codice Modificato

Figura 3.5.: Codice Originale

3.1.2.2. SimpleGbuff

SimpleGbuff è un programma in C++ creato per questo progetto, per implementare il calcolo dei G-Buffer spiegati in A.1.5.1. Prende in input da riga di comando:

- una scena 3D, in formato **WaveFront OBJ**
- un file di configurazione JSON in cui sono presenti tutte le camere[5] che deve renderizzare
- il prefisso del nome di esportazione dei file

In output restituisce le mappe di:

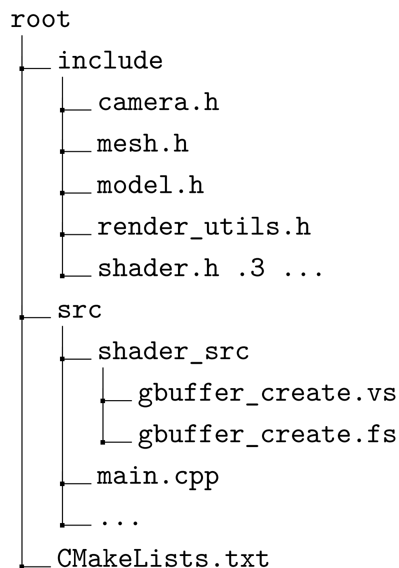
- Albedo
- Emission
- Roughness
- Metalness

[5] cioè tutti i punti di vista della scena

- Depth
- Normal

Tale applicativo si è reso necessario perchè i G-Buffer di output, appena elencati, non sono direttamente recuperabili attraverso Blender. Quindi SimpleGbuff fa solo una parte del lavoro di creazione dei dati. Difatti questo programma non va a renderizzare anche l'immagine finale, quindi non è un motore di rendering completo. Come si può già intuire per quanto riguarda la generazione delle camere e poi il calcolo dei due render aggiuntivi, ciò avverrà tramite Blender. È stato importante riuscire a far combaciare le camere della scena tra Blender e simpleGbuff. Il progetto, utilizza OpenGL ed è stato strutturato ad oggetti.

Ha la seguente struttura:



I puntini, nello schema delle cartelle, stanno a rappresentare che sono presenti dei file utili ai fini del funzionamento del programma (librerie ecc.) ma non di particolare interesse. Le varie classi sviluppate servono perlopiù a wrappare una serie di operazioni ripetitive che in alternativa sarebbero dovute essere implementate molte volte, dato che OpenGL opera molto a basso livello. Molto brevemente ora verrà presentata una panoramica del contenuto dei file presenti nell'albero delle directory subito sopra.

- **libreria camera** serve per creare, modificare e gestire le camere.
- **libreria mesh** è un wrapper che serve per gestire le mesh importate tramite **Assimp**. Si interfaccia direttamente con OpenGL e gestisce il caricamento in memoria delle mesh e anche il suo draw sul framebuffer.
- **libreria model** è un wrapper che serve a rappresentare la scena 3D importata all'interno del programma, quindi contiene una serie di mesh e di materiali ad esse collegate.

- **libreria render_utils** implementa una serie di funzioni utili per:
  - importazione delle camere da file json di configurazione.
  - creazione dei framebuffer relativi ad ogni G-Buffer sulla GPU.
  - disegno di tutta la scena sui framebuffer relativi ai G-Buffer, per ogni camera.
- **libreria shader** automatizza la parte di compilazione e attivazione (quando lo si vuole usare) di uno **shader program**[6], a partire dal **vertex shader** e dal **fragment shader** necessari per poter disegnare le mesh sul framebuffer.

Proprio in relazione a *shader.h* è di particolare interesse concentrarsi su **gbuffer_create.vs gbuffer_create.fs**, che sono rispettivamente il vertex shader e il fragment shader del programma, che hanno lo scopo di generare i G-Buffer. Sono scritti in un linguaggio chiamato GLSL[7], questo perchè vengono compilati da OpenGL per essere eseguiti tramite l'accelerazione hardware. Non sono altro che dei mini applicativi che vengono parallelizzati su sull'hardware grafico.

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec3 aTangent;
layout (location = 4) in vec3 aBitangent;

...
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;


void main()
{
...
vs_out.TBN = TBN;
vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
...

gl_Position = projection * view * vec4(vs_out.FragPos, 1.0);

}
```

Figura 3.6.: Vertex Shader

Il vertex shader si occupa di prendere in input un punto della rete e di proiettarlo sul piano della camera, quindi il valore di output deve essere assegnato a gl_Position. Ovviamente in una mesh ci sono anche centinaia di migliaia di vertici, quindi questo programma viene eseguito parallelamente su tutti i vertici.

[6] lo shader program in openGL è il programma che implementa la pipeline di rendering, e che verrà eseguito direttamente su GPU.

[7] è molto simile al C, fatto apposta per creare gli shader.

```
layout (location = 1) out vec4 gNormal;
layout (location = 2) out vec4 gAlbedo;
layout (location = 3) out vec4 gEmissive;
layout (location = 4) out float gRoughness;
layout (location = 5) out float gMetallic;
layout (location = 6) out float gDepth;

...

void main()
{

...
// NORMAL ------------------------
    gNormal = vec4(normalize(norm).rgb,1.0);

//ALBEDO -----------------------------
    if (material[0].albedo_mapping == 1){
        vec4 albedo_sample = texture(material[0].texture_albedo, texCoords);
        if(albedo_sample.a<0.3)
            discard;
        gAlbedo.rgb = albedo_sample.rgb;

        }
    else{
            gAlbedo.rgb = material[0].albedo;
    }
    gAlbedo.a = alpha_val;

//ROUGHNESS -----------------------------
    if (material[0].roughness_mapping == 1)
        gRoughness = texture(material[0].texture_roughness, texCoords).r;
    else
        gRoughness = material[0].roughness;

//METALLIC -----------------------------

if (material[0].metallic_mapping == 1)
        gMetallic = texture(material[0].texture_metallic, texCoords).r;
    else
        gMetallic = material[0].metallic;

//EMISSION -----------------------------
if (material[0].emission_mapping == 1)
        gEmissive = texture(material[0].texture_emission, texCoords);
    else
        gEmissive = vec4(material[0].emission,1.0);

//DEPTH -----------------------------------
    gDepth = gl_FragCoord.z;
}
```

Figura 3.7.: Fragment Shader

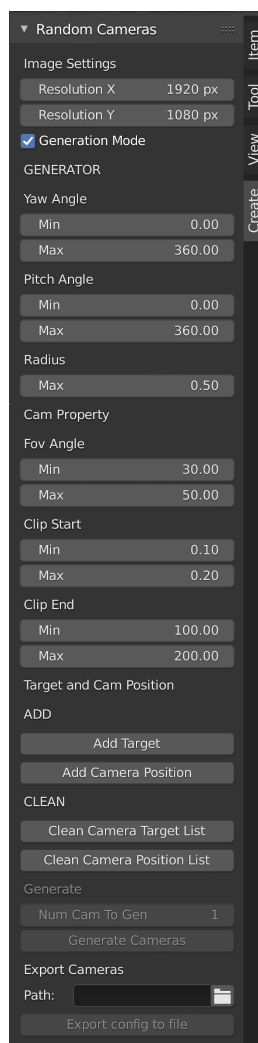
I vertici proiettati sul piano della camera, vengono poi raggruppati in triangoli, questi coprono una serie di pixel all'interno dell'immagine finale. Per ogni pixel che viene "coperto" in questo modo, il **fragment shader** effettua tutti i calcoli

necessari per definirne il colore finale nell'immagine. In questo caso non si ha un singolo framebuffer come target del rendering bensì, abbiamo 6 framebuffer[8], uno per ogni G-Buffer. Difatti come si può vedere nelle righe 15,22,32,39,45,50, si assegna l'informazione relativa al G-Buffer di riferimento. Questa tecnica di rendering su più framebuffer contemporaneamente si chiama **Multiple Render Targets**.

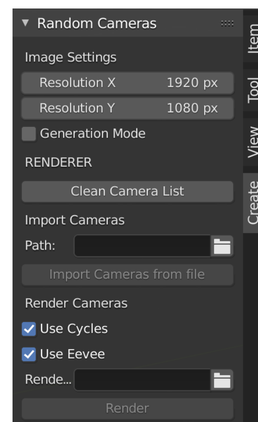
3.1.2.3. RandCameras per Blender

RandCameras è stato sviluppato per essere un addOn di Blender, nella GUI di Blender si inserisce sotto forma di pannello laterale che ha due modalità:

- Modalità Generazione
- Modalità Rendering



(a) Modalità Generazione

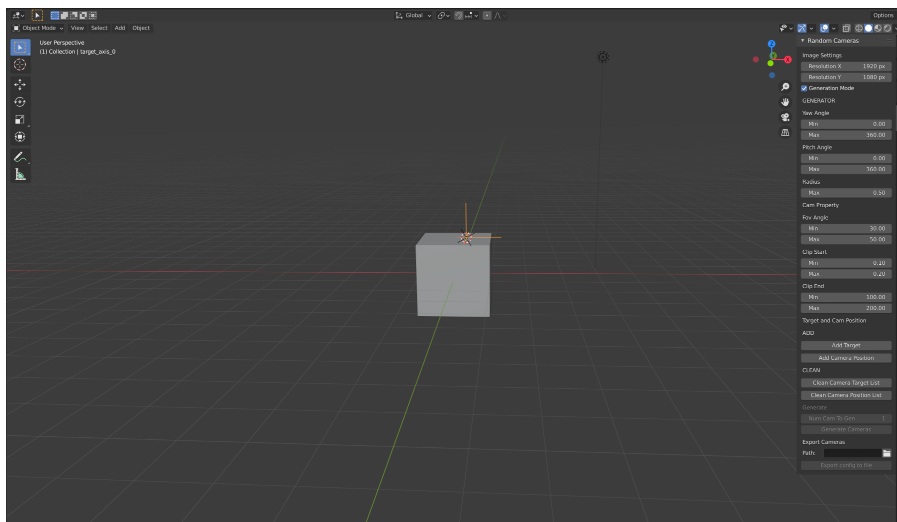


(b) Modalità Rendering

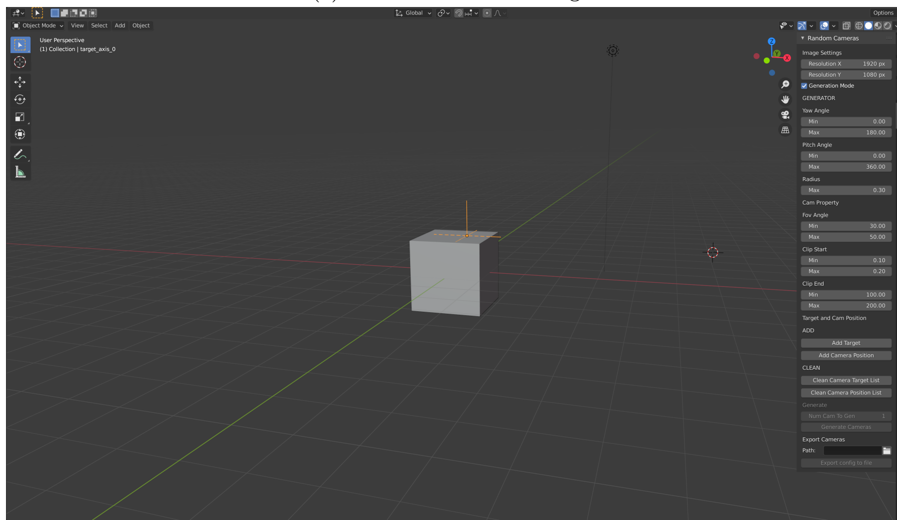
Figura 3.8.: Modalità di RandCameras

[8] le prime 6 righe del codice.

Questo programma ha lo scopo di impostare dei punti di controllo all'interno della scena 3D, cioè Target e CameraPosition, per poi generarvi una serie di camere casuali. Ci sono due tipi di punti di controllo perchè la logica di generazione delle camere consiste nell'avere un punto in cui c'è la posizione della camera (CameraPosition) ed un punto in cui la camera guarda (target). Target e CameraPosition sono dei volumi di spazio, definiti attraverso la scelta del range sull'angolo verticale e su quello orizzontale. In entrambi, viene scelto randomicamente un punto; sulla direzione definita dal segmento che va dal punto in CamPos a quello in Target si crea la camera. Quindi bisogna posizionare una serie di entrambi questi punti di controllo, attraverso i relativi pulsanti nella sezione ADD del pannello in figura 3.8a. Prima di premere sui pulsanti di ADD, per scegliere in che posizione nello spazio generare gli assi, basta spostare il cursore di Blender e gli assi verranno creati centrati su di esso.



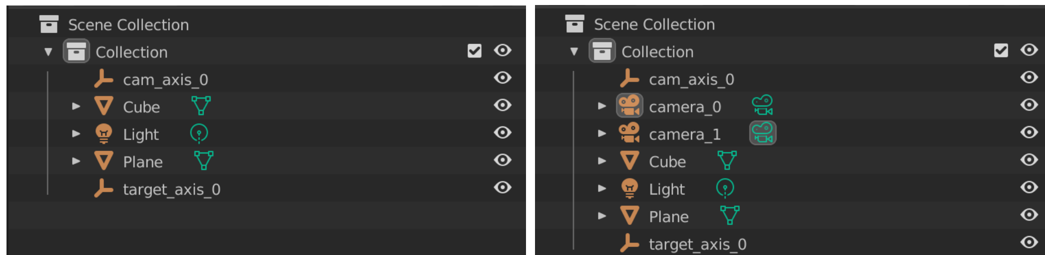
(a) Generazione di un Target



(b) Spostamento del cursore per definire la posizione di una CameraPosition

Figura 3.9.: Esempio di Generazione e Posizionamento dei punti di controllo

Una volta che si sono posizionati una serie di punti di controllo, è possibile generare le camere in maniera casuale[9], quindi guardando l'insieme degli oggetti su Blender si ottiene qualcosa simile alla figura 3.10b.



(a) Collezione con almeno 1 Target ed 1 CamPos (b) Collezione dopo la generazione di 2 camere

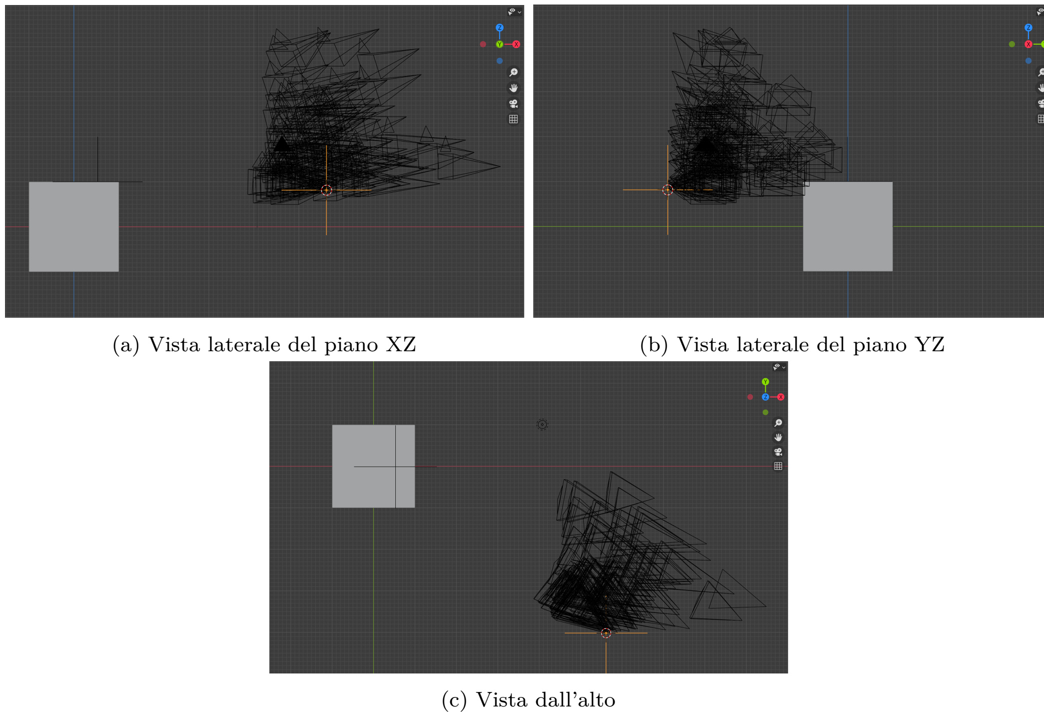
Figura 3.10.: Esempio Pre e Post Generazione

Anche se la creazione risulta realmente casuale, è possibile specificare una serie di parametri per caratterizzare ogni *Target* o ogni *CameraPosition* in modo tale che una volta posizionati, essi permettano di regolare a piacimento la porzione di volume della scena che occupano. Facendo riferimento ad una parte della Modalità Generazione del pannello, si può vedere in figura 3.11.

Figura 3.11.: Dettaglio del pannello sugli angoli

Prendendolo in analisi si nota come si può configurare (per ogni punto di controllo), l'angolo di **YAW**, l'angolo di **PITCH** e il **raggio**. Queste impostazioni descrivono le massime ampiezze dell'angolo verticale e orizzontale, che unito al raggio portano a circorscrivere in una porzione di spazio 3D la scelta di un punto random in fase di creazione della camera. Infatti prendendo ad esempio un caso base con solo un Target e solo un CameraPosition, considerando il CP con le caratteristiche specificate in figura 3.11, e considerando che il Target ingloba invece una sfera unitaria, se si generano 100 camere, si può ottenere il risultato in figura 3.12.

[9]attraverso il tasto *Generate Cameras* in figura 3.8a



(a) Vista laterale del piano XZ

(b) Vista laterale del piano YZ

(c) Vista dall'alto

Figura 3.12.: Esempio di distribuzione controllata delle camere nello spazio

Sempre guardando la 3.12 si osserva come con quella configurazione si riescano a vincolare i punti di origine delle camere all'interno di uno spicchio della sfera che si crea implicitamente con questo meccanismo.

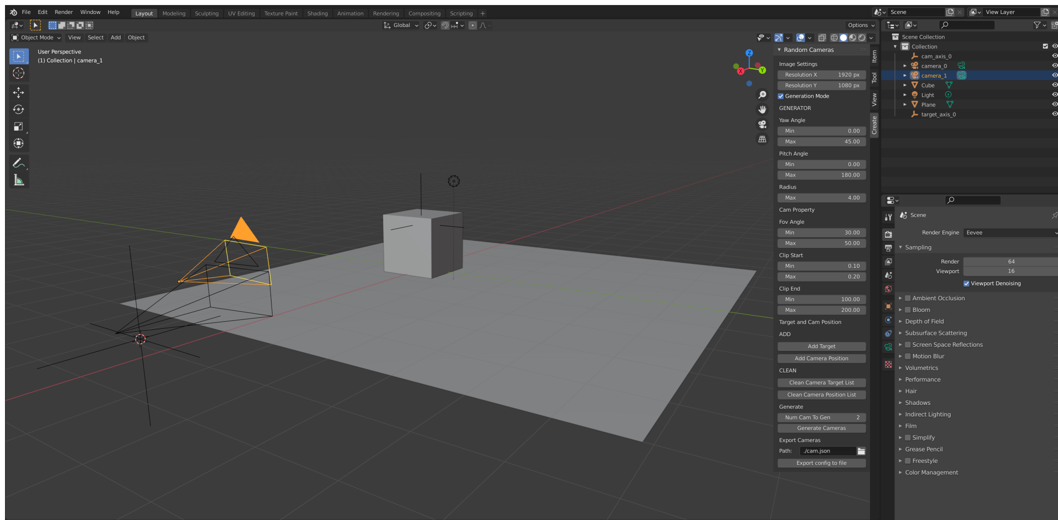
Una parte importante di questo addOn è la possibilità di esportare le configurazioni delle camere, perchè questo serve a simpleGbuff per poter fare il suo compito.

```
{"render_info":
        {
        "im_width": 1920,
        "im_height": 1080
        },
        "cameras": [
                {
                        "type": "PERSP",
                        "lens_unit": "FOV",
                        "angle_x": 0.5567703247070312,
                        "clip_start": 0.14402131736278534,
                        "clip_end": 115.98780059814453,
                        "proj": [3.4, 0.0, 0.0, 0.0,
                                0.0, 6.2, 0.0, 0.0,
                                0.0, 0.0, -1.0, -0.2,
                                0.0, 0.0, -1.0, 0.0],
                        "view": [-0.1, 0.9, 1.4e-07, -0.1,
                                0.0, 0.0, 0.9, -0.9,
                                0.9, 0.1, -0.0, -7.2,
                                -0.0, -0.0, -0.0, 1.0]},
                               ...
                ]
}
```

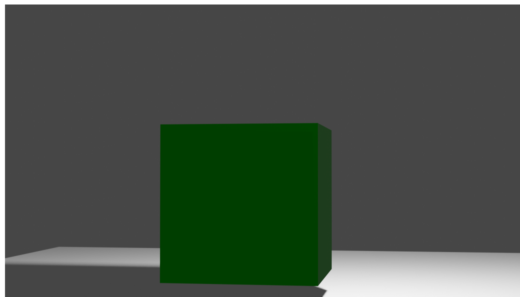
Figura 3.13.: Esempio di file JSON contenente le camere

Per finalizzare la panoramica delle funzioni principali di RandCameras, passiamo

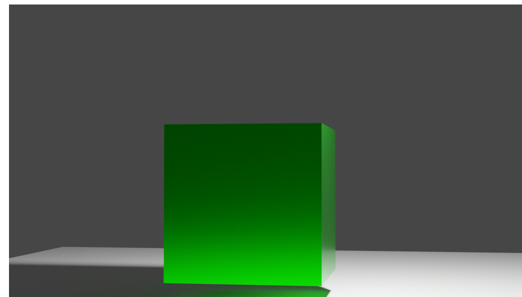
in rassegna la *Modalità Rendering* in figura 3.8b, in cui come si vede è possibile andare a calcolare il render Eevee e Cycles di tutte le camere create. Il processo viene sintetizzato dalle figure in 3.14.



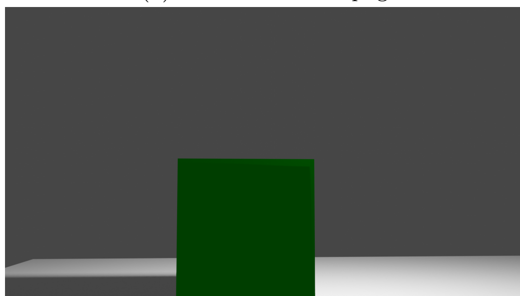
(a) Scena esempio con camere generate casualmente



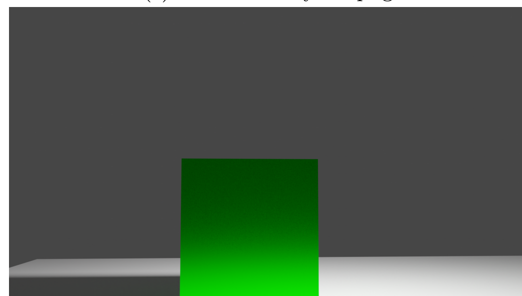
(b) camera_0_eevee.png



(c) camera_0_cycles.png



(d) camera_1_eevee.png



(e) camera_1_cycles.png

Figura 3.14.: Esempio di utilizzo della Modalità rendering

Tutto il funzionamento visto viene realizzato attraverso un singolo file python che contiene tutto il codice. Esso include una serie di classi che estendono quelle della Blender API per la realizzazione di componenti aggiuntivi[10]. Quindi in realtà quasi tutto il codice segue la strutturazione imposta dalla Blender API per creare

[10] tra cui il pannello laterale visto in figura 3.8.

componenti aggiuntivi e utilizza molte delle funzioni già messe a disposizione dal modulo bpy. Le parti più significative del codice sono quelle :

- relative alla logica di estrapolazione di un punto casuale in una porzione di spazio, realizzato dalla funzione **getRandomPointFromSpace** .
- relative la creazione di una camera con caratteristiche prefissate, realizzato da **createCam**.
- relative all'aggiunta di una camera casuale, realizzato da **addRandomCam**.
- relative all'esportazione della camera, realizzato da **exportCamerasConfig**.

```
1  # restituisce un vector 3D che è il punto nel world space randomico a partire
2  # da uno degli empty object che gli viene passato
3  # empty_def è un membro o di target_list o di cam_pos_list
4  def getRandomPointFromSpace(empty_def):
5      # calcolo dei parametri randomici
6      pparam = {'yaw' : 0.0, 'pitch': 0.0 , 'rad_scale' : 0.0} # position params
7      pparam['yaw'] = radians(random.uniform(empty_def['min_yaw'],empty_def['max_yaw']))
8      pparam['pitch'] = radians(random.uniform(empty_def['min_pitch'],empty_def['max_pitch']))
9      pparam['rad_scale'] = random.random()
10     # calcolo del punto nel local space, lppos (local point position)
11     lppos = {'x': cos(pparam['yaw'])*cos(pparam['pitch']),
12             'y':sin(pparam['yaw'])*cos(pparam['pitch']),
13             'z': sin(pparam['pitch']) }
14
15     for axis in lppos:
16         lppos[axis] *=   pparam['rad_scale'] * empty_def['rad']
17     # vado a portare il punto dal local space al world space, moltiplicando per
18     # la relativa world matrix, la posizione locale
19     lppos_vec = mathutils.Vector((lppos['x'],lppos['y'],lppos['z'], 1.0))
20     wppos_vec = empty_def['empty_obj'].matrix_world @ lppos_vec  # world point position
21     return wppos_vec
```

Figura 3.15.: Implementazione di getRandomPointFromSpace

La peculiarità di questa funzione è che il calcolo del punto casuale viene fatto prima nelle coordinate locali degli assi Target o CameraPosition, poi viene riportato in coordinate globali come si vede alla riga 20 di 3.15.

```python
# crea una camera con un nome specificato e determinate caratteristiche di visualizzazione
def createCam(name, world_mat, fov_angle = radians(45.0), clip_start=0.1, clip_end= 100.0 ):
    bpy.ops.object.camera_add()
    camera = bpy.context.active_object
    camera.name = name
    # Set field of view
    camera.data.type = 'PERSP'
    camera.data.lens_unit = 'FOV'
    camera.data.angle_x = fov_angle # in radians
    camera.data.clip_start = clip_start
    camera.data.clip_end = clip_end
    camera.matrix_world = world_mat

    return camera
```

Figura 3.16.: Implementazione di createCam

In figura 3.16 si possono vedere tutte le impostazioni che definiscono una camera. Si notano principalmente due cose:

- la posizione della camera viene definita tramite una matrice di trasformazione, la world_mat. Quindi questa matrice, che invertita corrisponderà alla **View Matrix**, è direttamente una caratteristica dell'oggetto in Blender.
- per quanto riguarda la matrice di proiezione sul piano della camera, questa non è direttamente una caratteristica dell'oggetto camera in Blender, bensì viene calcolata di volta in volta in base alle caratteristiche *angle_x*, *clip_start*, *clip_end*.

Questa informazione torna utile per capire poi il modo in cui è possibile esportare le due matrici che si vedono all'interno del file json in figura 3.13. L'esportazione avviene nel codice in figura 3.17, e alle righe di codice 11 e 14, si trovano rispettivamente il calcolo della *projection matrix* e il calcolo della *view matrix*.

Infine la creazione delle camere randomiche viene fatta tramite il codice in figura 3.18.

```python
# funzione per l'esportazione delle view e projection mat per ogni camera
def exportCamerasConfig(camera_list, export_path):
    render_settings = bpy.context.scene.render
    data_camera = {'render_info':
                    {'im_width': render_settings.resolution_x ,
                    'im_height': render_settings.resolution_y},
                    'cameras':[]}
    for camera in camera_list:
        try:
            depsg = bpy.context.evaluated_depsgraph_get()
            camera['proj'] = camera['cam'].calc_matrix_camera(depsg,
                                                x =render_settings.resolution_x,
                                                y =render_settings.resolution_y)
            camera['view'] = camera['cam'].matrix_world.inverted()


            data_camera['cameras'].append({
                'type' : camera['cam'].data.type,
                'lens_unit': camera['cam'].data.lens_unit,
                'angle_x' : camera['cam'].data.angle_x,
                'clip_start' :  camera['cam'].data.clip_start,
                'clip_end' : camera['cam'].data.clip_end,
                'proj': matToList4x4(camera['proj']),
                'view' : matToList4x4(camera['view'])
                })
        except:
            del camera

    with open(export_path, 'w') as outfile:
        json.dump(data_camera, outfile)
```

Figura 3.17.: Implementazione di exportCamerasConfig

```python
# funzione per creare randomicamente una camera
# seleziona randomicamente 1 empty target ed 1 empty cam position
# prende un punto random dall'empty selezionato target e uno dalla cam position
# dopodichè calcola la lookAt matrix, genera la camera e la aggiunge alla lista delle camere
def addRandomCam(target_list, cam_pos_list, camera_list):
    # scelgo casualmente uno degli spazi target
    rand_index = random.randrange(0,len(target_list))
    # calcolo un punto di quello spazio target, riportato nel world space
    target_wppos = getRandomPointFromSpace(target_list[rand_index])
    # scelgo casualmente uno degli spazi camera
    rand_index = random.randrange(0,len(cam_pos_list))
    # calcolo un punto di quello spazio target, riportato nel world space
    cam_wppos = getRandomPointFromSpace(cam_pos_list[rand_index])
    # calcolo della lookAt matrix
    viewMat = lookAt(cam_wppos,target_wppos,mathutils.Vector((0.0,0.0,1.0)))
    # creazione della camera ed aggiunta alla lista delle camere
    cam_name = 'camera_'+str(len(camera_list))
    cam = createCam(cam_name,viewMat.inverted())
    # vado a generare casualmente
    scene = bpy.context.scene
    #fov
    cam.data.angle_x = radians(random.uniform(scene.min_fov,scene.max_fov))
    #clip start
    cam.data.clip_start = random.uniform(scene.min_clip_start,scene.max_clip_start)
    #clip end
    cam.data.clip_end = random.uniform(scene.min_clip_end,scene.max_clip_end)
    # calcolo la projection matrix
    depsg =bpy.context.evaluated_depsgraph_get()
    render_settings = bpy.context.scene.render
    camera_list.append({'cam': cam,
                        'view' : viewMat,
                        'proj' : cam.calc_matrix_camera(depsg,x =render_settings.resolution_x,
                        y =render_settings.resolution_y)})
    return camera_list
```

Figura 3.18.: Implementazione di addRandomCam

3.1.3. Framework Creazione Dataset

Quindi con la panoramica completa degli strumenti per la creazione del dataset, si può descrivere tutto il procedimento.

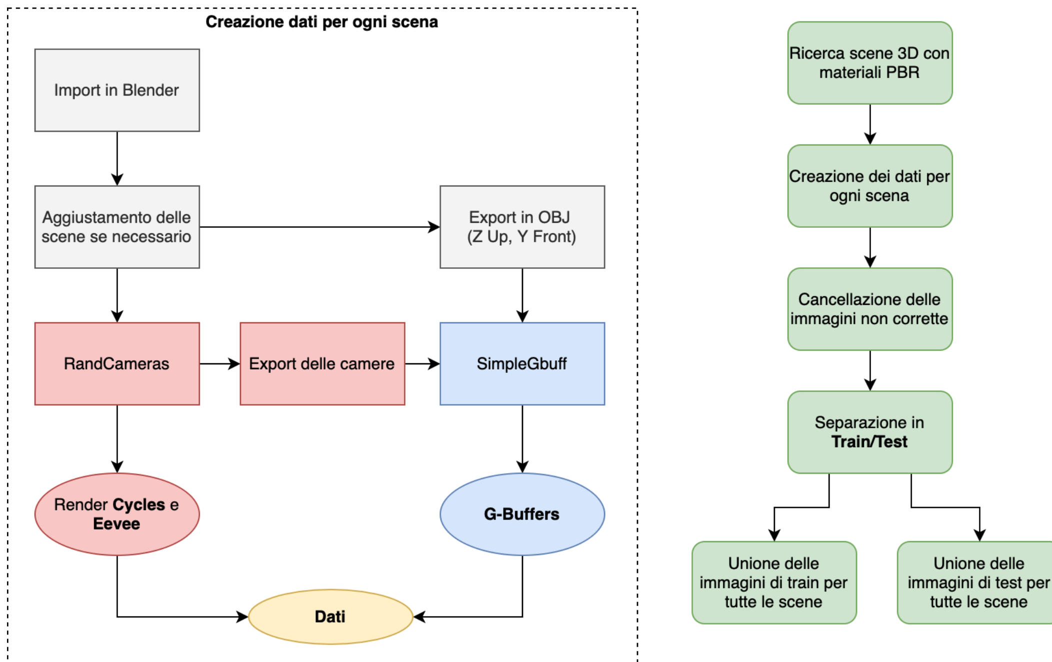


Figura 3.19.: Workflow per la creazione del dataset

Inizialmente sono state recuperate delle scene 3D da SketchFab come detto in precedenza, poi importate in Blender, dopodichè aggiustate per quello che riguarda i materiali. Questo preprocessamento delle scene si rende necessario per una serie di motivo, in primis perchè ci sono molti formati proprietari per quanto riguarda la rappresentazione delle scene tridimensionali e dei relativi materiali. Spesso poi le scene che si trovano gratuitamente online, non sono molto rifinite, quindi c'è sempre qualche piccolezza che non va bene. Le due cose messe insieme, impongono di dover fare piccoli aggiustamenti per arrivare ad avere un risultato utilizzabile. Una volta che le scene sono pronte in Blender, le si esportano nel formato OBJ[11]. È importante specificare che è necessario esportare usando come asse *up* l'asse Z e come asse *forward* l'asse Y. Questo vincolo serve perchè simpleGbuff tratta le mesh partendo dal presupposto che abbiano tale orientazione[12]. A questo punto per poter usare simpleGbuff è necessario calcolare con RandCameras una serie di camere casuali. Quindi su Blender, usando RandCameras, prima si posizionano una serie di punti di controllo e poi su di essi si calcola il numero scelto di camere casuali. Fatto questo si possono esportare le camere sul file di configurazione JSON. Poi si possono calcolare sia i render Cycles che Eevee su Blender, che i G-Buffer associati su simpleGbuff.

[11] con l'exporter modificato spiegato in precedenza
[12] generalmente l'asse Z è l'asse di forward e l'asse X (o Y) l'asse di up.

Sia simpleGbuff che RandCameras sono pensati per generare file che abbiano nomi coerenti. Ad esempio tutti i dati della camera 0, avranno come prefisso camera_0, ottendo quindi per ogni scena una directory così fatta:

```
scena1
├── camera_0_albedo.png
├── camera_0_depth.png
├── camera_0_normal.png
├── camera_0_position.png
├── camera_0_metalness.png
├── camera_0_roughness.png
├── camera_0_emissive.png
├── camera_0_eevee.png
├── camera_0_cycles.png
└── ...
```

È necessario rinominare tutti i dati per ogni scena, aggiungendo come prefisso il suo nome, perciò la cartella relativa ad ogni scena diventa:

```
scena1
├── scena1_camera_0_albedo.png
├── scena1_camera_0_depth.png
├── scena1_camera_0_normal.png
├── scena1_camera_0_position.png
├── scena1_camera_0_metalness.png
├── scena1_camera_0_roughness.png
├── scena1_camera_0_emissive.png
├── scena1_camera_0_eevee.png
├── scena1_camera_0_cycles.png
└── ...
```

Fatto ciò è necessario fare un'opera di pulizia, cancellando quei render[13] che rappresentano dei punti di vista che non hanno senso (es. che intersecano pareti, che inquadrano parti della scena poco significative tipo il cielo). Si tolgono anche i render che sono troppo simili ad altri, perchè può accadere che durante il calcolo casuale delle camere, alcune vengano molto vicine tra loro. Finita questa ripulitura, per ogni scena si separano i dati di train dai dati di test. Dopodichè è possibile unire i dati di tutte le scene e dividerli in due cartelle, una per il train e una per il test.[14]. Quest'ultima cosa è necessaria per semplificare il caricamento dei dati con pytorch, e sempre a questo scopo è stato creato uno script crea un file csv in cui per ogni riga ci sono specificati i path di tutte le immagini che compongono un dato (fare riferimento alla Fig. 3.1).

[13] con i relativi g-Buffer

[14] è necessario per semplificare poi il caricamento dei dati con pytorch.

alb	dep	norm	met	rough	em	eev	cy
s10_0_alb.png	s10_0_dep.png	s10_0_norm.png	s10_0_met.png	s10_0_rough.png	s10_0_em.png	s10_0_eev.png	s10_0_cy.png
s11_47_alb.png	s11_47_dep.png	s11_47_norm.png	s11_47_met.png	s11_47_rough.png	s11_47_em.png	s11_47_eev.png	s11_47_cy.png
s1_76_alb.png	s1_76_dep.png	s1_76_norm.png	s1_76_met.png	s1_76_rough.png	s1_76_em.png	s1_76_eev.png	s1_76_cy.png
s1_78_alb.png	s1_78_dep.png	s1_78_norm.png	s1_78_met.png	s1_78_rough.png	s1_78_em.png	s1_78_eev.png	s1_78_cy.png
s9_81_alb.png	s9_81_dep.png	s9_81_norm.png	s9_81_met.png	s9_81_rough.png	s9_81_em.png	s9_81_eev.png	s9_81_cy.png

Figura 3.20.: Esempio di file CSV ottenibile con lo script

3.2. Metodologia

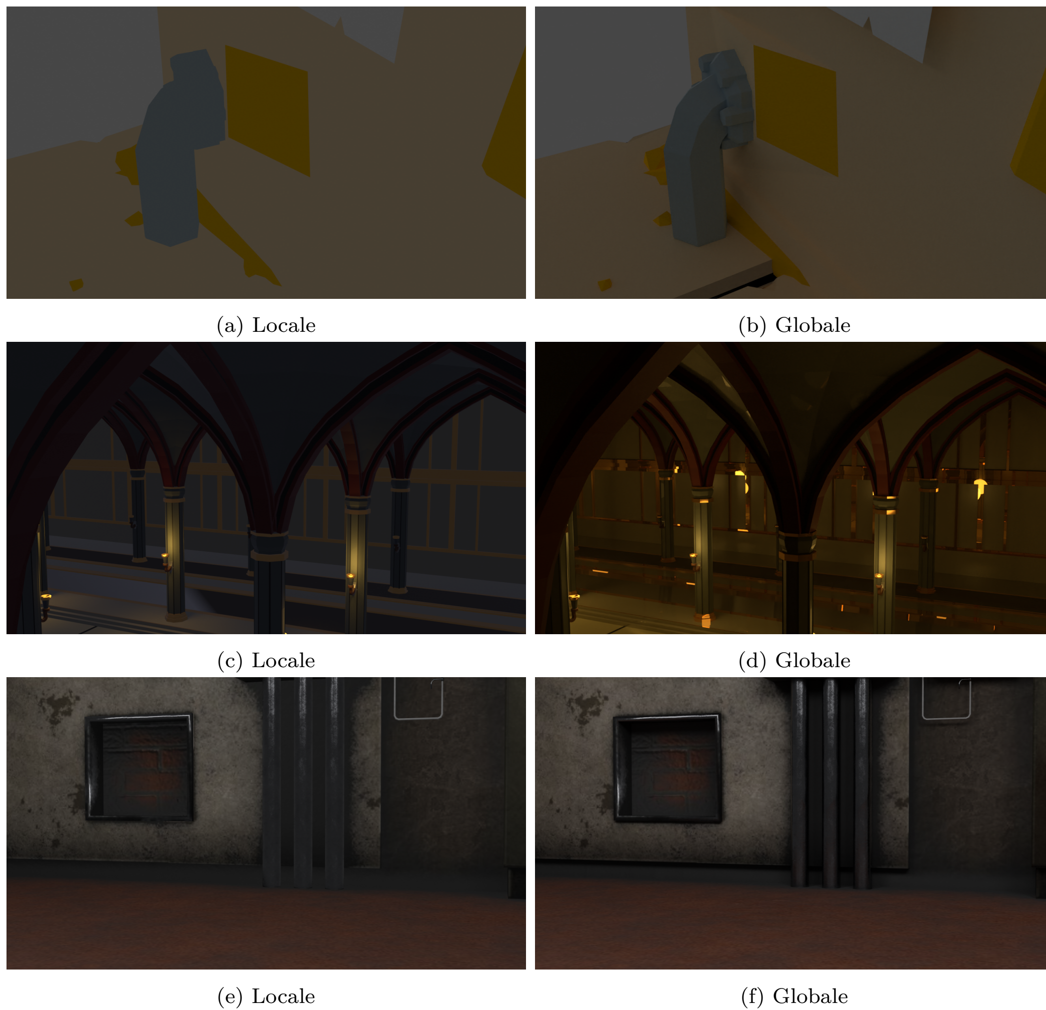
3.2.1. Panoramica

L'idea di fondo, del metodo che sta per essere presentato, è basata sul presupposto che la differenza sostanziale tra un algoritmo di illuminazione locale ed uno di illuminazione globale non risiede in caratteristiche legate all'equazione che descrive il comportamento della luce sui materiali, ma bensì nel cosa si prende in considerazione durante il calcolo di queste. Più nello specifico, in un algoritmo di illuminazione locale, lo si capisce dal termine, si considera l'effetto delle sorgenti luminose su ogni oggetto separatamente, come se non esistesse altro oltre ad esso. Questo porta ad ovvie limitazioni, dato che non si considerano in alcun modo gli effetti che si creano attraverso il "contatto" della luce con altri oggetti. Di contro un algoritmo di illuminazione globale va a prendere in considerazione tutte le mesh della scena quando bisogna calcolare il colore di ciò che si sta visualizzando a schermo. Il punto focale di questa premessa è proprio la consapevolezza che si possa usare la stessa equazione di illuminazione.

Come si può vedere in figura 3.21, in alcuni casi le differenze sono marginali, in altri casi sono molto marcate, però si nota che in termini di composizione e contenuto dell'immagine rimane tutto inalterato. Di conseguenza l'idea fondante consiste nel cercare di dare in input una serie di informazioni relative ad alcune caratteristiche della scena (G-Buffer[15]) per ottenere come output un *delta* che sommato ad una immagine calcolata con un algortimo di illuminazione locale, possa ricreare quanto più fedelmente possibile la stessa scena ma calcolata con un algoritmo di illuminazione globale. Lo scopo del modello è quello di arrivare a far ottenere un render il più simile possibile ad uno creato con Path-Tracing (in questo caso tramite Cycles). Si usa come input:

- G-Buffer che vengono generati durante il deferred-shading (Albedo, Metalness, Roughness, Emission, Depth, Normal)
- Render calcolato con Eevee (in cui è presente solo il calcolo delle ombre, come effetto di illuminazione globale)

[15]spiegati in A.1.5.1



(a) Locale (b) Globale

(c) Locale (d) Globale

(e) Locale (f) Globale

Figura 3.21.: Differenze tra illuminazione locale e globale

Si vuole ottenere come output, un'immagine che possiamo chiamare **FakeCycles**, calcolata come risultato della composizione del render EV (Eevee) e del **Delta** che viene generato attraverso il modello. L'approccio è pensato per andare a modificare un render già esistente, per migliorarlo in modo da avvicinarlo ad uno che rasenta il fotorealismo. Perciò si lavora direttamente cercando di trovare una differenza pixel a pixel, che nello spazio colore percettivo HSV possa essere mixato con i valori HSV del render EV in modo da ottenere il risultato atteso. Usare HSV al posto di RGB porta ad un cambio di prospettiva, perchè questo modo di rappresentare i colori è pensato per fare in modo che colori percettivamente vicini[16], siano effettivamente vicini nella rappresentazione. Perciò se, ad esempio, le modifiche da effettuare sono poche, quindi i colori per pixel di EV e di CY sono simili, la differenza sarà piccola pixel a pixel. Non sempre questo accade nella rappresentazione RGB[17]. Si vuole implicitamente arrivare a far calcolare al modello un *delta* che aggiunga una serie di effetti tipici degli

[16]ad esempio il rosso e il giallo
[17]si può notare con il colore Giallo

algoritmi di illuminazione globale (Cycles, Path-Tracing), non presenti però in un calcolo quasi puramente locale dell'illuminazione (Eevee + ombre). Ovviamente già preliminarmente si possono individuare una serie di possibili limitazioni dettate dalla natura delle informazioni passate al modello per realizzare il task. Ciò perchè queste non riguardano tutta la scena tridimensionale, ma solo la sua parte visibile dalla camera. Quindi il funzionamento potrebbe avvicinarsi a quello che si può definire un approccio **Screen Space**; ci sono una serie di algoritmi nella Computer Graphics che lo utilizzano, cioè usano solo le informazioni desumibili dalla camera. I limiti derivano dal fatto che qualsiasi cosa si trovi al di fuori dell'inquadratura è come se non esistesse più, comprese le parti nascoste degli oggetti presenti nella porzione di scena visibili nell'immagine.

3.2.2. DeltaNet

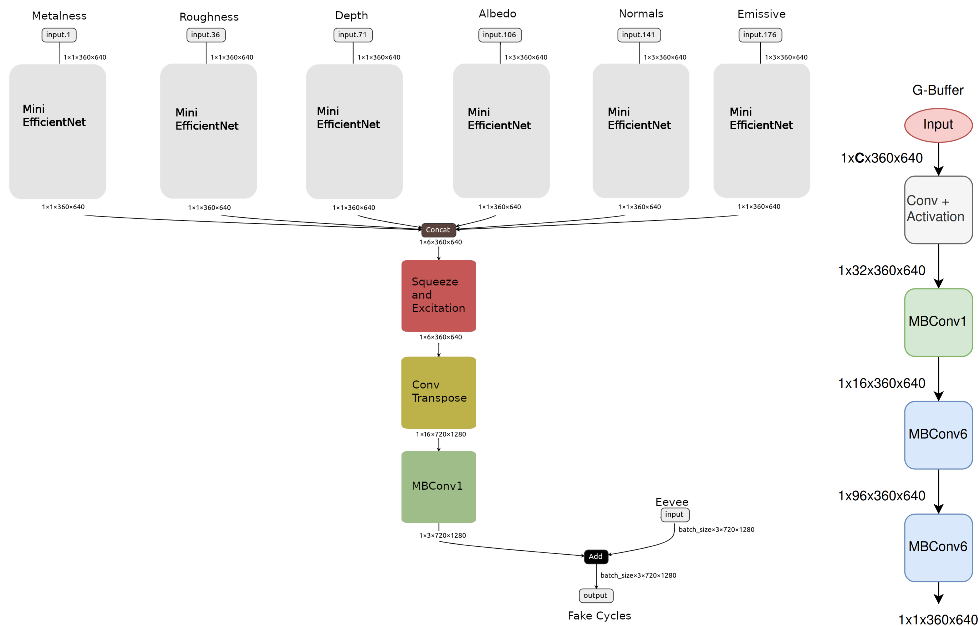
Proprio in funzione di questa generazione della differenza tra i due render, il modello prende il nome di **DeltaNet**. Per crearlo è stata costruita una CNN[18], dato che in questo modo c'è un alto grado di libertà nell'impostare il task che la rete deve realizzare, inoltre permette di ottenere una black box che fa direttamente quello che richiediamo, end-to-end, senza che si renda necessario l'inserimento di ulteriori step. Essendo un compito così generale e complesso, la scelta si è naturalmente indirizzata verso questa tecnologia. La rete è formata da una serie di blocchi mutuati dalla EfficientNet [[39]], cioè MBConv e Squeeze and Excitation[19] . La DeltaNet è basata su di essa perchè i blocchi appena citati permettono di realizzare reti profonde con un numero di parametri piuttosto basso, e l'efficienza che introducono è fondamentale dato che si lavora con immagini piuttosto grandi.

3.2.2.1. Struttura della rete

La rete prende in ingresso i 6 G-Buffer a dimensione 640x360[20]. Questa rete ha 6 branch che elaborano in maniera separata i G-Buffer, ognuno di questi branch ha come output 1 canale, che viene concatenato con gli altri 5; da questa fase escono 6 canali. L'idea è che ogni branch possa estrarre un diverso tipo di informazione dal G-Buffer che tratta, così poi da fonderla alla fine in un unico branch. Sempre propedeutico a questo trattamento separato i 6 canali vengono inseriti in un blocco di Squeeze-and-Excitation, che porta a pesare diversamente l'influenza dell'informazione generata dai 6 branch sul succesivo strato convoluzionale. Poi viene fatto un layer di ConvTranspose, utilizzato come funzione di Upsample (per ripristinare la dimensione dell'immagine alla dimensione originale di 1280x720), questo perchè ha il vantaggio di poter essere allenata rispetto ad una semplice funzione bicubica ad esempio.

[18]Convolutional Neural Network
[19]spiegati in A.4.2 e A.4.1
[20]dimezzata rispetto a 1280x720, necessario per riuscire ad inserire in memoria la rete per l'allenamento, dato che a dimensione originale avrebbe sforato la quantità di RAM della GPU.

Dopodichè viene applicato un blocco di MBConv[21] , che genera come output il Delta. Infine il risultato così calcolato, viene mixato al render fatto con Eevee. **FakeCycles** è il risultato del mix, che è un'immagine rappresentata in HSV.



(a) DeltaNet (b) MiniEfficientNet

Figura 3.22.: Schema del Generatore

La struttura di ognuno dei 6 branch sopra citati, è identica e si può vedere in figura 3.22b, è stata denominata MiniEfficientNet perchè segue la struttura di una EfficientNet ma ridotta notevolmente. Il modello completo avrebbe avuto troppi parametri, i tagli effettuati sono stati fatti per rendere la rete effettivamente utilizzabile, dato che questa parte è replicata 6 volte nella DeltaNet. È stata valutata anche la possibilità di creare direttamente una rete che generasse tutto il render, ma in questo modo il task sarebbe risultato ancora più di difficile realizzazione e ci sarebbe stata maggiore difficoltà nell'imporre alla rete la conservazione di forme e texture. Fatto in questo modo è come se si stessero aggiungendo dei vincoli al problema di ottimizzazione, perchè si vuole che l'immagine rimanga inalterata in struttura ma che cambi solo il contenuto visivo all'interno della struttura. Perciò generazione di differenze solo nei punti in cui è necessario farlo e non la creazione interamente di un'immagine che per conservare struttura e texture avrebbe richiesto ulteriori vincoli.

[21]spiegato in A.4.2.

3.2.3. Impostazione della Loss

Creato il modello, una parte cruciale per la riuscita è la scelta della loss, cioè della funzione obiettivo da minimizzare per allenarlo. In questo caso è molto difficile definire una funzione matematica di base da applicare, perchè il task è intrinsecamente complesso e una semplice misura di distanza non risulta essere sufficientemente "astratta" per guidare bene la rete durante l'allenamento[40]. In questi casi si utilizzano dei framework **GAN**[33], in cui il **generatore** viene affiancato da un'altra rete chiamata **discriminatore** che ha il compito di valutare ciò che viene creato dal generatore e fornire un valore[22] che viene usato come loss durante l'allenamento. Quindi a questo punto per ottenere dei buoni risultati è importante impostare bene il discriminatore, di cui si parlerà in 3.2.4. In questo caso si è optato più nello specifico per una **Wasserstein GAN** che nonostante impieghi più epoche per convergere assicura una certa affidabilità in termini di convergenza.

3.2.4. Discriminatore Percettivo

Lo scopo del discriminatore è cercare di capire le differenze che caratterizzano una immagine creata con Eevee da una creata con Cycles. Però queste differenze non sono facilmente rilevabili da un approccio a basso livello, perchè il layout delle due immagini in termini di composizione della scena, di oggetti sono identici, come anche le texture degli stessi oggetti. Quindi la soluzione proposta, mutuata in parte dal lavoro di Richter et al[34], serve per cercare di catturare una serie di informazioni più astratte rispetto a quelle desumibili dal semplice colore dell'immagine pixel a pixel. Rispetto al lavoro appena citato, il discriminatore è stato reso più semplice escludendo la parte che utilizza le segmentation map labellate, che nel paper di riferimento erano usate come meccanismo di miglioramento delle performance del discriminatore. Ciò perchè nel loro caso il problema è più complesso e consiste nella modifica dell'aspetto di alcuni elementi dell' immagine [23] ed inoltre nel loro caso non è possibile avere delle immagini perfettamente sovrapponibili a quelle generate, come ground truth.

[22]che indica la "correttezza" del dato prodotto.

[23]ad esempio la rete riusciva a modificare le montagne nell'immagine aggiungendo alberi, o modificava l'aspetto del fogliamo sugli alberi ecc ecc.

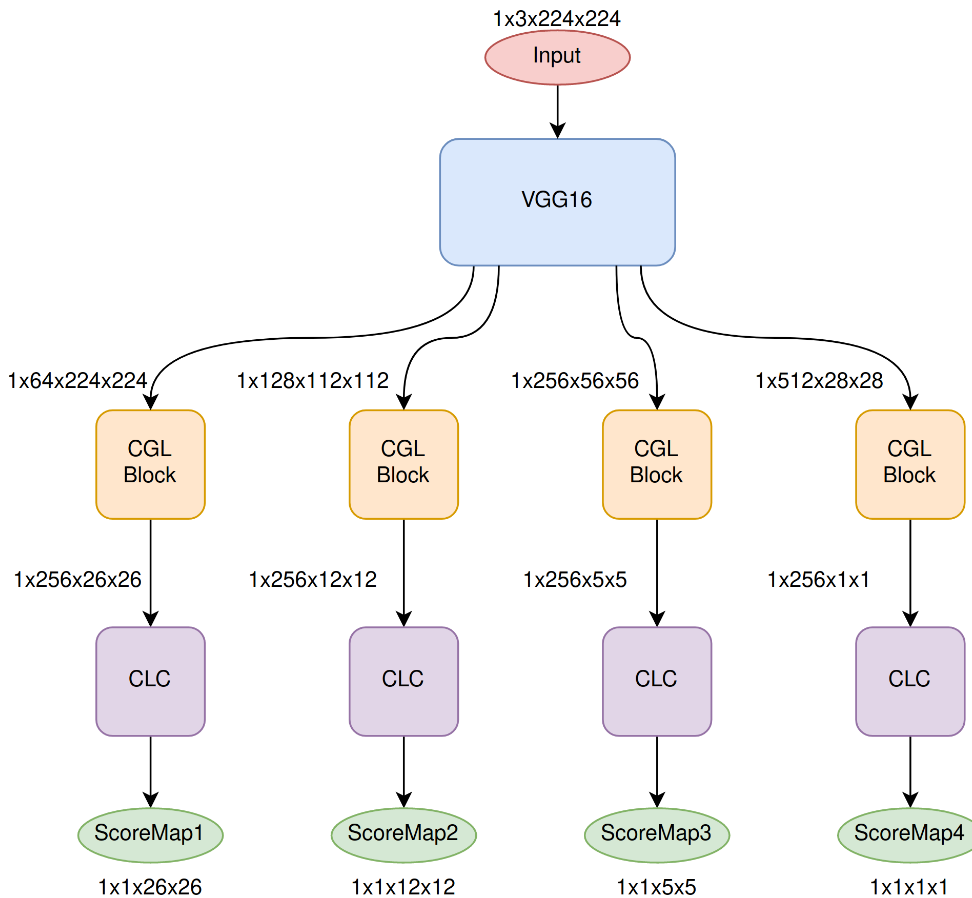


Figura 3.23.: Schema del Discriminatore

Per catturare questo concetto astratto che è parte degli studi nell'**Image Quality**, è stata utilizzata come base una **VGG16** preallenata. Questa viene usata per estrarre le features percettive che sono l'input per la rete discriminatrice vera e propria. Come si può vedere in figura 3.2.4 vengono usati più strati della VGG16, perchè ogni strato fornisce un diverso livello di astrazione dell'informazione contenuta nell'immagine. Quindi ci sono 4 piccoli discriminatori che lavorano parallelamente e forniscono 4 score differenti, uno per ogni livello percettivo. Questo discriminatore in fase di train del generatore, fornirà lo score[24] all'immagine creata, che fungerà da loss per il generatore. Il discriminatore viene allenato per separare il più possibile lo score del render Cycles dal render modificato dal Generatore[25], inoltre viene allenato 3 volte per ogni epoche rispetto al generatore e alla fine di ogni allenamento viene fatto un weight clipping, per limitare il valore dei pesi della rete[26]. Il discriminatore usando VGG16, prende in input immagini in **RGB** di dimensione **224x224**, quindi sia il render **Cycles** che **FakeCycles** vengono scalate a 224x224, e FC viene trasformata

[24] una media dei 4 score generati.
[25] questo perchè è una Wasserstein GAN
[26] la VGG16 è esclusa dal calcolo del gradiente e dall'aggiornamento dei pesi

da HSV a RGB.

3.2.5. Dettagli su Implementazione e Allenamento

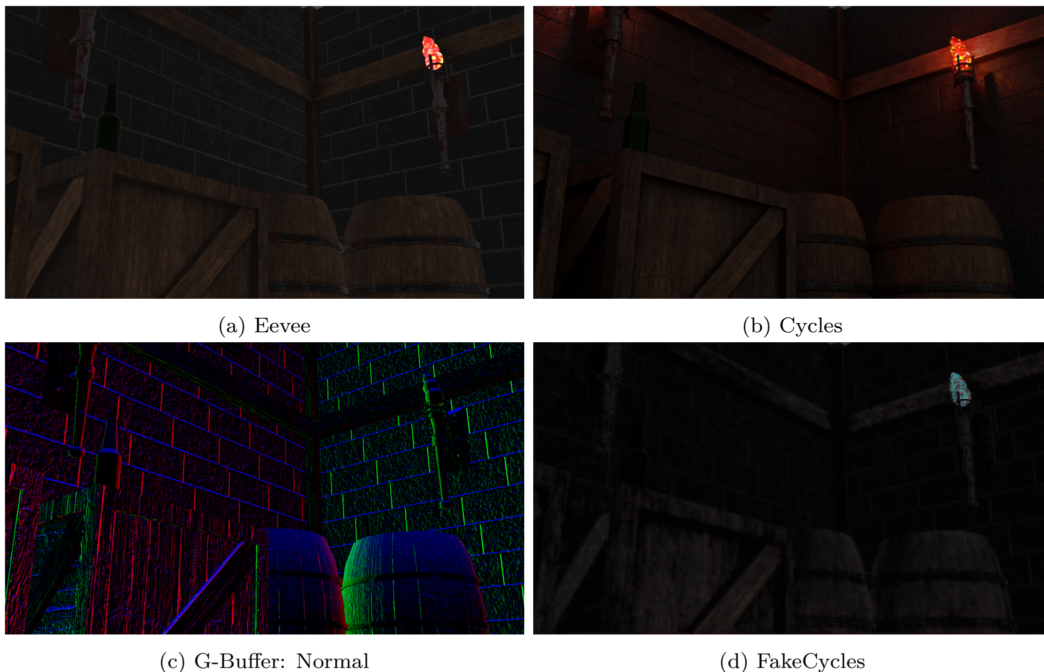
L'allenamento è stato effettuato su circa 1000 dati nella forma in figura 3.1, con 500 epoche,con batch size pari a 1, usando learning rate iniziale di 0.0002 per 250 epoche e poi con decadimento ogni 50 epoche. È stato utilizzato RMSProp come ottimizzatore. La grandezza del dataset è stata limitata dal numero di scene a disposizione (11 secene 3D), purtroppo non essendo abbastanza grandi il numero di camere casuali per ogni scena è stato limitato a 100, perchè c'era il rischio di ottenere troppe immagini vicine tra loro e quindi non creare abbastanza diversità nel dataset.

Capitolo 4.

Risultati e Commenti

In questo capitolo verranno commentati i risultati ottenuti tramite la rete, evidenziando capacità, limiti, evidenziando poi un caso d'uso in cui la rete funziona in maniera efficace, spiegando perchè in quel caso si ottengono dei risultati migliori rispetto agli altri contesti. Si darà una interpretazione al funzionamento della rete, poi si analizzeranno alcuni casi particolari che evidenziano un comportamento più complesso rispetto a quello prospettato.

4.1. Casi Normali



(a) Eevee (b) Cycles

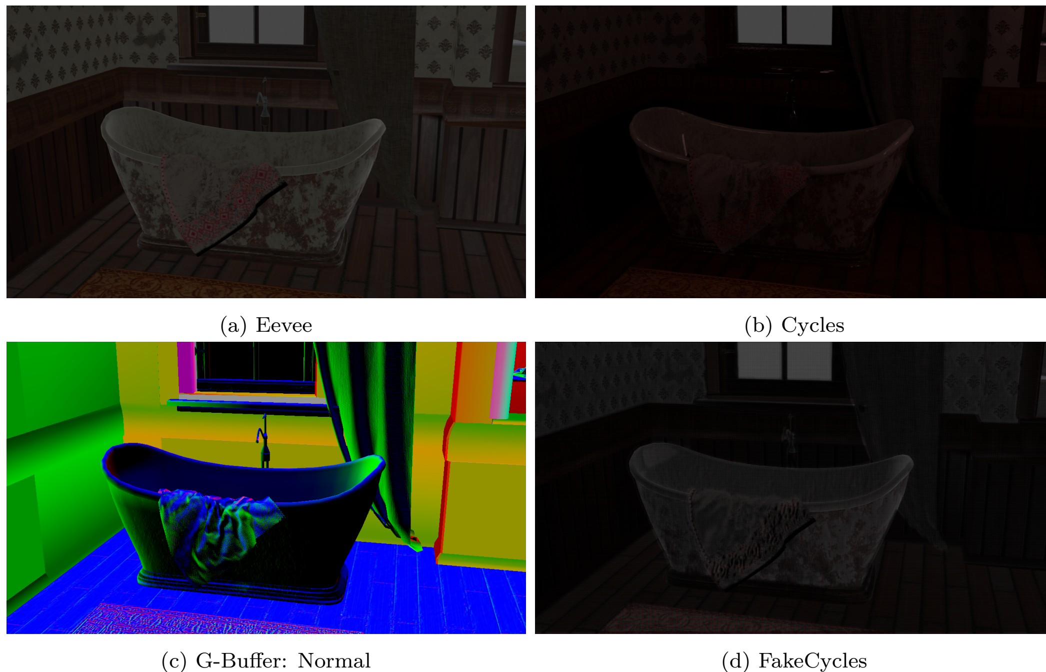
(c) G-Buffer: Normal (d) FakeCycles

Figura 4.1.: Scena 1

Come si può vedere dalle immagini della figura 4.1, l'immagine di input e l'immagine di output[1], guardandole hanno una apparenza davvero simile, tranne per una caratteristica cruciale. Questa negli algoritmi real time manca sempre, cioè il

[1]rispettivamente Eevee e Cycles.

considerare gli oggetti con materiale emissivo, come se fossero una sorgente luminosa. In questo caso quindi è come se la scena non fosse realmente illuminata per quanto riguarda l'immagine da trasformare, difatti vediamo come gli oggetti sembrino quasi splendere nel buio, in maniera del tutto impropria. Solo la torcia dovrebbe essere visibile data la sua caratteristica di emissività. Tutto questo lo riscontriamo nell'immagine di riferimento in cui si può vedere come la torcia emani luce verso la scena, creando ombre tra gli oggetti e illuminandoli leggermente anche tingendo ciò che viene toccato dalla luce con un colore tendente all'arancione/rosso. Nel render modificato si nota come ci sia uno scurimento generale di tutta l'immagine[2], una conservazione della maggiore luminosità della sorgente emissiva ma con un evidente artefatto che ne modifica il colore e aggiunge una sorta di griglia sopra di essa. Non si notano altri effetti se non una migliore ombreggiatura sul barile destro che assomiglia più a quella dell'immagine di riferimento che a quella dell'immagine iniziale. Complessivamente le modifiche non bastano per poter delineare un miglioramento nel render.



(a) Eevee

(b) Cycles

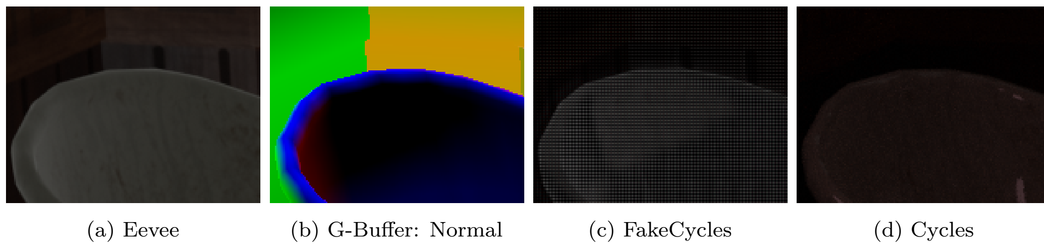
(c) G-Buffer: Normal

(d) FakeCycles

Figura 4.2.: Scena 2

Anche nella seconda scena in figura 4.2, si vede immediatamente uno scurimento generale dell'immagine, con la rete che va anche a introdurre all'interno della vasca, nella parte sinistra quello che all'apparenza potrebbe sembrare un riflesso e che non è presente in figura 4.2a. La spiegazione in questo caso sembra essere immediata se si guarda la mappa delle normali in figura 4.2c, dato che in corrispondenza di quella zona si osserva un cambiamento nell'orientamento della normale alla superficie della vasca, che delimita all'incirca quella specie di triangolo (figura 4.3).

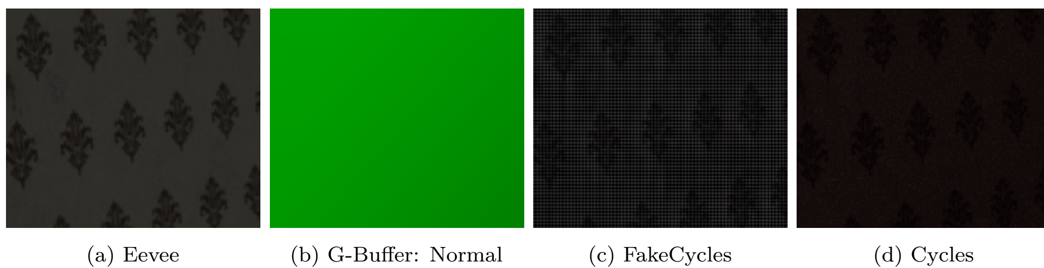
[2]presente anche nel resto dei render modificati.



(a) Eevee (b) G-Buffer: Normal (c) FakeCycles (d) Cycles

Figura 4.3.: Scena 2 dettaglio dentro la vasca

In tal proposito, un altro comportamento simile avviene sull'asciugamano che pende dalla vasca, in cui la rete mette in risalto maggiormente l'aspetto tridimensionale di questo. Si può già iniziare a trarre spunto da ciò per ipotizzare che il modello faccia in gran parte riferimento alla mappa delle normali e ai gradienti presenti al suo interno, per modificare l'aspetto del render. È interessante notare che viene anche messa in risalto la texture della parete, che in 4.2d sembra avere in rilievo il motivo al suo interno (figura 4.4). Questo però non si riesce a giustificare tramite la fig 4.2c, suggerendo che il comportamento del modello sia più elaborato rispetto ad un semplice blend delle normali con l'immagine da modificare. Nelle successive scene si potranno riscontrare queste intuizioni. Nel complesso però il render non è peggiorato in questo caso.



(a) Eevee (b) G-Buffer: Normal (c) FakeCycles (d) Cycles

Figura 4.4.: Scena 2 dettaglio sul muro

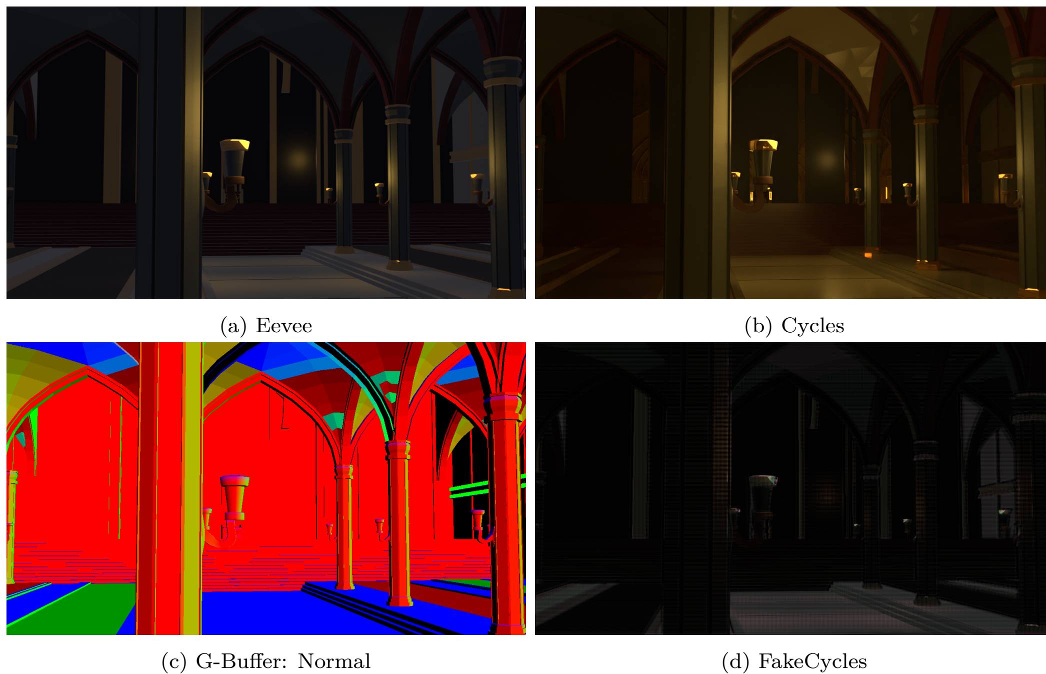
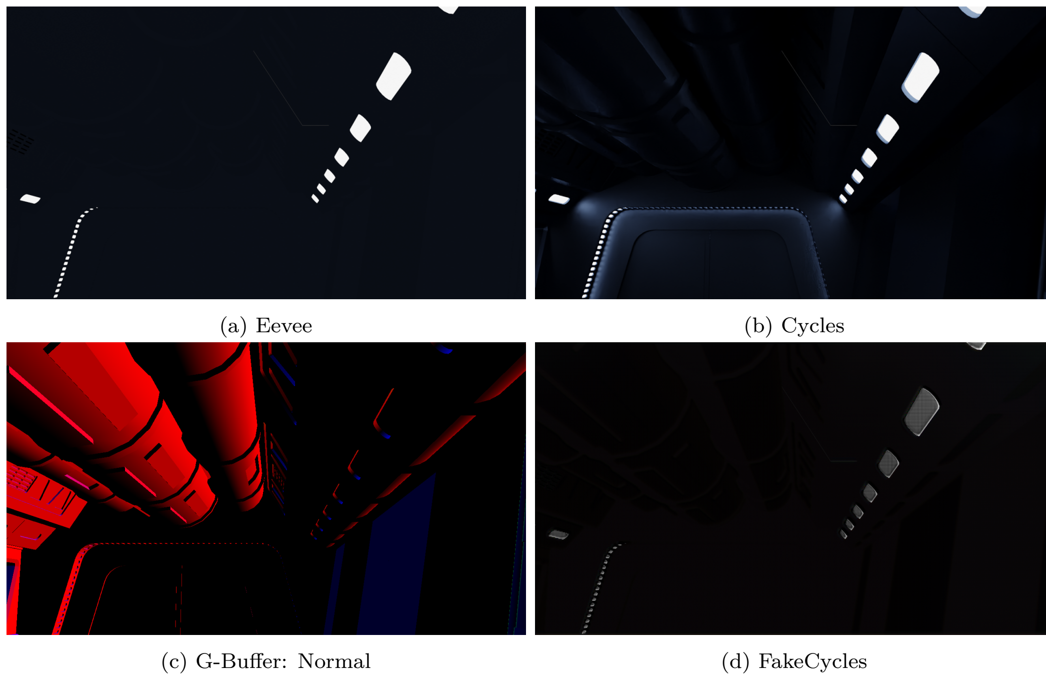


Figura 4.5.: Scena 4

Nella scena 4 in figura 4.5, si può osservare un caso che mette in crisi la rete, cioè la presenza di molte superfici altamente riflettenti. Qui non c'è molto da notare, tranne che la scena in questo caso non contiene oggetti con materiali emissivi ma bensì delle vere e proprie luci, quindi il render in figura 4.5a è molto meno piatto e fa risaltare quantomeno la struttura tridimensionale delle mesh. Non sono presenti i riflessi in essa, e nemmeno ci sono in 4.5d. Inoltre manca la tinta gialla che caratterizza 4.5b, che si crea dall'illuminazione indiretta data dal "rimbalzo" della luce gialla presente sopra le torce, su tutte le superfici della scena. Anche in questo caso in 4.5d non è presente illuminazione indiretta ma solo un progressivo scurimento dell'immagine. Non risulta un miglioramento dell'image quality complessiva, ma un peggioramento.

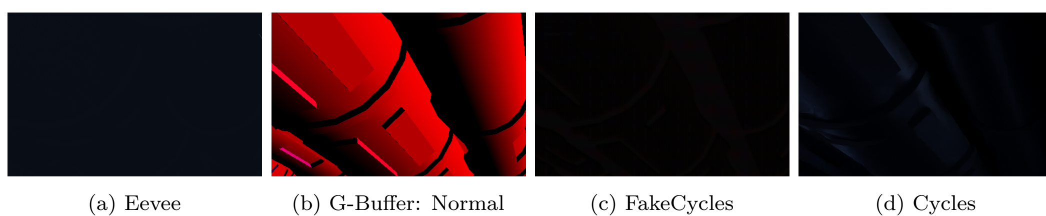


(a) Eevee (b) Cycles

(c) G-Buffer: Normal (d) FakeCycles

Figura 4.6.: Scena 5

Anche nella scena 5 in 4.6, non sono presenti sorgenti luminose, se non le lampade con materiale emissivo a lato del corridoio e sul bordo della porta in fondo, chiaramente visibile in 4.6b. Anche qui la texture emissiva viene messa in risalto ma come in 4.1d, c'è un pattern che rovina la qualità dell'immagine. Sempre in riferimento ad una delle intuizioni sopra espresse, si vede come nell'immagine 4.6a, scompaia del tutto il dettaglio dei tubi sul soffitto, ben visibile in 4.6c, questo particolare si ripropone[3] nel render modificato in figura 4.6d, seguendo però la struttura definita dalle normali e non la corretta illuminazione visibile nell'immagine di riferimento (figura 4.7).

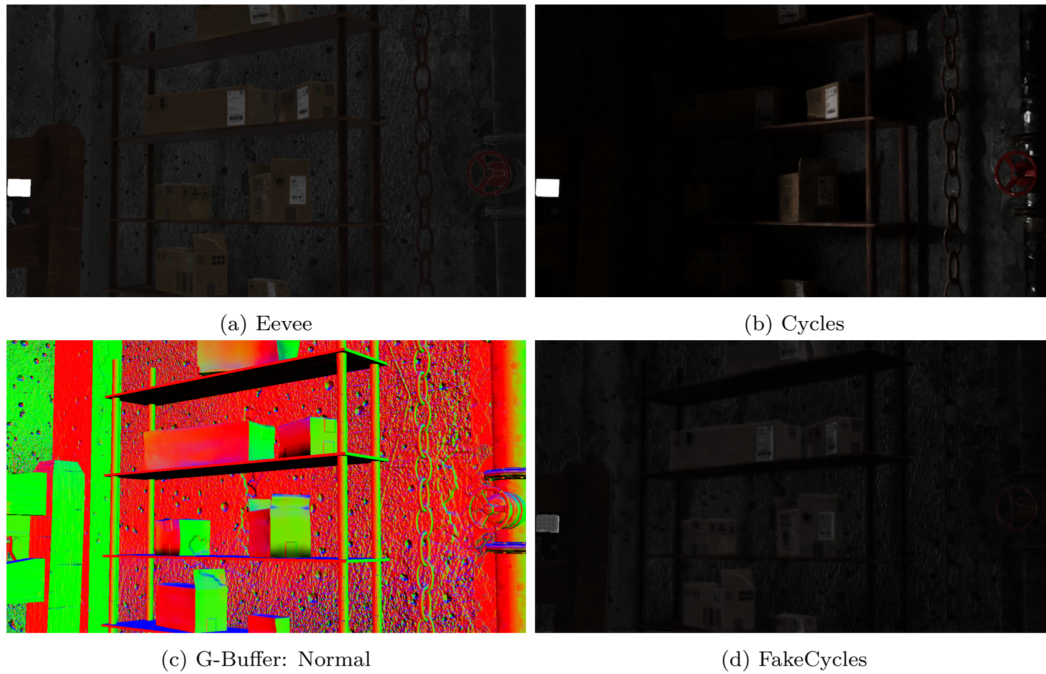


(a) Eevee (b) G-Buffer: Normal (c) FakeCycles (d) Cycles

Figura 4.7.: Scena 5 dettaglio soffitto

Si possono riscontrare gli stessi limiti sopra elencati nella scena 6 in 4.8, in cui la rete opera uno scurimento del render e rende maggiormente realistica l'apparenza della valvola circolare presente sul tubo nella parte destra dell'immagine. Non risultano altre modifiche.

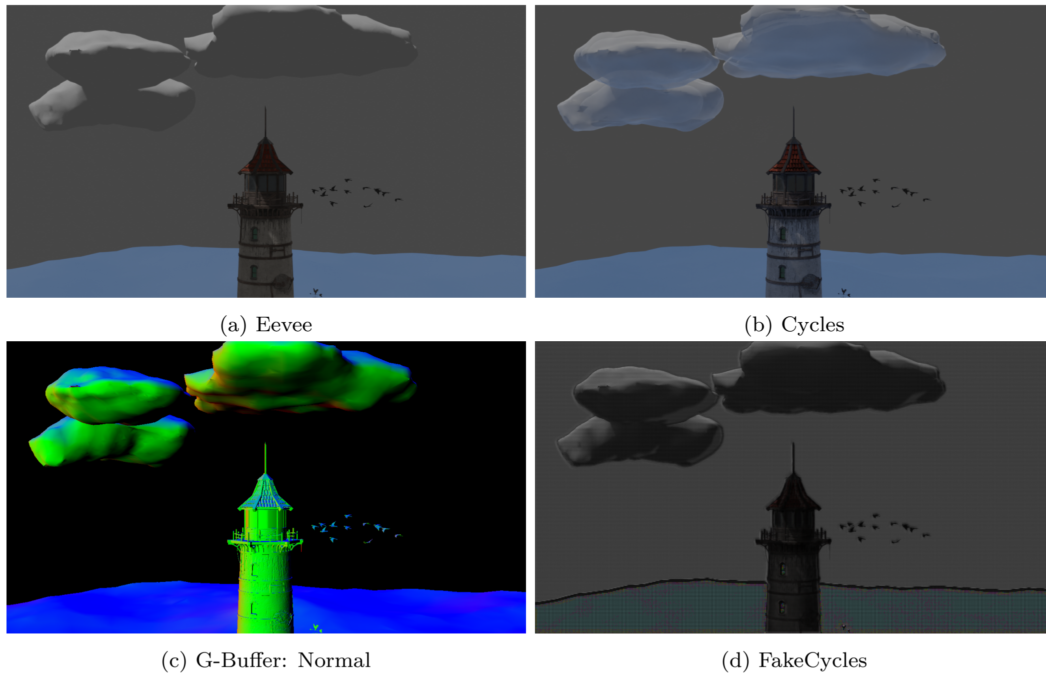
[3]non eccessivamente visibile data la scarsa luminosità dell'immagine.



(a) Eevee (b) Cycles

(c) G-Buffer: Normal (d) FakeCycles

Figura 4.8.: Scena 6



(a) Eevee (b) Cycles

(c) G-Buffer: Normal (d) FakeCycles

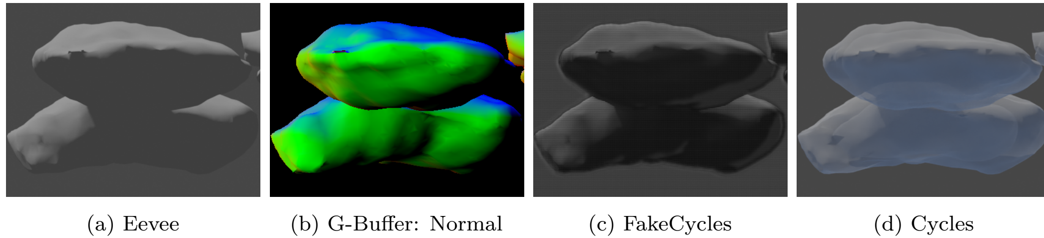
Figura 4.9.: Scena 7

La scena 7 in 4.9 ha sia una sorgente luminosa[4] che un oggetto emissivo, cioè il mare[5]. Qui le differenze tra 4.9a e 4.9d sono piuttosto sottili e si limitano alla tinta

[4]un sole che illumina in obliquo da sopra le nuvole.

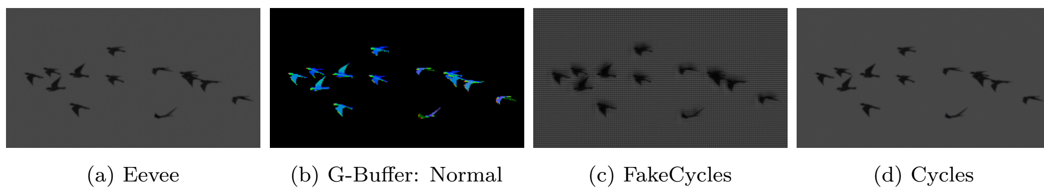
[5]è un po' inusuale questa scelta, ma la scena era già così quando è stata recuperata dal sito.

azzurra che viene proiettata dal mare verso il faro e la parte bassa delle nuvole. In questo frangente il modello va a scurire selettivamente parti dell'immagine, andando a modificare l'ombreggiatura al di sotto delle nuvole (figura 4.10), facendolo in maniera concorde al gradiente presente in 4.9c.



(a) Eevee (b) G-Buffer: Normal (c) FakeCycles (d) Cycles

Figura 4.10.: Scena 7 dettaglio nuvole

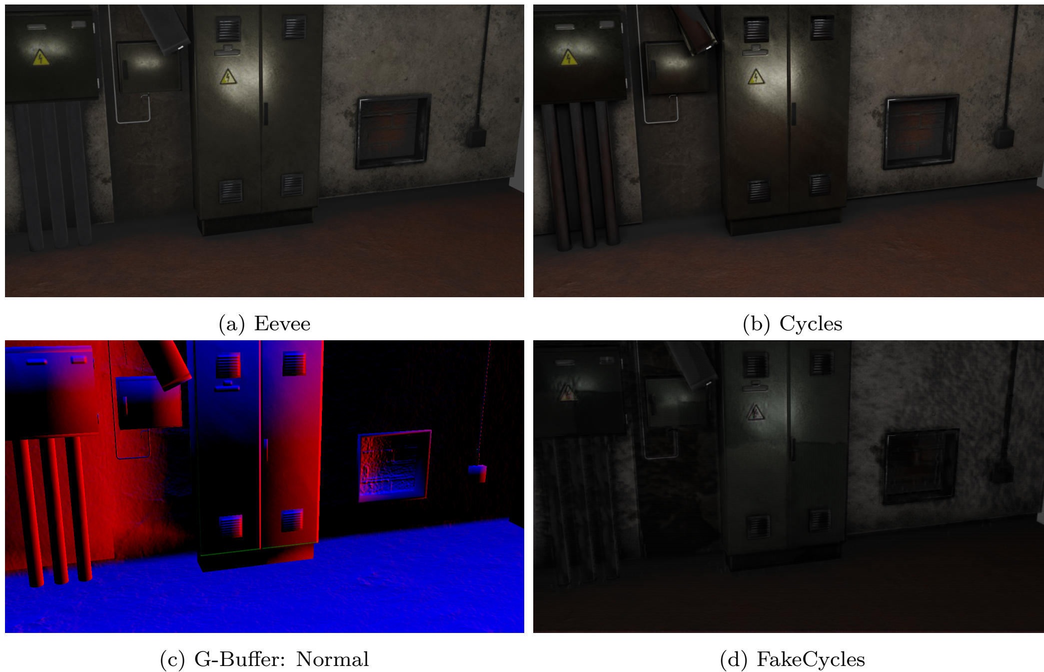


(a) Eevee (b) G-Buffer: Normal (c) FakeCycles (d) Cycles

Figura 4.11.: Scena 7 dettaglio uccelli

Viene scurito pure il faro; come atteso l'oggetto emissivo viene sovrastato dal pattern a griglia ma in compenso in questo caso le normali aiutano anche per far risaltare la forma della superficie del mare. È importante notare che in 4.9d si vede come la rete crei l'effetto di **occlusione ambientale**[6] attorno agli oggetti, ad esempio il piccolo alone che circonda il contorno del faro, la stessa cosa succede con le rondini in cielo (figura 4.11). In questo caso si crea impropriamente questo effetto, dato che non ci sono mesh molto vicine tra loro che potrebbero dare vita a questa feature visiva.

[6] è l'alone che si può vedere su mesh vicine tra loro, serve per creare profondità tra gli oggetti e simula le ombre di contatto che si creano tra cose a stretto contatto.



(a) Eevee

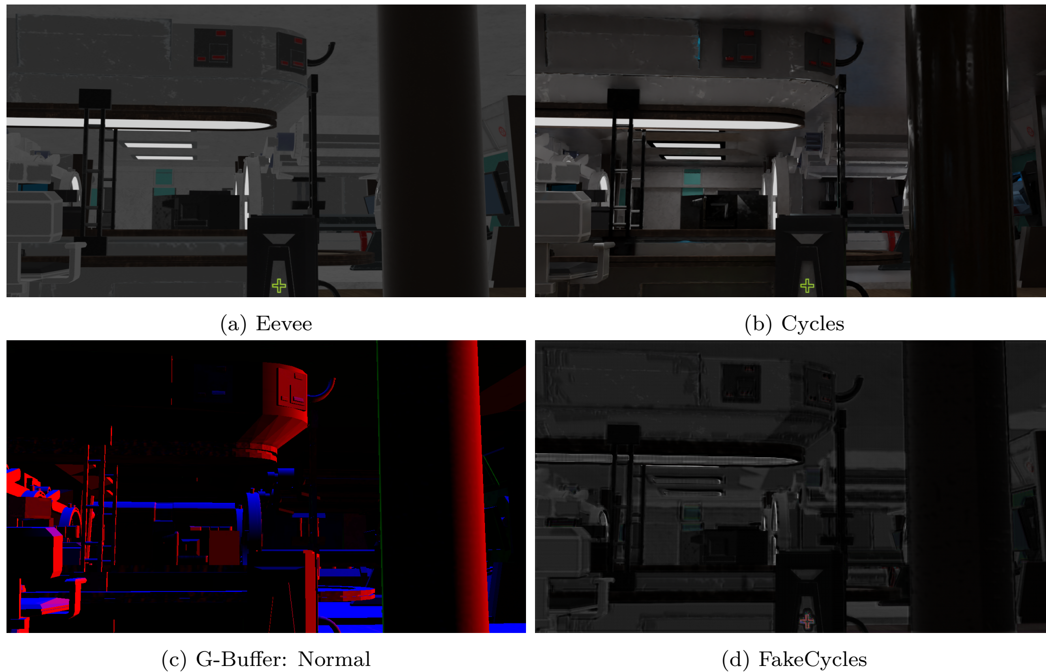
(b) Cycles

(c) G-Buffer: Normal

(d) FakeCycles

Figura 4.12.: Scena 8

Come per la scena 7, le modifiche al render della scena 8 in figura 4.12, sono estremamente marginali, e principalmente riguardanto il riflesso sul tubo che penzola dal soffitto e i tre tubi presenzi sotto il pannello a sinistra dell'immagine. In questa scena è molto più evidente che in altre scene l'influenza piuttosto massiccia delle normali nel processo di modifica del render. Infatti come si vede in 4.12d sui 3 pannelli, partendo da sinistra, è visibile una sorta di suddivisione in 4; questo è in linea con il cambiamento delle normali in figura 4.12c. Però se si fa attenzione sul muro presente sulla destra dell'immagine, si può vedere che la rete esalta la texture del muro, per donarle una maggiore tridimensionalità.



(a) Eevee (b) Cycles

(c) G-Buffer: Normal (d) FakeCycles

Figura 4.13.: Scena 9

Nella scena 9 non è presente illuminaizone se non gli oggetti emissivi, essendo la scena per la maggior parte composta da materiali molto diffusivi, si ottiene che 4.13a sia piuttosto piatta. Nonostante si evidenzino i soliti limiti in termini di resa della sorgente emissiva e in termini di riflessioni[7] la rete riesce a donare un maggior senso di profondità.

In tutti i render modificati è presente un pattern a griglia simile a quello che si crea sui materiali con caratteristiche di emissione, che potrebbe essere causato dall'utilizzo dei G-Buffer ad una dimensione dimezzata, per riuscire a caricare la rete in memoria.

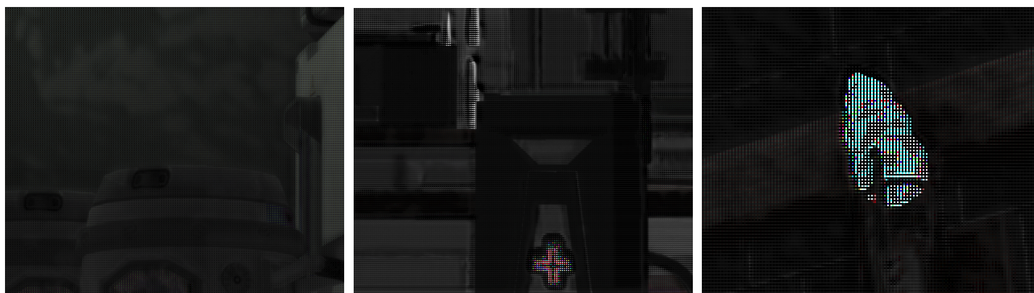


Figura 4.14.: Esempi dell'artefatto

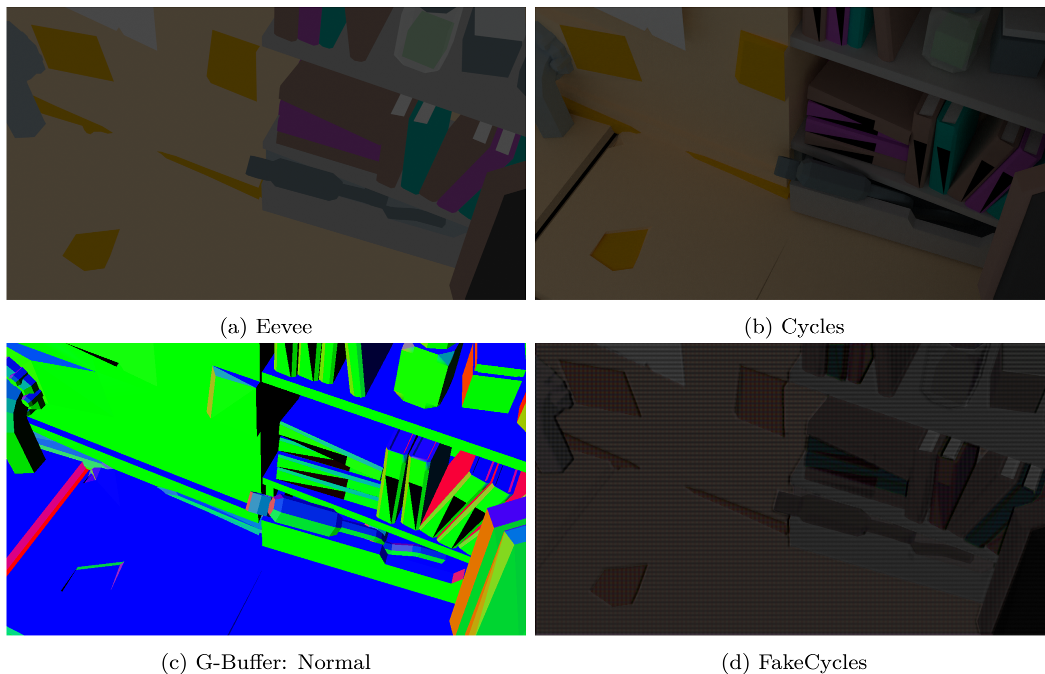
Quindi alla luce dei risultati evidenziati si può affermare che è plausibile l'ipotesi secondo la quale la mappa delle normali abbia un peso decisamente importante nel ritocco del render, inoltre però la rete non si limita ad un uso triviale di questo

[7] si notino la colonna e le aree sul soffitto che emettono luce.

G-Buffer, bensì applica e crea altri effetti[8] e lo fa in maniera differente in base al contesto, come si vedrà in 4.3. È fondamentale notare che tutto questo viene realizzato senza che la rete deformi in maniera significativa la struttura della scena, l'aspetto degli oggetti e le loro texture[9].

4.2. Caso Favorevole

Come si è potuto intendere dalla sezione 4.1, ci sono delle condizioni in cui il modello riesce ad essere più efficace nell'operare un reale miglioramento dell'immagine. Il caso più favorevole di tutti risiede nell'avere una scena con nessuna luce, quasi esclusivamente materiali diffusivi e l'illuminazione data solo da qualche oggetto con caratteristica di emissione. A questo punto la rete deve principalmente usare le normali per ricostruire la geometria, e in più aggiunge un po' di occlusione ambientale.



(a) Eevee (b) Cycles

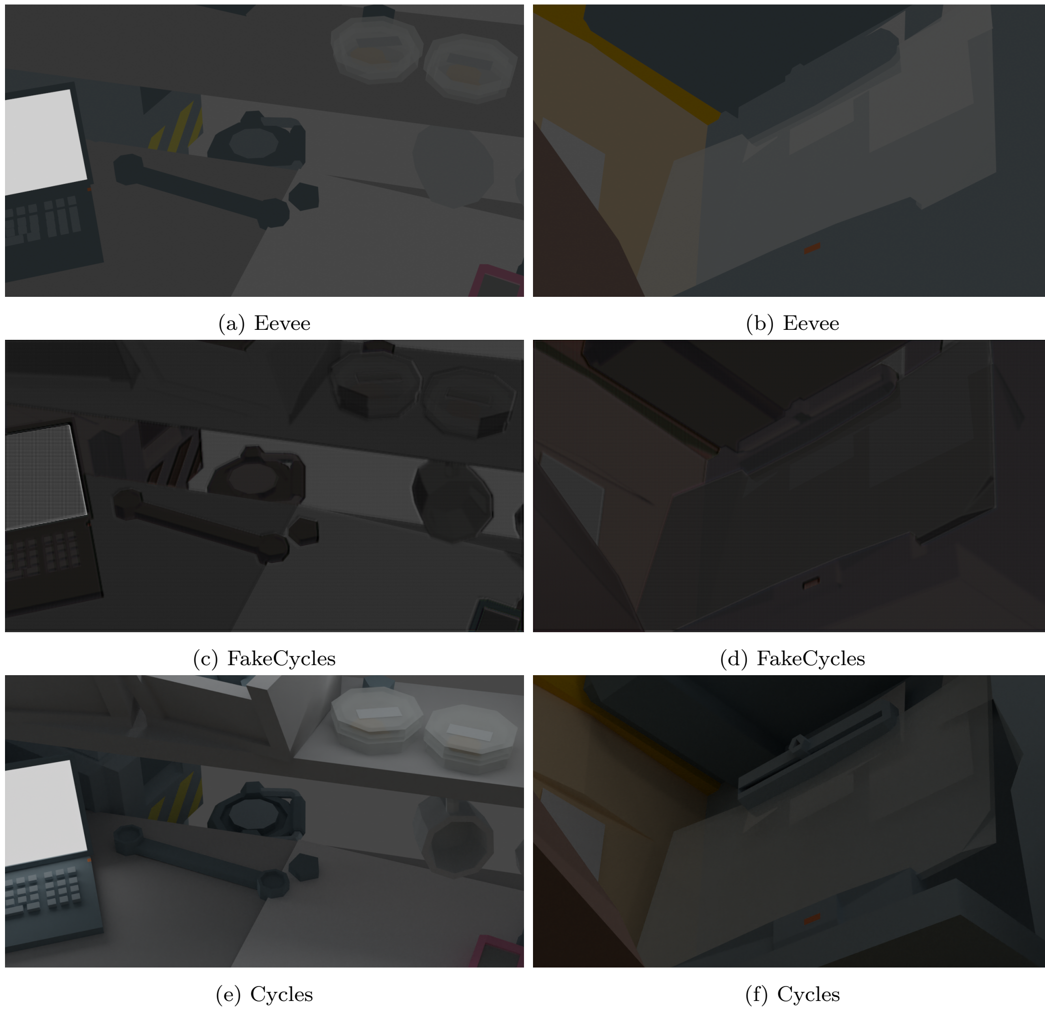
(c) G-Buffer: Normal (d) FakeCycles

Figura 4.15.: Scena 3

Nella figura 4.16 si vedono perfettamente gli effetti di questo miglioramento ed almeno in questo contesto si può affermare che il render modificato sia molto più vicino all'immagine di riferimento rispetto a quella da modificare.

[8] come l'occlusione ambientale, vedasi 4.9.

[9] a meno di alcuni casi in cui lì modifica in maniera impropria ma credibile, come in 4.21



(a) Eevee (b) Eevee

(c) FakeCycles (d) FakeCycles

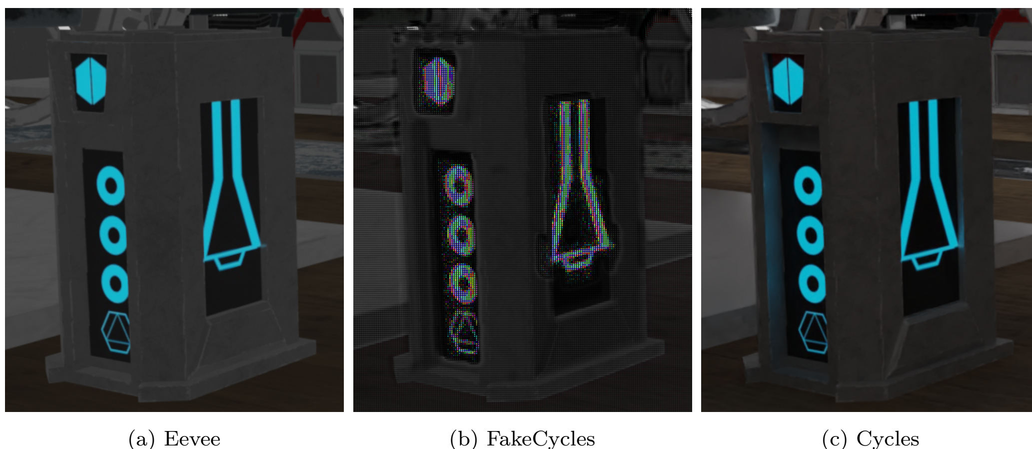
(e) Cycles (f) Cycles

Figura 4.16.: Altri esempi della scena 3

4.3. Casi Particolari

In questa sezione si vedrà come in alcuni casi il risultato in output contenga del contenuto che è stato generato dalla rete direttamente. Cioè, come si vede dalle immagini 4.19d, 4.20d, 4.21d, ci sono degli elementi nell'immagine che sono stati generati e non sono direttamente associabili a nessuno dei G-Buffer che sono stati dati in input alla rete. Esaminandoli più nel dettaglio vediamo che in:

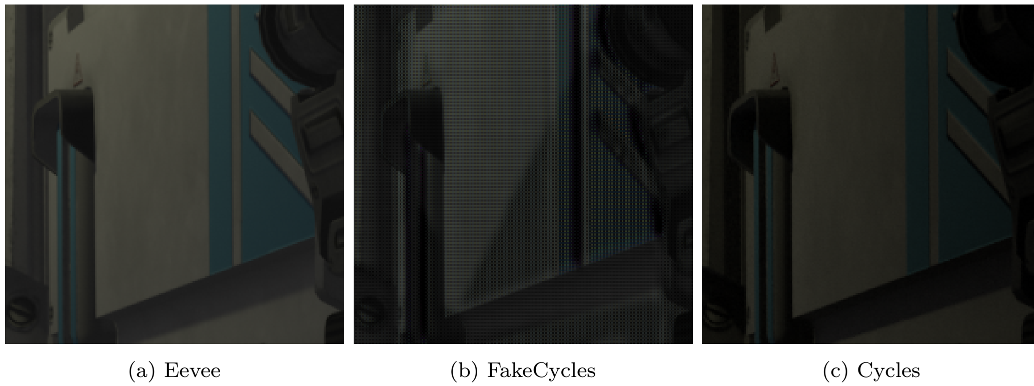
- 4.19 se si concentra l'attenzione sull'oggetto sulla sinistra dell'immagine, si può vedere come sulla sua parte sinistra[10], l'output viene ricostruito e si avvicina all'immagine di riferimento, al contrario di 4.19a che invece appiattisce la struttura intorno ai cerchi luminosi. È interessante perchè viene proprio modificata la forma apparente di quell'oggetto e queste modifiche non sono frutto di un utilizzo triviale dei G-Buffer, dato che al contrario dei casi espressi in 4.1 questi dettagli non sono presenti nè nelle normali nè in nessuna delle immagini date in input (figura 4.17).
- 4.20d concentrando l'attenzione sul pannello laterale in alto sulla sinistra dell'immagine, si può vedere che è presente una forma triangolare che viene totalmente creata, di nuovo, dal nulla, dato che non c'è nessun dato di input che suggerisca in alcun modo la presenza di quella forma in quel preciso punto (figura 4.18).
- 4.21d si nota che la texture della parete di fondo è totalmente nuova e credibile, non è desumibile dai dati di input direttamente ed è frutto solamente della concezione di "correttezza" del Generatore.



(a) Eevee (b) FakeCycles (c) Cycles

Figura 4.17.: Scena 9 dettaglio

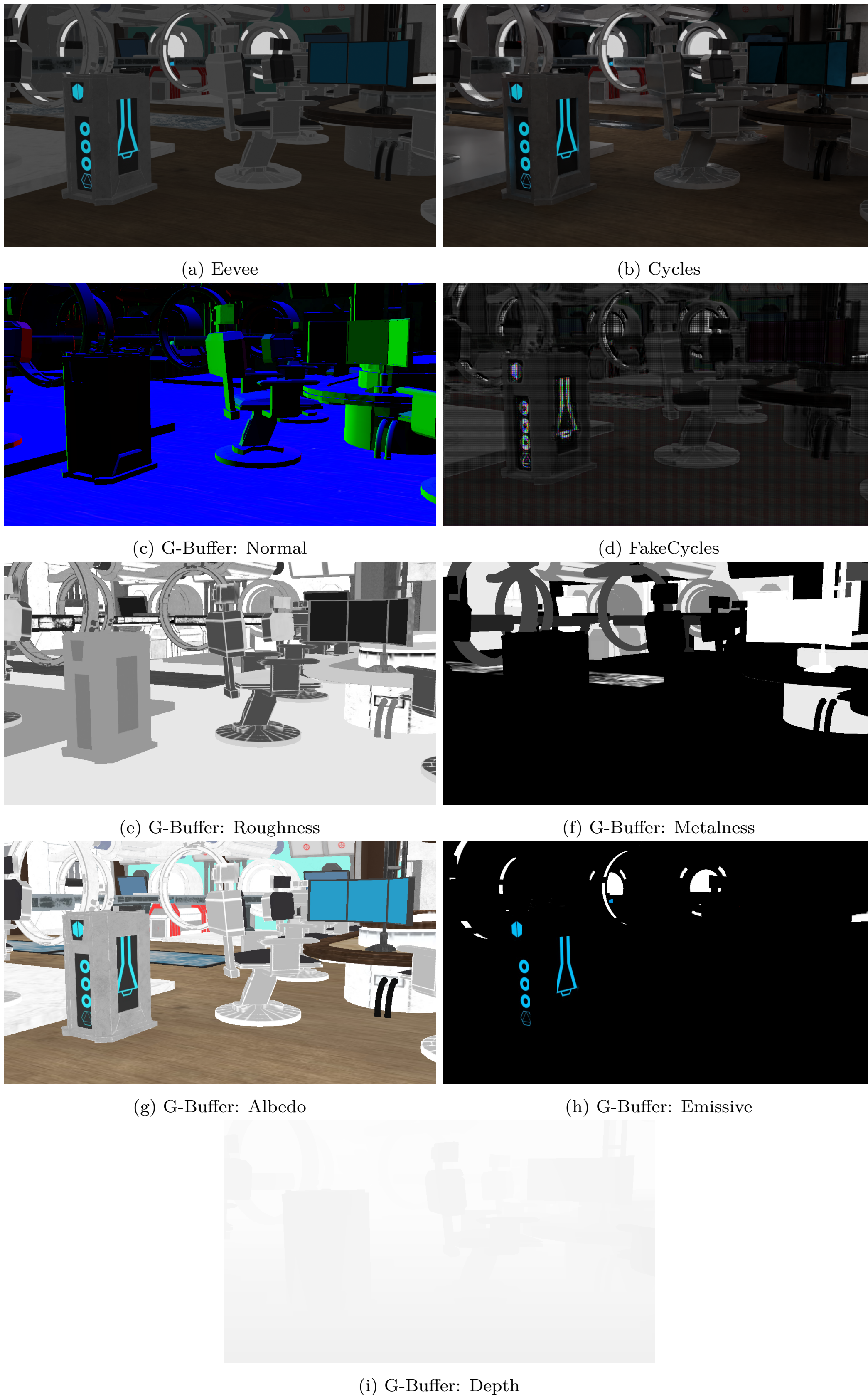
[10]dove sono presenti i 3 cerchi luminosi.



(a) Eevee (b) FakeCycles (c) Cycles

Figura 4.18.: Scena 11 dettaglio

Quindi detto questo si può affermare che il modello non offra un semplice blend di uno dei G-Buffer o parte di esso, con l'immagine. Al contrario riesce ad inserire anche delle modifiche sostanziali a forme e texture nella misura in cui l'immagine di partenza lo richieda. Questo comportamento non banale è stato forzato e trovato grazie all'allenamento tra Generatore e Discriminatore e all'equilibrio a cui convergono entrambi. Quindi ciò che si vede nei render modificati è ciò che G è in grado di generare e che D valuta positivamente. Questo ci fa capire anche in parte anche come il discriminatore valuta le immagini, considerando molto la visibilità della struttura tridimensionale della scena, e anche la forma delle texture, che spesso vengono parzialmente modificate o fatte risaltare.



(a) Eevee

(b) Cycles

(c) G-Buffer: Normal

(d) FakeCycles

(e) G-Buffer: Roughness

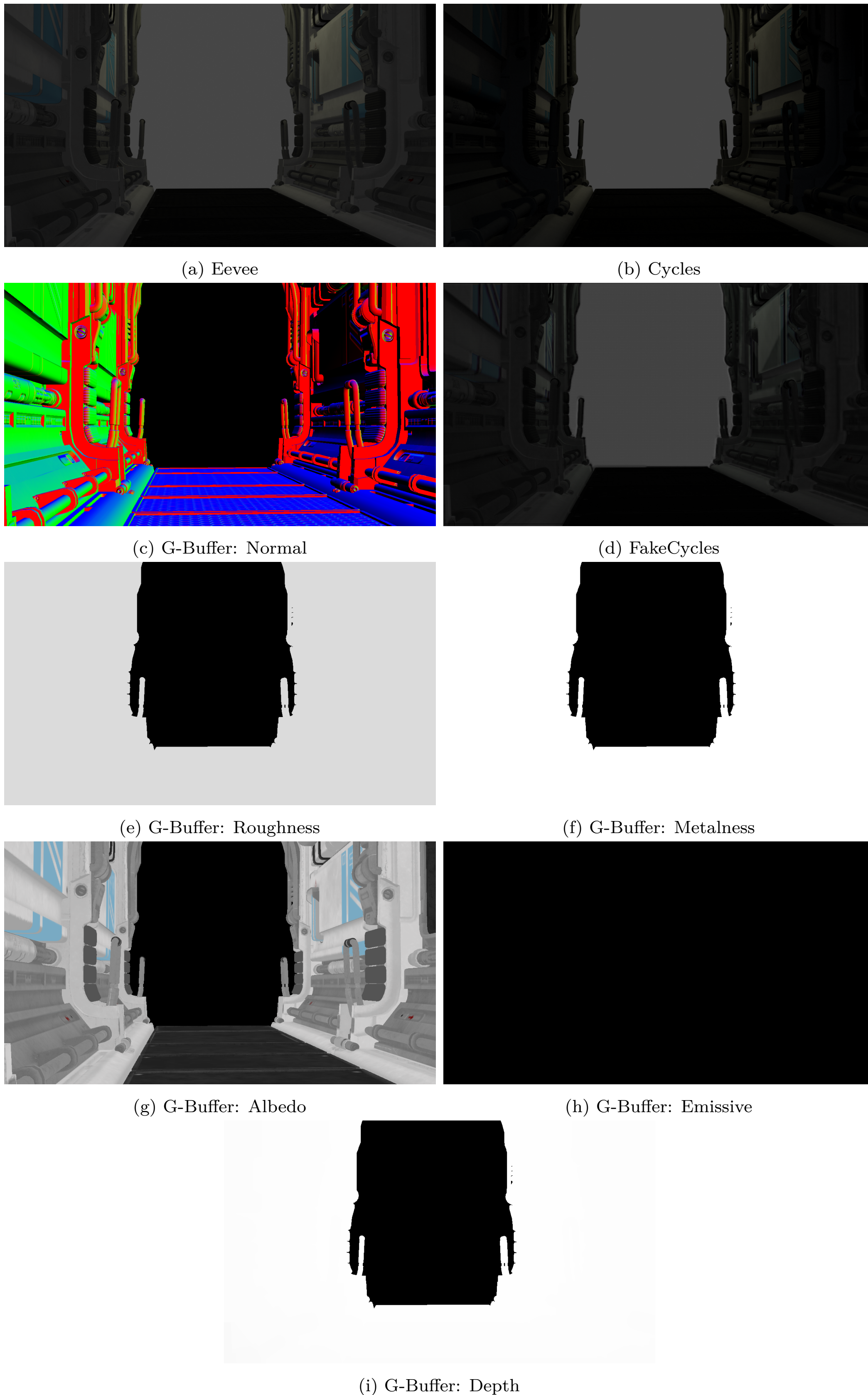
(f) G-Buffer: Metalness

(g) G-Buffer: Albedo

(h) G-Buffer: Emissive

(i) G-Buffer: Depth

Figura 4.19.: Caso particolare scena 9



(a) Eevee

(b) Cycles

(c) G-Buffer: Normal

(d) FakeCycles

(e) G-Buffer: Roughness

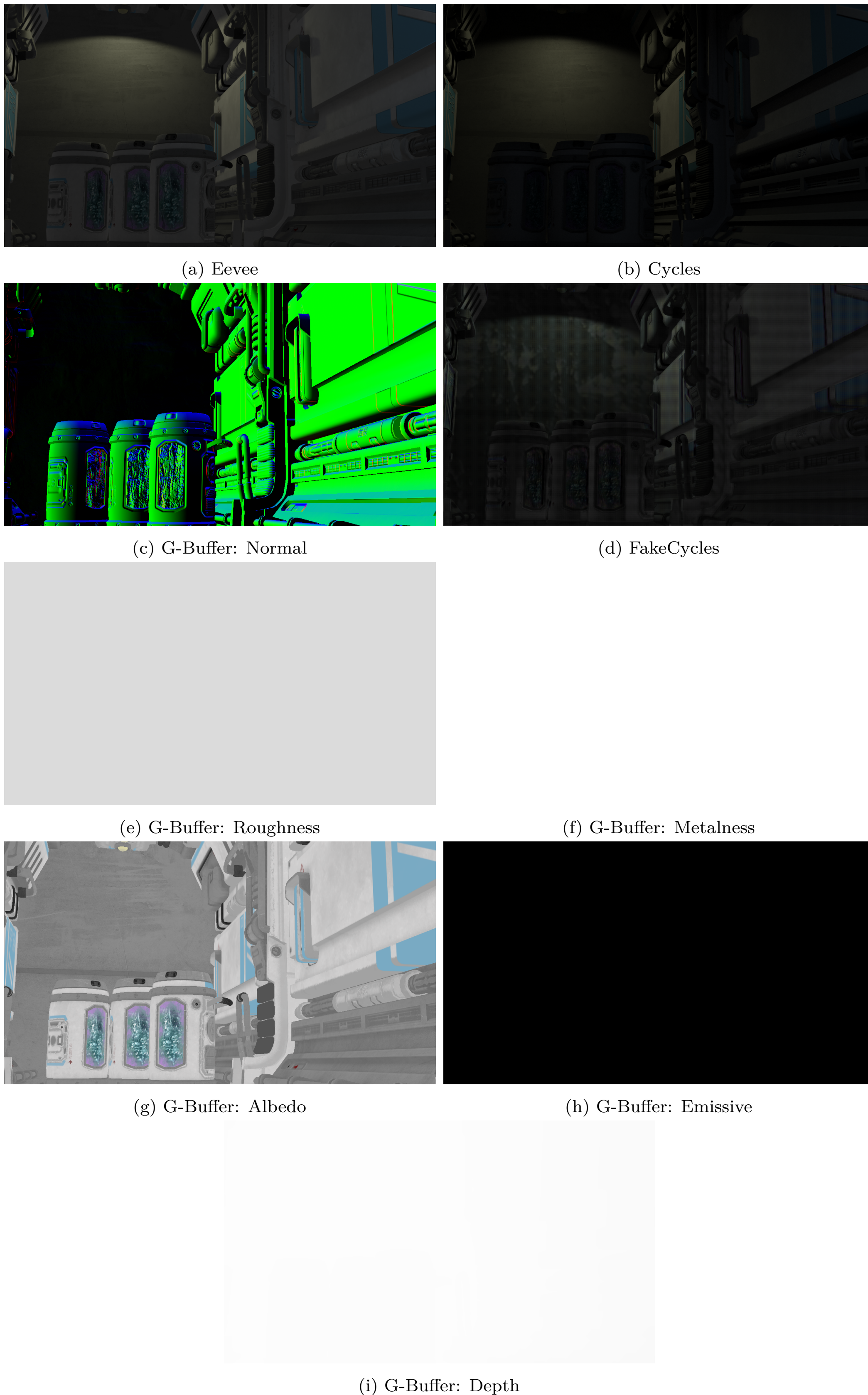
(f) G-Buffer: Metalness

(g) G-Buffer: Albedo

(h) G-Buffer: Emissive

(i) G-Buffer: Depth

Figura 4.20.: Caso particolare 1 scena 11



(a) Eevee

(b) Cycles

(c) G-Buffer: Normal

(d) FakeCycles

(e) G-Buffer: Roughness

(f) G-Buffer: Metalness

(g) G-Buffer: Albedo

(h) G-Buffer: Emissive

(i) G-Buffer: Depth

Figura 4.21.: Caso particolare 2 scena 11

CASO	**CONSIDERAZIONI**
Normale	La rete riesce ad esaltare la tridimensionalità di alcune scene in cui mancano sorgenti luminose. Apparente utilizzo preponderante delle normali per accentuazione dell'effetto tridimensionale. Non riesce però a gestire correttamente i materiali riflettenti, non creando nessun tipo di riflesso. Viene introdotto un pattern di rumore su tutta l'immagine, con particolare attenzione ai materiali emissivi. Viene creata l'occlusione ambientale, anche se spesso impropriamente. Generalmente si vede come la rete operi anche un inscurimento dell'immagine corretto. Molto importante risulta essere la constatazione che anche colori e forme vengono preservati nella maggior parte dei casi.
Favorevole	Considerando casi in cui la scena risulta composta solo da materiali diffusivi, i limiti della rete risultano presenti anche se molto meno evidenti. Grazie alla ricostruzione della geometria che opera la rete, unita all'occlusione ambientale, il risultato complessivamente è un miglioramento rispetto all'immagine di partenza.
Particolare	In alcuni casi la rete mostra un comportamento articolato, riuscendo a modificare le texture di alcuni oggetti nell'immagine, ad aggiungere dettagli e ricostruire la geometria in maniera credibile ma non accurata rispetto alle informazioni fornite. A fronte di un apparente utilizzo triviale delle normali, grazie a questi casi si capisce che la rete non si limita ad usare un solo G-Buffer, anzi a volte crea dettagli dal nulla, senza avere uno specifico riferimento riscontrabile nell'input.

Figura 4.22.: Tabella riepilogativa sui risultati

Capitolo 5.

Conclusioni e Sviluppi Futuri

5.1. Conclusioni

In questa tesi sono stati presentati due macro lavori. Il primo riguarda la creazione di strumenti utili(3.1.2) per generare dei dataset propedeutici alla ricerca nell'ambito della Computer Graphics. Questi semplificano la vita dei ricercatori e mettono loro a disposizione dei meccanismi automatici per creare grandi quantità di dati con uno sforzo minimo. È stato necessario sviluppare questi tool proprio perchè lo sviluppo della rete neurale si basa sui dati a disposizione. Il secondo lavoro riguarda la verifica della validità di un approccio alternativo, a quelli presenti nello stato dell'arte, per il task di miglioramento dell'Image Quality dei render. Questo metodo si è rivelato poco efficace nel generare effetti di riflessione, di bloom, al punto da distorcere completamente i colori[1] nell'immagine nelle parti in cui sono presenti oggetti emissivi. Di contro si nota un buon comportamento per quanto riguarda scene contenenti solo oggetti con materiali aventi caratteristiche fisiche quasi totalmente diffusive. In tale circorstanza la rete si dimostra in grado di generare efficacemente l'occlusione ambientale, che dona maggiore profondità e geometria a immagini che senza questo miglioramento risulterebbero totalmente piatte, senza la possibilità nemmeno di distinguere le forme delle mesh presenti nell'inquadratura. Purtroppo però anche in questa circostanza il modello introduce una sorta di pattern, un artefatto che complessivamente limita molto la qualità percepita dall'utente. In conclusione si può affermare che l'approccio non dia i risultati sperati e che quindi vanno fatti dei cambiamenti sostanziali per poter raggiungere l'obiettivo di migliorare la qualità dei render.

5.2. Sviluppi Futuri

Gli sviluppi futuri possono essere molteplici, ma si sono individuate 3 papabili direzioni da seguire:

- l'utilizzo in maniera differente dei G-Buffer all'interno della rete, cercando di integrarli meglio all'interno della struttura e non solo come input.

[1] al punto di peggiore l'immagine di partenza.

- l'utilizzo di informazioni che non si limitino semplicemente all'inquadratura ma cercare di codificare, in un formato utilizzabile dal modello, tutta la scena 3D ed ogni sorgente di illuminazione al suo interno [2]
- invece di lavorare sulla rappresentazione finale dell'immagine[3], si può pensare di modificare o generare il render in uno spazio di rappresentazione intermedia, in un modello **Encoder-Decoder**.

Il primo possibile sviluppo individuato segue l'idea che con gli stessi dati, ma una struttura più articolata della rete si può arrivare a far estrarre al modello delle feature dai G-Buffer che fuse meglio tra loro possano portare ad ottenere un buon risultato. Quindi che ci sia margine di miglioramento su ciò che è stato proposto nella tesi.

La seconda direzione, si basa sulla premessa che i G-Buffer non siano sufficienti per replicare l'illuminazione globale in maniera soddisfacente. Ne deriva che bisogna trovare una codifica per poter utilizzare l'informazione su tutta la scena 3D, e anche tutte le sorgenti di illuminazione presenti in essa. Si potrebbe pensare di creare una **CubeMap** [4] attorno all'osservatore, così da avere le informazioni dei G-Buffer in ogni possibile direzione di vista. Ciò fornirebbe ulteriori informazioni utili sulla scena che circonda l'osservatore, descrivendo quindi tutto il contesto che ha intorno. In questo modo si supererebbero anche le limitazioni di un approccio *ScreenSpace* accennate in 3.2.1.

L'ultima, e con tutta probabilità la più promettente, possibilità di approfondimento, si basa sull'ipotesi che la rappresentazione dell'immagine in un formato colore, come matrice di pixel, non sia ideale per generare trasformazioni dell'immagine che vadano ad aggiungere informazioni come riflessi, bloom dei materiali emissivi, ed altri effetti grafici. Si potrebbe provare a lavorare su uno spazio di rappresentazione intermedio, più astratto, che possa permettere di modificare l'aspetto dell'immagine senza però ritoccarla direttamente pixel per pixel. Questo si può eventualmente realizzare tramite una architettura **Encoder-Decoder**, che implementi una funzione di identità nel dominio delle immagini renderizzate.

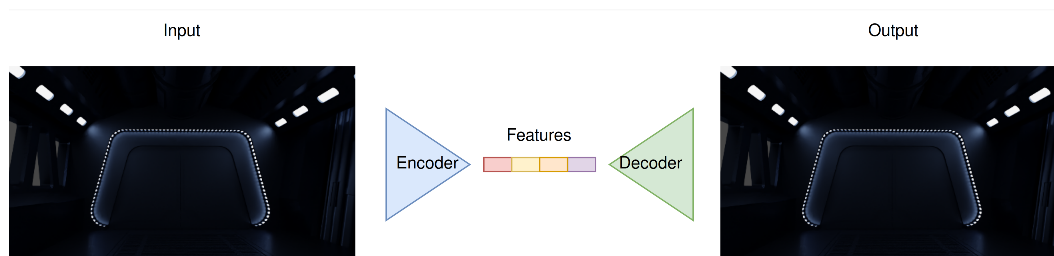


Figura 5.1.: Funzione identità tramite Enc-Dec

[2] non facendo distinzione tra sorgenti luminose e oggetti con texture emissive, che in un algoritmo offline vengono in ogni caso considerati fonti di luce.

[3] cioè la matrice di pixel RGB o HSV

[4] spiegata in A.1.2

Estrapolando poi le feature che si vanno a creare dopo la fase di Encoding, si avrebbe una rappresentazione intermedia, interna al modello. Questa si troverebbe in uno spazio di feature latente che permetta di compattare la descrizione di un'immagine[5], e che garantisca che una modifica fatta su questo descrittore astratto, si proietti anche in maniera controllata sull'output della fase di Decoding. Un approccio affine a questo anche se applicato in maniera totalmente diversa è quello relativo al lavoro di Riegler et al. [16] in cui si parte proprio da un vettore di feature per generare una nuova vista dell'immagine (lo schema del loro procedimento è riassunto nell'immagine in figura 5.3 presa direttamente dal paper [16]).

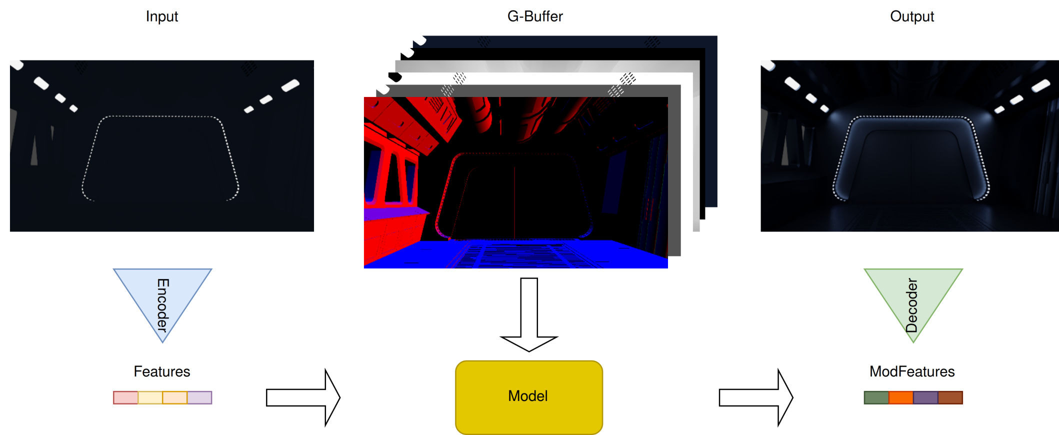
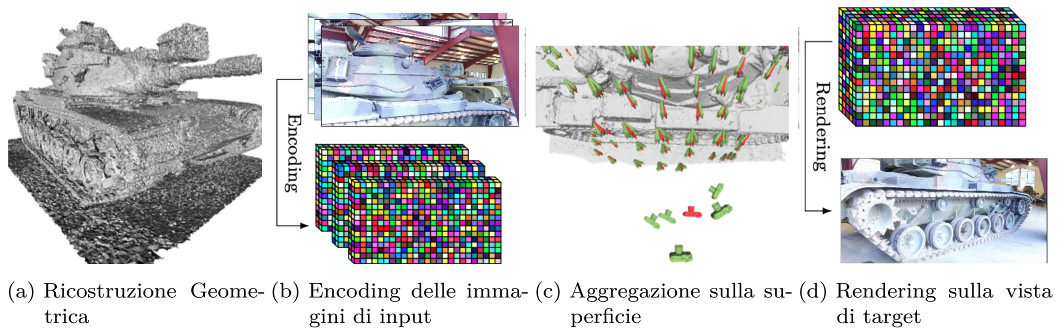


Figura 5.2.: Schema modello con Enc-Dec



(a) Ricostruzione Geometrica (b) Encoding delle immagini di input (c) Aggregazione sulla superficie (d) Rendering sulla vista di target

Figura 5.3.: Schema riassuntivo Stable View Synthesis.

Brevemente, la figura 5.3 riassume in questo modo:

- 5.3a Viene costruita la geometria della scena attraverso structure-from-motion, multiple-view stereo e meshing.
- 5.3b Tutte le immagini sorgente sono codificate in vettori di features tramite una CNN.

[5] volendo si può vedere come un sistema di compressione che sulla carta dovrebbe essere quasi lossless.

- 5.3c Dato un nuovo punto di vista (camera rossa), i vettori di feature delle immagini di input (camere verdi), sono aggregati sulla struttura geometrica.
- 5.3d Infine l'immagine di output della vista di target viene generata tramite una CNN a partire dal vettore di feature sintetizzato nel passo precedente.

Un ulteriore proposta può essere avanzata perchè non risulta banale nemmeno allenare la funzione identità, si potrebbe provare attraverso uno schema di allenamento come in figura 5.4.

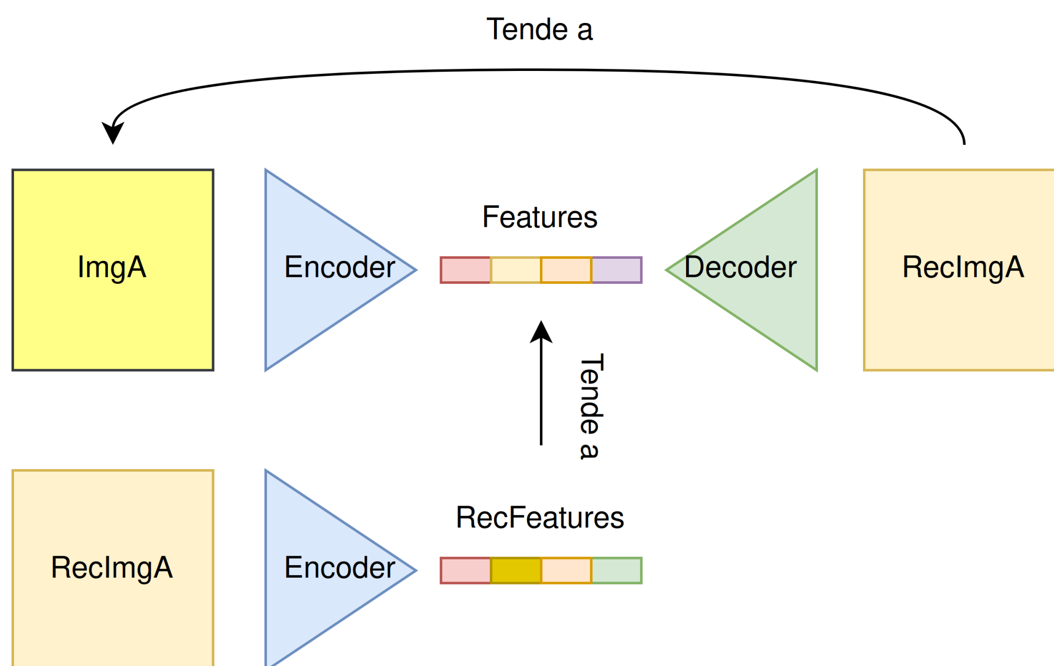


Figura 5.4.: Allenamento Ciclico per la funzione identità Enc-Dec

Infine per agevolare ancora di più la convergenza di questo ciclo, si può valutare l'allenamento in contemporeanea su più immagini diverse, andando a inserire anche una metrica per separarle dal punto di vista delle feature generate[6] tra loro, per evitare che tutto il ciclo collassi su di una soluzione triviale.

[6]anche una piccola differenza tra le due immagini è importante che si traduca nell'ottenere un vettore di feature unico e distinguibile per ognuna di esse.

Appendice A.

Appendice

A.1. Computer Graphics

A.1.1. OpenGL

OpenGL, è una API multilinguaggio, che detta le specifiche per l'utilizzo dell'hardware grafico. Si utilizza per creare applicazioni 2D/3D altamente performanti e permette di astrarre lo sviluppo software su GPU, grazie alla sua natura di specifica, rendendo il processo di creazione degli applicativi indipendente dall'hardware sottostante. In OpenGL, sono presenti i dettagli di cosa ogni funzione realizza, poi sono i singoli costruttori di hardware a doversi far carico della loro implementazione.

A.1.2. Cubemap

Il termine CubeMap deriva dalla tecnica di cube mapping. Essa viene spesso usata per creare effetti di riflessioni nella computer graphics. Come dice il nome, consiste nel fare il rendering di una scena 6 volte da un unico punto di vista, come se l'osservatore si trovasse all'interno di un cubo, ed ogni volta guardasse una faccia diversa del cubo. Quindi tutto l'ambiente che circonda l'osservatore viene proiettato sulle sei facce quadrate che lo circondano. Questa tecnica permette appunto di generare riflessioni, di calcolare le ombre derivanti da una sorgente luminosa che irradia in maniera sferica la luce, o anche per creare uno skybox[1].

A.1.3. Illuminazione locale vs globale

Una parte fondamentale della pipeline di rendering è quella relativa allo shading. Lo shading ha il compito di definire il colore di ogni pixel dell'immagine e questo avviene calcolando l'equazione di illuminazione per ogni superficie[2] colpita dalla luce emessa da almeno una sorgente luminosa. Gli algortimi di illuminazione locale per calcolare il colore di un pixel associato ad una superficie illuminata, considerano solo ed esclusivamente la sorgente luminosa, le caratteristiche del materiale della superficie e la posizione dell'osservatore[3]. Non viene considerata in alcun modo la

[1] cioè un background tridimensionale a tutta la scena
[2] è una semplificazione concettuale del processo che avviene realmente.
[3] necessaria per definire l'esistenza di eventuali riflessi.

presenza di altri oggetti nella scena e non si generano effetti di ombre o illuminazione indiretta[4]. Al contrario le tecniche di illuminazione globale prendono in considerazione l'interazione che la luce ha anche con gli altri oggetti, quindi si vengono a creare ombre, riflessioni, illuminazione indiretta, occlusione ambientale e tutti gli altri effetti che la luce crea nella realtà[5].

A.1.4. Physically Based Rendering

Physically Based Rendering o PBR, consiste in una serie di tecniche che permettono di creare un render simulando in maniera credibile il modo in cui la luce interagisce con il mondo reale. Ci sono molti possibili modelli di illuminazione per il PBR, in generale però un modello per essere considerato *physical based* è necessario che soddisfi 3 condizioni:

- Deve essere basato sul modello delle **Microfacet**.
- Deve rispettare il principio di conservazione dell'energia.
- Devono usare una funzione di **BRDF** physical based.

Il modello Microfacet descrive ogni superficie di qualsiasi oggetto, come composta da una serie di microscopiche superfici piatte che sono dei perfetti specchi (figura A.1). In base a quanto queste microsuperfici sono orientate caoticamente[6], si ha che la superficie sarà più o meno diffusiva.

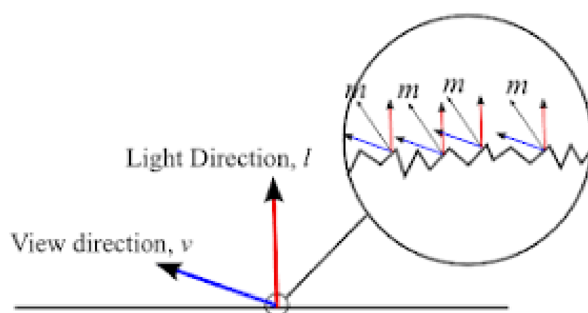


Figura A.1.: Modello Microfacet.

L'allineamento delle microfacet si può statisticamente approssimare attraverso il parametro di **Roughness** che va da 0 a 1. Con il valore minimo si ha un materiale in cui le riflessioni saranno concentrate in una piccola area e saranno più definite e brillanti, mentre con il massimo di roughness si avranno i riflessi distribuiti su un'area più ampia e saranno di minore intensità, come in fig A.2.

[4]senza nessun trucco per simularla, le parti non direttamente illuminate dalla sorgente di luce risulterebbero completamente nere.
[5]anche se non riprodotti con una fedeletà assoluta, dato che si tratta in ogni caso di approssimazioni
[6]cioè disallineate tra loro.

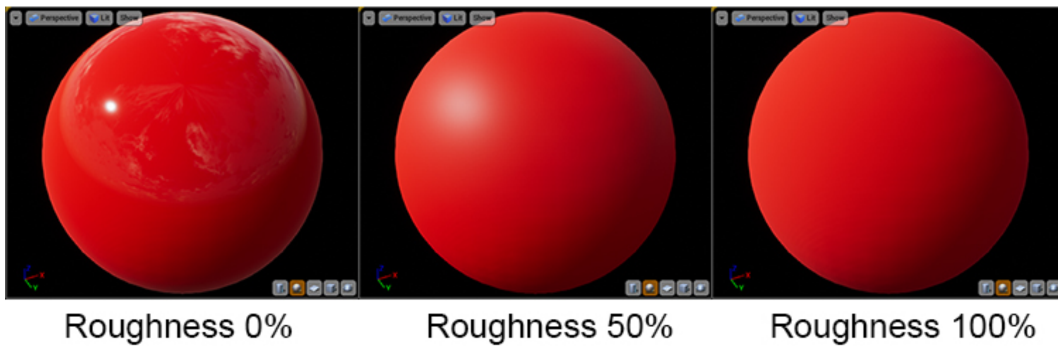


Figura A.2.: Esempio dell'influenza della Roughness sul materiale.

Chiaramente questo modello utilizza una forma di conservazione dell'energia: cioè l'energia luminosa che incide su una superficie deve essere sempre maggiore o uguale a quella che ne esce (escluso per materiali emissivi). Guardando l'immagine in figura A.2, si vede come al variare dell'allineamento delle microfacet, si ottiene che viene distribuita in maniera più uniforme l'energia luminosa che viene riflessa ottenendo però una minore luminosità di ogni singola parte della sfera. Se si fosse ottenuto un riflesso di pari entità a quello della roughness a 0, però distribuito su tutta la sfera, si sarebbe violato il principio di conservazione dell'energia, dato che a quel punto la superficie avrebbe emesso energia e non riflesso quella in entrata. Inoltre è importante fare una distinzione tra luce diffusa e speculare. Questo perchè quando un raggio di luce colpisce una superficie, una parte di esso viene riflessa immediatamente, l'altra parte viene rifratta all'interno del materiale. La parte che viene direttamente riflessa prende il nome di **luce speculare**, mentre quella che entra nella superficie e viene assorbita, prende il nome di **luce diffusa**. Solitamente non tutta l'energia luminosa che penetra nelle superfici viene assorbita, anzi, continua il suo percorso collidendo all'interno finchè non si esaurisce l'energia oppure finchè non viene rimbalzata nuovamente di fuori, quest'ultima contribuisce a definire il colore della superficie. Generalmente però l'effetto dello scatter interno della luce viene ignorato[7], e si semplifica considerando che tutta la luce rifratta viene assobita e viene diffusa su una piccola area attorno al punto in cui colpisce il raggio luminoso. Una considerazione a parte va fatta per le superfici metalliche, dato che reagiscono in maniera differente alla luce, dato che tutta la luce rifratta viene assorbita senza nessun tipo di diffusione correlata. Quindi per le superfici metalliche si considera solo la luce speculare, e perciò essi non hanno nessun colore generato dalla diffusione dei raggi sulla superficie. Chiaramente tornando al discorso della conservazione dell'energia, la luce riflessa e quella rifratta sono mutualmente esclusive, cioè nel calcolo dell'illuminazione, la parte che viene riflessa viene sottratta dal totale della luce che collide con la superficie, e quella che rimane è luce rifratta. Tutti questi concetti vengono racchiusi dall'equazione chiamata **reflectance equation**:

[7] ci sono tecniche di rendering che considerano questo effetto, chiamato **subsurface scattering**

$$L_o(p, \omega_o) = \int_\Omega f_r(p, \omega_i, \omega_o) L_i(p, \omega_i) n \cdot \omega_i d\omega_i$$

Questa equazione viene espressa in termini radiometrici. Le grandezze al suo interno sono:

- la radianza che si denota con L[8]
- l'emisfero centrato sulla normale, che si denota con Ω
- p, il punto infinitesimale colpito dalla luce, e da cui viene riflessa la luce
- ω_o e ω_i che sono rispettivamente, l'angolo solido verso cui viene proiettata la luce e quello da cui proviene la luce[9]
- n, cioè la normale nel punto p
- f_r, l'equazione di BRDF

Questa equazione descrive la luce che si irradia da un punto p, verso la direzione di vista dell'osservatore ω_o, e si calcola attraverso la somma[10], della radianza della luce proveniente da tutte le direzioni ω_i, all'interno dell'emisfero Ω, scalata di un fattore f_r.

Per concludere la descrizione di questa equazione è fondamentale parlare della **BRDF**, cioè la *bidirectional reflective distribution function*. Questa prende in input la direzione ω_i della luce incidente su p, la direzione verso la quale si sta valutando la radianza ω_o, la normale alla superficie n e a che rappresenta il parametro relativo alla roughness della microsuperficie. Tale funzione approssima quanto di ogni singolo raggio ω_i contribuisce alla luce finale riflessa verso l'osservatore, in base alle proprietà del materiale. Per essere fisicamente plausibile la BRDF deve rispettare la legge di conservazione dell'energia. Tra le tante esistenti, la più utilizzata è la Cook-Torrance BRDF.

$$f_r = k_d f_{\text{lambert}} + k_s f_{\text{cook-torrance}}$$

Ricordando che $k_d + k_s = 1$, per la conservazione dell'energia, in cui k_d e k_s sono rispettivamente la percentuale di luce rifratta e riflessa. In questo caso la f_{lambert} viene approssimata piuttosto semplicemente con:

$$f_{\text{lambert}} = \frac{c}{\pi}$$

[8] serve per quantificare la forza della luce che arriva da una specifica direzione.

[9] entrambi questi angoli solidi vengono considerati infinitesimali e possono considerarsi delle direzioni e non delle aree.

[10] in questo caso l'integrale

In cui c è l'albedo, cioè il colore della superficie. Più complesso risulta il discorso legato alla parte speculare, che viene espressa come:

$$f_{\text{cook-torrance}} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

Questa è composta da 3 funzioni principali D, F e G , che approssimano una parte specifica delle proprietà di riflettività di una superficie.

- D, la **normal distribution function**, che approssima la quantità di microfacet che sono allineate con l'**half-way vector**[11], questa è influenzata dalla roughness del materiale.
- F, l'**equazione di Fresnel**, che descrive quanto una superficie è riflettente a diversi angoli di incidenza della luce.
- G, la **geometry function** che descrive la proprietà di *self-shadowing*[12] dei microfacet, anche questa proprietà dipende molto dalla roughness del materiale.

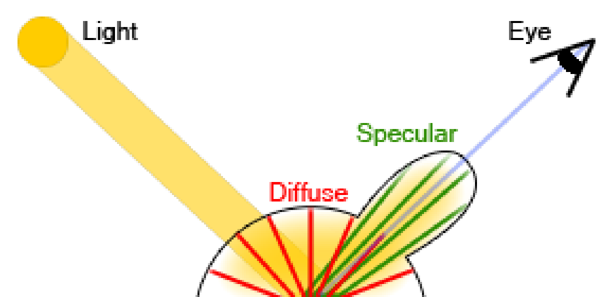


Figura A.3.: Esempio di BRDF.

A.1.5. Deferred Shading vs Forward Shading

Il rendering è un processo che viene compiuto in più stage, sotto forma di pipeline. Questo comprende una serie di passaggi riassumibili in fig A.4, che servono per passare da una rappresentazione geometrica di una scena tridimensionale, ad una serie di pixel colorati che rappresentano l'immagine finale che si vuole ottenere.

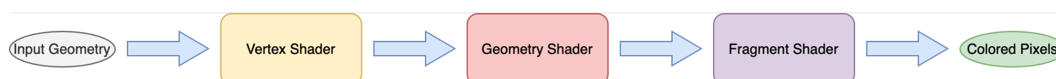


Figura A.4.: Schema esemplificativo di una moderna pipeline di rendering.

[11] è il vettore direzione che si trova a metà tra la direzione di vista e la direzione della luce, su di un punto.
[12] cioè quanto le microfacet tra di loro si fanno "ombra", a causa dell'allineamento caotico.

In questo contesto il **Forward Rendering** consiste nell'eseguire direttamente tutta la pipeline per ogni mesh[13] fino ad ottenere i pixel colorati nell'immagine per ogni oggetto [14]. Il funzionamento è illustrato in figura A.5.

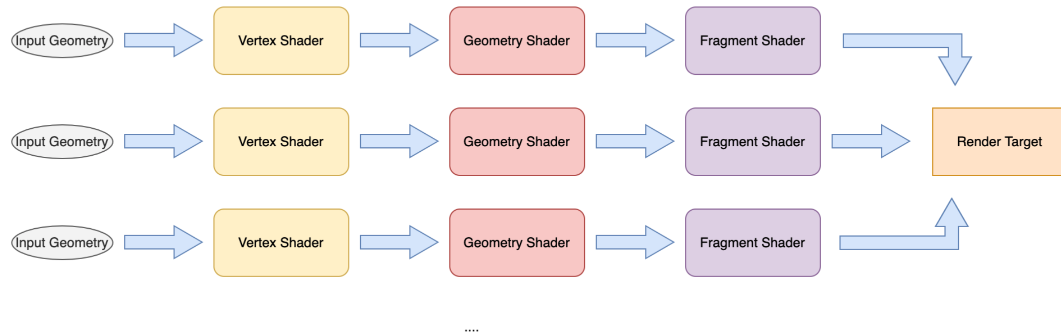


Figura A.5.: Forward Rendering.

Dato che il fragment shader effettua i calcoli per l'illuminazione degli oggetti, è chiaro che molti dei calcoli che vengono fatti, cioè quelli sui pixel che poi non verranno visualizzati, non sono necessari e rallentano tutto il processo. Questo è il grande limite di tale tecnica. Il **Deferred Rendering** di contro, divide il processo di rendering in due parti, nella prima parte viene processata tutta l'informazione riguardante la geometria degli oggetti, così da arrivare ad ottenere delle informazioni preliminari (sarebbero i G-Buffer in A.1.5.1) per poter effettuare il calcolo dell'illuminazione e quindi del colore finale dei pixel dell'immagine, una sola volta. Lo schema in figura A.6 illustra quanto spiegato, i *render target* multipli sono i G-Buffer.

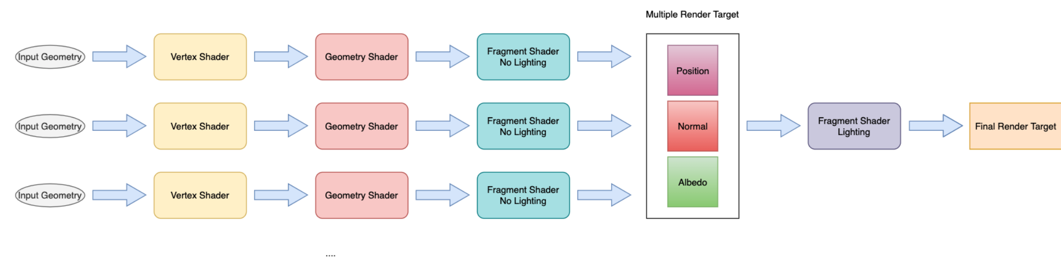


Figura A.6.: Deferred Rendering.

Chiaramente lo scopo di tutto questo, è ridurre sensibilmente il numero di calcoli per il processo di **lighting**[15], dato che non viene eseguito più separatamente per tutti gli oggetti e per ogni parte dell'oggetto, bensì viene eseguito solo sulle parti visibili di un oggetto che sono codificate all'interno dei G-Buffer.

[13] la mesh è la rappresentazione più usata per gli oggetti 3D, consiste nella definizione dei vertici che formano l'oggetto e delle facce che questi compongono.

[14] poi considerando la distanza dell'oggetto rispetto alla camera si deciderà quali pixel visualizzare e quali no, in base a questa misura verranno scartati i pixel che appartengono a parti dell'oggetto che risultano coperte da altre mesh più vicine alla camera, sulla direzione di vista.

[15] cioè il calcolo dell'illuminazione degli oggetti.

A.1.5.1. G-Buffer

Come accennato in A.1.5, queste immagini sono un prodotto intermedio della pipeline del *deferred shading.* Ognuno di questi rappresenta una caratteristica puntuale dell'immagine, cioè per ogni pixel è definita una caratteristica in base al G-Buffer in cui si trova. Ad esempio nel G-Buffer relativo alle normali, ogni pixel consiste in tre valori che rappresentano l'orientamento della normale in quel punto. Questi vengono generati attraverso una normale pipeline di rendering come quella in figura A.5, però il fragment shader in quel caso non effettua il lighting, ma si limita a scrivere su tutti i G-Buffer in contemporanea[16] , l'informazione relativa ad ognuno di essi. Queste poi verranno usate per calcolare l'illuminazione su tutta l'immagine, ma come si è potuto intendere, questa verrà computata solo per quello che la camera effettivamente vede e non anche per le parti coperte degli oggetti.

[16]perciò si parla di Multiple Render Target.

A.2. Blender

Blender è un software gratuito e open source per la creazione di contenuti 3D e 2D. Supporta tutta una serie di funzioni fondamentali per gli artisti che lavorano in questo ambito:

- Modellazione
- Rigging
- Animazione
- Simulazione
- Rendering
- ...

Oltre a questo, grazie alla sua natura open, chiunque può incrementare le feature del programma sviluppando componenti aggiuntivi. Infatti è presente una corposa documentazione che spiega il funzionamento di tutto Blender e come creare addOn. Il tutto si realizza tramite la Python API A.2.1.

A.2.1. Python API

Blender ha all'interno un suo interprete python, che viene caricato quando si avvia il programma. Questo interprete si occupa di realizzare tutto ciò che riguarda l'interfaccia e le funzioni all'interno di Blender. Quindi questo rimane sempre attivo in background in attesa che una qualsiasi funzione venga richiamata, per poterla eseguire. Tutto il codice python può essere eseguito tramite questo interprete, e l'unica particolarità che ha risiede nei moduli **bpy** e **mathutils** che permettono di accedere ai dati di Blender, alle classi e alle funzioni. Lo sviluppo degli addOn passa proprio dall'importare questi due moduli, dopodichè si può accedere a tutto quello che c'è dentro il programma e si può integrare il componente aggiuntivo nell'interfaccia estendendo le classi di riferimento.

A.2.2. Eevee vs Cycles

Eevee è il renderer real-time di Blender, fatto per la velocità e interattività pur mantenendo il rendering di materiali PBR. Può essere usato nella viewport per avere un'anteprima interattiva, ma anche per avere render finali di alta qualità.

Cycles è un path tracer PBR, per ottenere la massima qualità possibile del rendering finale in termini di accuratezza della resa dei materiali e dell'interazione della luce tra gli oggetti della scena.

Eevee e Cycles usano gli stessi shader PBR per i materiali. Eevee al contrario di cycles non usa ray tracing quindi fa rasterizzazione classica. Dato questo, eevee ha

bisogno di utilizzare una serie di algoritmi aggiuntivi per poter simulare come la luce interagisce tra gli oggetti con diversi materiali. La creazione di questi ultimi risulta essere molto dispendiosa in termini di tempo e richiede uno sforzo quasi totalmente manuale per raggiungere dei risultati visivamente credibili.

A.3. HSV

HSV è un modello di colore percettivo cilindrico, che rende la rappresentazione dei colori più semplice da comprendere per gli esseri umani. Le tre dimensioni sono:

- Hue: specifica l'angolo che definisce il colore vero, la tinta.
- Saturation: controlla la quantità di colore da usare. 100% corrisponde al colore puro, 0% crea una scala di grigio.
- Value: controlla la brillantezza del colore. Cioè un colore con 0% è nero completamente, mentre con il valore a 100% non c'è nessuna parte di nero mixata con il colore.

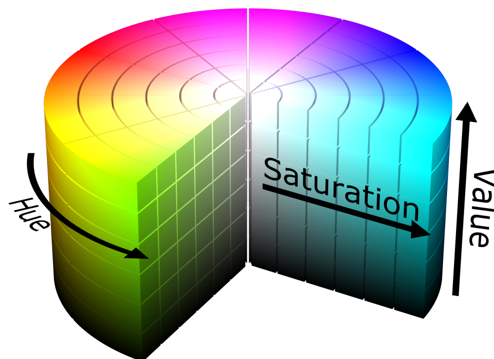


Figura A.7.: Cilindro HSV.

A.4. Deep Learning

A.4.1. Squeeze and Excitation

È un meccanismo di self-attention che permette di pesare in maniera diversa i canali o kernel di un tensore.

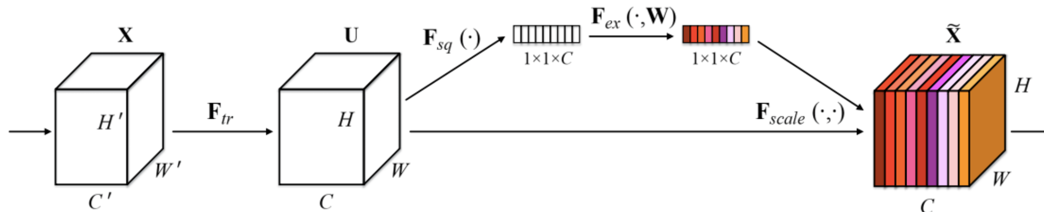


Figura A.8.: Schema dello Squeeze and Excitation

In sostanza, a tutti i kernel in un filtro viene tradizionalmente data uguale importanza (vale a dire, viene calcolata la media del risultato di ciascuna operazione del kernel). Ma squeeze-and-excitation (SE) assegna pesi ai kernel (o canali, a seconda del punto di vista), quindi un kernel/canale potrebbe avere un'influenza maggiore di un altro su come appare la successiva mappa di attivazione. Lo fa comprimendo una feature map C*H*W in un vettore C-dimensionale tramite il raggruppamento, seguito da un layer lineare e una funzione di attivazione che riduce la dimensione a C/r (r è un numero naturale specificato dall'utente), e infine, un altro layer lineare seguito dalla funzione Sigmoid che restituisce i valori C. La pesatura non è costante ma dinamica (cioè input diversi producono pesi diversi). Questo è fondamentale perché i pesi costanti sono, almeno in teoria, inutili poiché i valori nei kernel associati a un canale potrebbero semplicemente apprendere quei pesi. [17].

A.4.2. MBConv

MBConv è un elemento fondante introdotto nella MobileNetv2 [41]. È un blocco definito *Inverted Residual* composto con una logica **narrow->wide->narrow**, cioè va prima di tutto ad espandere il numero di canali attraverso **Pointwise Convolution**[18], dopodichè applica una **Depthwise Convolution**, poi un blocco SE ed infine si comprime di nuovo il numero di canali riducendoli sempre attraverso una **Pointwise Convolution**. Poi se il numero di canali di input corrisponde con il numero di canali di output, si somma l'input con l'output[19]. Come si vedrà in 3.2.2.1 qeusta definizione generale si prende forma in due blocchi generalmente:

- MBConv1 : in questo caso il fattore di espansione del numero di canali è 1 quindi in realtà la prima *Pointwise Convolution* non viene eseguita.

[17] riferimento a Spiegazione Squeeze and Excitation
[18] usando un kernel convoluzionale di dimensione 1x1.
[19] come qualsiasi altro blocco residuale

- MBConv6 : in questo caso il fattore di espansione del numero di canali è di 6, come si vede in figura A.9.

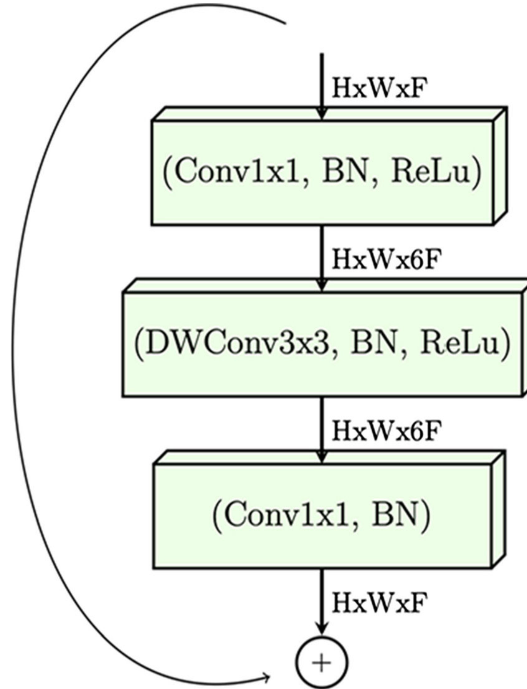


Figura A.9.: Configurazione MBConv6.

La particolare struttura di questo blocco è dovuta principalmente ad una notevole differenza in termini di efficienza rispetto alle classiche convoluzioni 2D. Perchè con Depthwise e Pointwise convolution, si possono ottenere gli stessi risultati che si possono ottenere con le convoluzioni normali, ma con un numero di operazioni matematica estremamente inferiore, inoltre hanno il vantaggio[20] di poter avere una maggiore profondità della rete dato che introducono un numero piuttosto basso di parametri rispetto alle convoluzioni classiche.

[20] o svantaggio, dipende se la rete ha già un numero sufficiente di parametri.

A.5. Formato Wavefront .OBJ

OBJ è un formato open, che rappresenta in maniera semplice la geometria 3D, riporta le posizioni di ogni vertice, la coordinata UV di ogni vertice, le normali, e le facce che rendono ogni poligono definito come una lista di vertici e di coordinate UV relative.

```
# List of geometric vertices, with (x, y, z [,w]) coordinates, w is optional and defaults to 1.0.
v 0.123 0.234 0.345 1.0
v ...
...
# List of texture coordinates, in (u, [,v ,w]) coordinates, these will vary between 0 and 1. v, w are optional and default to 0.
vt 0.500 1 [0]
vt ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Parameter space vertices in ( u [,v] [,w] ) form; free form geometry statement ( see below )
vp 0.310000 3.210000 2.100000
vp ...
...
# Polygonal face element (see below)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
# Line element (see below)
l 5 8 1 2 4 9
```

Figura A.10.: Esempio di file OBJ.

In aggiunta a questa definizione puramente geometrica, troviamo solitamente almeno un file MTL. Questo contiene il dettaglio sui materiali che descrivono l'aspetto esteriore dei poligoni della mesh.

```
newmtl Textured
   Ka 1.000 1.000 1.000
   Kd 1.000 1.000 1.000
   Ks 0.000 0.000 0.000
   d 1.0
   illum 2
   # the ambient texture map
   map_Ka lemur.tga

   # the diffuse texture map (most of the time, it will be the same as the
   # ambient texture map)
   map_Kd lemur.tga

   # specular color texture map
   map_Ks lemur.tga

   # specular highlight component
   map_Ns lemur_spec.tga

   # the alpha texture map
   map_d lemur_alpha.tga

   # some implementations use 'map_bump' instead of 'bump' below
   map_bump lemur_bump.tga

   # bump map (which by default uses luminance channel of the image)
   bump lemur_bump.tga

   # displacement map
   disp lemur_disp.tga

   # stencil decal texture (defaults to 'matte' channel of the image)
   decal lemur_stencil.tga
```

Figura A.11.: Esempio di file MTL.

Bibliografia

[1] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[2] Qifeng Chen and Vladlen Koltun. Photographic image synthesis with cascaded refinement networks. *CoRR*, abs/1707.09405, 2017.

[3] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans. *CoRR*, abs/1711.11585, 2017.

[4] Peihao Zhu, Rameen Abdal, Yipeng Qin, and Peter Wonka. SEAN: image synthesis with semantic region-adaptive normalization. *CoRR*, abs/1911.12861, 2019.

[5] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. *CoRR*, abs/1903.07291, 2019.

[6] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Guilin Liu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. Video-to-video synthesis. *CoRR*, abs/1808.06601, 2018.

[7] Liming Jiang, Changxu Zhang, Mingyang Huang, Chunxiao Liu, Jianping Shi, and Chen Change Loy. TSIT: A simple and versatile framework for image-to-image translation. *CoRR*, abs/2007.12072, 2020.

[8] Paul Debevec, Camillo Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Proc. of SIGGRAPH 96*, 96, 09 2001.

[9] Harry Shum and Sing Bing Kang. A review of image-based rendering techniques. volume 4067, pages 2–13, 05 2000.

[10] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface light fields for 3d photography. SIGGRAPH '00, page 287–296, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[11] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *Proceedings of the 28th Annual*

Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01, page 425–432, New York, NY, USA, 2001. Association for Computing Machinery.

[12] Johannes Kopf, Michael F. Cohen, and Richard Szeliski. First-person hyper-lapse videos. *ACM Trans. Graph.*, 33(4), July 2014.

[13] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. 37(6):257:1–257:15, 2018.

[14] Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew DuVall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. Immersive light field video with a layered mesh representation. 39(4):86:1–86:15, 2020.

[15] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *CoRR*, abs/2003.08934, 2020.

[16] Gernot Riegler and Vladlen Koltun. Stable view synthesis. *CoRR*, abs/2011.07233, 2020.

[17] Sai Bi, Kalyan Sunkavalli, Federico Perazzi, Eli Shechtman, Vladimir G. Kim, and Ravi Ramamoorthi. Deep cg2real: Synthetic-to-real translation via image disentanglement. *CoRR*, abs/2003.12649, 2020.

[18] Jing Liao, Yuan Yao, Lu Yuan, Gang Hua, and Sing Bing Kang. Visual attribute transfer through deep image analogy. *CoRR*, abs/1705.01088, 2017.

[19] E. Reinhard, M. Adhikhmin, B. Gooch, and P. Shirley. Color transfer between images. *IEEE Computer Graphics and Applications*, 21(5):34–41, 2001.

[20] Yijun Li, Chen Fang, Jimei Yang, Zhaowen Wang, Xin Lu, and Ming-Hsuan Yang. Universal style transfer via feature transforms. *CoRR*, abs/1705.08086, 2017.

[21] F. Pitie, A.C. Kokaram, and R. Dahyot. N-dimensional probability density function transfer and its application to color transfer. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1434–1439 Vol. 2, 2005.

[22] François Pitié, Anil C. Kokaram, and Rozenn Dahyot. Automated colour grading using colour distribution transfer. *Computer Vision and Image Understanding*, 107(1):123–137, 2007. Special issue on color image processing.

[23] Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. Deep photo style transfer. *CoRR*, abs/1703.07511, 2017.

[24] Roey Mechrez, Eli Shechtman, and Lihi Zelnik-Manor. Photorealistic style transfer with screened poisson equation. *CoRR*, abs/1709.09828, 2017.

[25] Ming-Yu Liu, Thomas M. Breuel, and Jan Kautz. Unsupervised image-to-image translation networks. *CoRR*, abs/1703.00848, 2017.

[26] Jaejun Yoo, Youngjung Uh, Sanghyuk Chun, Byeongkyu Kang, and Jung-Woo Ha. Photorealistic style transfer via wavelet transforms. *CoRR*, abs/1903.09760, 2019.

[27] Yijun Li, Ming-Yu Liu, Xueting Li, Ming-Hsuan Yang, and Jan Kautz. A closed-form solution to photorealistic image stylization. *CoRR*, abs/1802.06474, 2018.

[28] Xueting Li, Sifei Liu, Jan Kautz, and Ming-Hsuan Yang. Learning linear transformations for fast image and video style transfer. pages 3804–3812, 2019.

[29] Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A. Efros, and Trevor Darrell. Cycada: Cycle-consistent adversarial domain adaptation. *CoRR*, abs/1711.03213, 2017.

[30] Xun Huang, Ming-Yu Liu, Serge J. Belongie, and Jan Kautz. Multimodal unsupervised image-to-image translation. *CoRR*, abs/1804.04732, 2018.

[31] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, abs/1703.10593, 2017.

[32] Taesung Park, Alexei A. Efros, Richard Zhang, and Jun-Yan Zhu. Contrastive learning for unpaired image-to-image translation. *CoRR*, abs/2007.15651, 2020.

[33] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[34] Stephan R. Richter, Hassan Abu Alhaija, and Vladlen Koltun. Enhancing photorealism enhancement. *CoRR*, abs/2105.04619, 2021.

[35] Oliver Nalbach, Elena Arabadzhiyska, Dushyant Mehta, Hans-Peter Seidel, and Tobias Ritschel. Deep shading: Convolutional neural networks for screen-space shading. *CoRR*, abs/1603.06078, 2016.

[36] Hassan Abu Alhaija, Siva Karthik Mustikovela, Andreas Geiger, and Carsten Rother. Geometric image synthesis. *CoRR*, abs/1809.04696, 2018.

[37] Jia Zheng, Junfei Zhang, Jing Li, Rui Tang, Shenghua Gao, and Zihan Zhou. Structured3d: A large photo-realistic dataset for structured 3d modeling. *CoRR*, abs/1908.00222, 2019.

[38] Wenbin Li, Sajad Saeedi, John McCormac, Ronald Clark, Dimos Tzoumanikas, Qing Ye, Yuzhong Huang, Rui Tang, and Stefan Leutenegger. Interiornet: Mega-scale multi-sensor photo-realistic indoor scenes dataset. *CoRR*, abs/1809.00716, 2018.

[39] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[40] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. *CoRR*, abs/1801.03924, 2018.

[41] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

Ringraziamenti

Questa forse è la parte più difficile di tutta la tesi. Per una persona come me è sempre un po' complesso riuscire a ringraziare abbastanza la mia fidanzata Giorgia, i miei parenti e i miei amici. Sono questi però i momenti in cui si fa un passo avanti nella vita, finisce una parte e ne inizia un'altra e allora viene spontaneo fare dei bilanci. Io sono sempre un po' severo quando si tratta di fare delle valutazioni, ma generalmente solo con me stesso, perchè credo che difficilmente avrei potuto sperare di avere Giorgia al mio fianco, mia Madre Romana, mia Sorella Stella, i miei Fratelli Osvaldo e Tino. Non mi dimentico di te papà, accontentati della dedica però. Giorgia vorrei solo dirti che senza la tua presenza, la tua complicità e il tuo amore costante, non sarei riuscito ad affrontare quasi nulla di quello che è successo in due anni. Come posso dire che la vita mi ha tolto tanto ultimamente, posso dire che mi ha ricompensato donandomi te. La cosa ancora bella in tutto questo e che oltre a loro, che basterebbero per mille vite, ci sono un numero davvero ragguardevole di Amici, ed è incredibile pensare che sono Amici degni di essere appellati in tal modo. So che sono una persona difficile a volte, ma avete sempre saputo accettarmi e volermi bene comunque. Quindi non posso non citarvi tutti o quasi. Ivan e Lorenzo, voi mi avete visto arrivare fino a qua da tanto tempo davvero, non so nemmeno cosa dirvi in più perchè sapete già cosa penso. Francesco e Claudio, siete due fenomeni, senza di voi l'università non sarebbe stata la stessa, senza i vostri orari sballati, ma soprattutto senza i CD appesi sulla costa del conero(solo per chi c'era). Matteo e Giacomo, anche grazie a voi le domeniche ad Ancona hanno avuto un gusto diverso, di solito quello della norcina (ancora sto a rosica per la schedina vostra). Manuel e Paolo, siamo cresciuti insieme in tutti i sensi, anche se a volte lontani non si è mai affievolita l'amicizia. Antonio, Danilo e Riccardo, senza di voi e senza Prevenisco, le nostre radici, non sarei la persona che sono, Grazie. Tutti gli altri amici dell'Università, ragazzi, abbiamo passato davvero delle belle giornate insieme, anche senza fare chissà cosa siamo riusciti a divertirci sempre. Davide, ci conosciamo da relativamente poco ma ti ringrazio veramente tanto per tutto il supporto, spesso anche tecnico, meriti di stare a lavorare in Olanda, e so che farai grandi cose in futuro, sei un Amico. Tutti quelli che non ho citato direttamente non se la prendano, sappiate che se vi ho invitato alla festa di laurea vi voglio bene ugualmente. Poi vorrei ringraziare davvero molto il Professor Zingaretti che mi ha dato fiducia e mi ha permesso di realizzare (o perlomeno provare a realizzare) l'idea da cui è nata questa tesi. Infine vorrei ringraziare immensamente Marco, che mi ha seguito e ha cercato di aiutarmi in ogni modo possibile, togliendosi anche del tempo libero (rimane valido l'invito a bere).

Grazie anche a chi ha letto questa tesi, spero che le idee proposte e l'esposizione sia risultata chiara e interessante.