



**UNIVERSITÀ
DI TORINO**

Università degli Studi di Torino

Corso di Laurea Magistrale in Informatica

Titolo

Visualizzazione Immersiva e Interattiva di Volumi Medici
attraverso Ray Marching

Relatore

Grangetto Marco

Correlatore

Lucenteforte Maurizio

Candidato
Orrù Michael
Matricola 883850

Anno Accademico 2023/2024

DICHIARAZIONE DI ORIGINALITÀ

Dichiaro di essere responsabile del contenuto dell'elaborato che presento al fine del conseguimento del titolo, di non aver plagiato in tutto o in parte il lavoro prodotto da altri e di aver citato le fonti originali in modo congruente alle normative vigenti in materia di plagio e di diritto d'autore. Sono inoltre consapevole che nel caso la mia dichiarazione risultasse mendace, potrei incorrere nelle sanzioni previste dalla legge e la mia ammissione alla prova finale potrebbe essere negata.

Abstract

Questo elaborato esplora diverse tecniche per il rendering volumetrico, con un'attenzione particolare al metodo del Ray Marching. Il Ray Marching è una famiglia di metodi di rendering in cui i raggi di vista vengono tracciati iterativamente basandosi su una specifica metrica o struttura dati. Ogni raggio è tracciato per segmenti con l'obiettivo di individuare un punto di campionamento ottimale nella scena o all'interno del volume. Su tali punti viene successivamente valutato il modello di illuminazione e si calcolano il colore e l'opacità da assegnare a ciascun pixel. L'elaborato esamina un Ray Marcher già implementato e, dopo un'analisi approfondita, ne identifica punti di forza e debolezze. Viene proposta un'ottimizzazione del Ray Marcher e vengono riportati i risultati di un'analisi comparativa tra le due implementazioni. L'implementazione e l'analisi sono state svolte nel contesto della rappresentazione e visualizzazione di immagini mediche, in particolare CT scan.

Ringraziamenti

Grazie a

Marco Grangetto e Maurizio Lucenteforte per avermi dato l'opportunità di lavorare al progetto denominato "Visualizzazione immersiva e interattiva di volumi medici (CT scan) a fini diagnostici", base di partenza di questa tesi.

Marco Grangetto, Maurizio Lucenteforte, Davide Cavagnino e Agata Marta Soccini per essere stati i miei professori ed avermi fornito tutte le basi per sviluppare questa tesi.

Luna View per aver finanziato il progetto ed in particolare Davide Trombini per essere stato un punto di riferimento.

Marco (Cipo), che c'è sempre stato per me e mi ha sempre fornito una via di fuga dai libri quando era necessario.

Desidero esprimere la mia più profonda gratitudine ai miei genitori, che hanno sempre creduto in me, incoraggiandomi e sostenendomi nello studio, consentendomi di costruire un futuro su misura per me.

Ultimo, ma non meno importante, desidero rivolgere il mio più profondo ringraziamento a Giulia, la mia fidanzata, che mi ha supportato e sopportato durante tutti gli anni di Università. È solo grazie a lei se questa tesi di Laurea presenta pochi, se non addirittura nessun, errore grammaticale.

Sommario

Elenco delle Figure	vii
Elenco delle Equazioni	viii
Elenco delle Funzioni di Shading	ix
1 Introduzione	1
1.1 Definizione e Applicazione dei Voxel	2
1.2 La Scala Hounsfield	5
1.2.1 La Radiodensità	5
1.3 DICOM: Digital Imaging and Communications in Medicine	5
2 Direct Volume Rendering	7
2.1 Funzione di Trasferimento	8
2.2 Calcolo dell'Illuminazione	10
2.2.1 Illuminazione Locale	11
2.2.2 Illuminazione Globale	12
3 Ray Marching	17
3.1 Volumetric Ray Marching	19
3.1.1 Ottimizzazione del Volumetric Ray Marching	22
4 Progetto LunaVIEW	25
4.1 TBRaymarcherPlugin: Implementazione e Caso di Studio	25
4.1.1 Il Materiale: M_Raymarch	28
4.1.2 Gli Shader	30
5 TBRaymarcherPlugin: un Nuovo Modello d'Illuminazione	39
5.1 I Nuovi Shader	40
6 I due Modelli: Confronto Quantitativo	47
7 I due Modelli: Confronto Qualitativo	49
8 Conclusioni	53

Appendice A Volumetric Ray Tracing	55
A.1 Volumetric Path Tracing	59
Appendice B Texture-Based Volume Rendering	63
B.1 Variante con utilizzo di Texture 3D	65
Appendice C Splatting	67
Appendice D Shear-Warp Factorization	71
D.1 Ottimizzazione dello Shear-Warp Factorization	73
Appendice E Marching Cubes	75
E.1 Asymptotic Decider	79
Appendice F Lavori Correlati	81
F.1 GVDB	81
F.2 Cinematic Rendering	81
Bibliografia e Sitografia	83

Elenco delle figure

1	Griglia regolare	3
2	Funzione di trasferimento: look-up table	9
3	Illuminazione locale per rendering volumetrico	12
4	Sphere Tracing Ray Marching: esempio bidimensionale	18
5	Ray Marching: esempio terminazione di un'iterazione dell'algoritmo . .	19
6	Volumetric Ray Marching Plane Intersections	20
7	Compositing con over-operator	21
8	M_Raymarch: grafo del materiale	28
9	Confronto qualitativo: esempio 1	50
10	Confronto qualitativo: esempio 2	51
11	Confronto qualitativo: esempio 3	51
12	Point by Point Shading	56
13	Ray Tracing: albero di ricorsione	57
14	Path Tracing: condizione critica	60
15	Bidirectional Path Tracing	61
16	Slicing	63
17	Slicing perpendicolare alla direzione di vista	65
18	Splatting	69
19	Kernel gaussiano: impronta	70
20	Shear-Warp Factorization	71
21	Shear-Warp Factorization: proiezione prospettica	73
22	Shear-Warp Factorization: ottimizzazione con Run-Length Encoding .	74
23	Marching cubes: configurazioni	75
24	Marching cubes: esempio di denominazione dei vertici	76
25	Marching cubes: 256 possibili marcature dei vertici	76
26	Marching cubes33: configurazioni	78
27	Marching Cubes: caso ambiguo	79

Elenco delle Equazioni

1	Trasformazione del coefficiente di attenuazione lineare	5
2	Funzione di trasferimento	8
3	Illuminazione locale: Blinn-Phong	11
4	Illuminazione globale: James Kajiya	13
5	Illuminazione globale: approssimazione volumetrica	14
6	Legge di Beer-Lambert	14
7	Coefficiente di estinzione per rendering volumetrico	15
8	Legge di Beer-Lambert per volumi eterogenei	15
9	In-scattering	15
10	Volume Rendering Equation	16
11	Volume Rendering Equation per sistemi real-time	16
12	Over operator	19
13	Ricostruzione del volume attraverso convoluzione con kernel	68
14	Splatting: impronta	68
15	Splatting: compositing	68
16	Shear-Warp Factorization	71
17	Interpolazione bilineare per asymptotic decider	80

Elenco delle Funzioni di Shading

1	PerformRaymarchCubeSetup	30
2	RayAABBIntersection	32
3	PerformWindowedLitRaymarch	33
4	AccumulateWindowedRaymarchStep	36
5	NewPerformWindowedLitRaymarch	40
6	NewAccumulateWindowedRaymarchStep	43

1 Introduzione

Negli ultimi anni, lo sviluppo e la rapida diffusione dei visori per la realtà virtuale hanno introdotto nuove sfide e opportunità nel campo della visualizzazione immersiva, soprattutto nell'ambito medico. L'utilizzo sempre più frequente di sistemi di realtà aumentata e virtuale sta rivoluzionando la formazione e le pratiche mediche, come dimostrano iniziative come PrecisionOS [Pre], OssoVR [Oss], TA-VR Surgery [Sol] e INSIGHT HEART [RES], che si concentrano sulla simulazione, sulla rappresentazione realista dei dettagli anatomici e sulla formazione chirurgica attraverso scenari virtuali rappresentativi di situazioni reali.

Il settore sanitario è uno dei principali beneficiari dell'uso della realtà aumentata e virtuale. Queste tecnologie offrono agli operatori sanitari esperienze di formazione più realistiche, contribuendo a ridurre il tasso di insuccessi e fornendo soluzioni pratiche per facilitare operazioni che non possono essere apprese in modo meccanico. Secondo una ricerca di Goldman Sachs Global Investment [Res16], si prevede che il mercato della realtà aumentata in ambito sanitario raggiungerà un valore di 5,1 miliardi di dollari entro il 2025.

Il plugin open source per Unreal Engine, sviluppato da Tomas Bartipan e noto come TBRaymarcherPlugin [Bar], è stato scelto come punto di partenza per questa tesi. Questo plugin fornisce gli strumenti fondamentali per una visualizzazione immersiva di volumi medici direttamente all'interno dell'editor di gioco. Esso include un algoritmo di rendering basato su voxel, specificamente un Ray Marcher, e una struttura di asset per visualizzare le CT scan in formato Dicom.

L'obiettivo principale di questo elaborato è affrontare la necessità cruciale, nel settore medico, di una visualizzazione dettagliata e realistica dei volumi in tempo reale, ci si concentra quindi su due aspetti di ottimizzazione. Sebbene il modello di rendering fosse già ottimizzato per la visualizzazione interattiva in tempo reale, sono state esaminate e riportate, nell'Appendice, tutte le possibili alternative al Ray Marcher per il rendering di voxel. Infine, viene proposta un'ottimizzazione del modello ottico utilizzato dal Ray Marcher per valutare il colore e l'opacità da assegnare a ciascun pixel. Un'analisi comparativa, sia quantitativa che qualitativa tra i due modelli, è stata condotta e documentata.

1.1 Definizione e Applicazione dei Voxel

Il voxel è l'unità di base di un volume tridimensionale, così come il pixel è l'unità di base di un'immagine bidimensionale.

Per capire in dettaglio cos'è un voxel e come si è arrivati alla sua definizione bisogna comprendere come un volume viene rappresentato all'interno di un calcolatore. Un dato che rappresenta un volume è un insieme di campioni $(x, y, z, [p_1, p_2, \dots])$ che legano il vettore di proprietà $[p_1, p_2, \dots]$ ad una posizione (x, y, z) all'interno del volume. I campioni vengono selezionati accuratamente a determinati intervalli lungo ogni direzione del volume. L'insieme dei campioni può essere isotropo, ovvero, i campioni sono prelevati a intervalli uguali lungo ogni direzione del volume. D'altro canto, come più spesso accade, il volume può essere anisotropo, in questo caso gli intervalli possono essere diversi lungo ogni direzione.

Lo spazio tra ogni campione non è descritto in memoria, ma deve essere ricostruito al fine di visualizzare un volume. La ricostruzione avviene attraverso interpolazione e la soluzione più semplice e comune è la Nearest-Neighbor Interpolation, una funzione di interpolazione di ordine zero: il valore assegnato ad una data posizione è il valore del campione più vicino ad essa. Questo procedimento crea, per ogni campione, un volume cuboide centrato in esso dove l'insieme dei valori delle proprietà è costante. Il volume centrato in un campione è denominato voxel.

Un'alternativa per eseguire l'interpolazione è sfruttare soluzioni di ordine maggiore, come ad esempio: l'interpolazione lineare, quadratica o cubica.

Riassumendo, nell'ambito della computer grafica 3D, un voxel rappresenta un valore su una griglia regolare in uno spazio tridimensionale.

Una griglia regolare viene definita come

“ la tassellatura di uno spazio euclideo n-dimensionale mediante parallelotopi congruenti. ”

Graficamente, una griglia regolare, si presenta così:

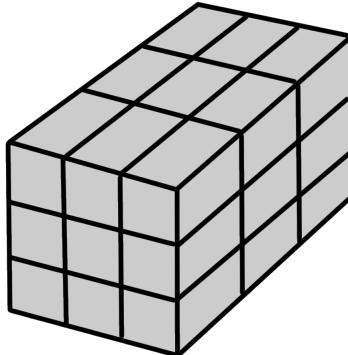


Figura 1: Griglia regolare. Fonte: immagine autoprodotta.

I voxel sono stati applicati per la prima volta nel campo della diagnostica medica, in particolare per la visualizzazione e l’analisi di immagini tridimensionali del corpo umano. Questa tecnologia ha permesso di ottenere una visione dettagliata dell’anatomia interna in modo più accurato rispetto alle immagini bidimensionali, contribuendo notevolmente alla valutazione di diverse condizioni mediche.

Successivamente, i voxel sono stati utilizzati anche nell’industria dei videogiochi, ma l’uso dei voxel in questo campo si diversifica dalla loro applicazione in medicina: nei videogiochi, i voxel sono stati spesso utilizzati per rappresentare terreni, oggetti e ambienti in modo più dettagliato rispetto agli approcci basati su poligoni. Purtroppo, l’uso dei voxel nei videogiochi ha subìto un certo declino a favore di tecniche più avanzate basate su poligoni, specialmente con l’avvento delle moderne schede grafiche che sono in grado di gestire scene complesse basate su modelli poligonali senza compromettere le prestazioni.

Nel campo della diagnostica medica, invece, la ricerca è ancora molto attiva e la tecnologia largamente utilizzata. Si preferisce utilizzare i voxel invece dei poligoni per le seguenti ragioni:

- Minore necessità di pre-elaborazione: i dati ottenuti da una tomografia computerizzata o una risonanza magnetica sono dati volumetrici. Il rendering basato su poligoni richiede di convertire i dati volumetrici in una forma poligonale, il che può comportare la perdita di informazioni o la comparso di artefatti. Il rendering di voxel evita questa fase di pre-elaborazione, preservando meglio l’integrità dei dati originali.

- Conservazione dei dettagli: poiché il voxel è la più piccola unità di dato in uno spazio tridimensionale, il rendering di voxel permette di catturare meglio i dettagli delle strutture anatomiche.

Per le motivazioni sopracitate, da qui in avanti, l'attenzione verrà posta sul rendering di voxel per volumi medici. E' bene precisarlo in quanto alcuni passi del processo di rendering variano in base alla sua finalità.

Si è detto che un voxel è descritto da una posizione e da un insieme di proprietà, ad esempio: opacità, colore, densità, materiale e velocità. Nel caso delle immagini mediche, in particolare delle CT scan (tomografia computerizzata), un voxel è unicamente rappresentato dall'opacità, che, nel contesto medico, si calcola in Hounsfield units (HU). Tra i valori descrittivi di un voxel non è inclusa la posizione globale (coordinate), è molto importante, invece, conoscere la posizione del voxel nella struttura dati che costituisce il volume. La posizione relativa del voxel rispetto ai suoi vicini è calcolata dal sistema di rendering: questa caratteristica rende i voxel perfetti per rappresentare volumi non omogenei.

Esistono due alternative per visualizzare un volume descritto attraverso i voxel, il rendering diretto del volume attraverso una delle seguenti tecniche:

- Ray Marching [Bru16]
- Volumetric Ray Tracing [AK95]
- Volumetric Path Tracing [LW96]
- Texture-Based Volume Rendering [IKLH04]
- Splatting [Wes91]

oppure, attraverso il rendering indiretto del volume. I metodi di questa famiglia si pongono come obiettivo l'estrazione di iso-superfici poligonali che seguono i contorni determinati da certi valori di soglia. La soluzione più comune è il Marching Cube [LC87], altre soluzioni sono:

- Marching Tetrahedra [Nie08]
- Dividing Cubes [BMN96].

1.2 La Scala Hounsfield

La scala Hounsfield è una scala quantitativa utilizzata per descrivere la radiodensità. La scala Hounsfield è una trasformazione lineare del coefficiente di attenuazione lineare dei materiali rispetto alla radiodensità dell'acqua distillata a pressione e temperatura standard. L'acqua distillata ha radiodensità pari a 0 *HU* e la radiodensità dell'aria è pari a -1000 *HU*. Si è deciso di utilizzare l'acqua come punto di riferimento perché costituisce circa il 60% della composizione corporea e perché la sua densità è nota. Da questa scelta deriva che materiali più densi dell'acqua hanno un valore maggiore di 0 *HU*. Materiali meno densi dell'acqua hanno un valore minore di 0 *HU*. Il coefficiente di attenuazione lineare è una misura della facilità con cui un materiale può essere attraversato da un fascio di luce ed è indicato con la lettera greca μ .

In un voxel con coefficiente di attenuazione lineare μ , il corrispondente valore *HU* è dato dalla formula:

$$HU = 1000 \cdot \frac{\mu - \mu_{water}}{\mu_{water} - \mu_{air}} \quad (1)$$

Nel contesto della tomografia computerizzata, la scala Hounsfield consente la visualizzazione e la quantificazione delle varie densità dei tessuti corporei.

1.2.1 La Radiodensità

La radiodensità descrive l'incapacità dei raggi X di attraversare un determinato materiale. Nei risultati delle scansioni CT o di altre tecniche di imaging, i materiali che ostacolano significativamente il passaggio dei raggi X sono noti come "radiopachi". Questi materiali appaiono bianchi nelle immagini e includono strutture come le ossa e i denti. D'altra parte, i materiali che consentono il passaggio dei raggi X senza ostacolarli significativamente sono denominati "radiotrasparenti". Questi materiali appaiono più scuri nelle immagini e la loro scurezza è proporzionale alla radiotrasparenza del materiale, infatti, l'aria risulta essere totalmente nera. Altre strutture radiotrasparenti sono i polmoni, i muscoli e i tessuti adiposi.

1.3 DICOM

Dicom è l'acronimo di Digital Imaging and Communications in Medicine ed è uno standard che definisce i criteri per il mantenimento di immagini mediche, ad esem-

pio le CT scan.

Lo standard Dicom non definisce nessun algoritmo di compressione, è semplicemente un modo per definire come i dati debbano essere codificati, archiviati e interpretati.

Un file Dicom è composto da circa 3000 tag, i più importanti sono:

- **{0x0008, 0x0018} Image UID:** identifica univocamente l'immagine.
- **{0x0008, 0x0060} Modality:** tipologia di apparecchiatura che ha acquisito i dati.
- **{0x0018, 0x0050} Slice thickness:** spessore in mm della fetta.
- **{0x0018, 0x0088} Spacing between slices:** distanza tra due fette in mm, misurata rispetto al centro delle due fette.
- **{0x0020, 0x0032} Patient position:** posizione spaziale dell'immagine rispetto al paziente.
- **{0x0020, 0x0037} Patient position cosines:** orientamento dell'immagine rispetto al paziente.
- **{0x0028, 0x0010} Num rows:** dimensione verticale dell'immagine in pixel.
- **{0x0028, 0x0011} Num cols:** dimensione orizzontale dell'immagine in pixel.
- **{0x0028, 0x0030} Pixel spacing:** distanza tra i centri di due pixel adiacenti, sia in direzione orizzontale che verticale.
- **{0x0028, 0x0100} Bits allocated:** numero di bit utilizzati per pixel. Il valore deve essere uguale per ogni fetta del campione.
- **{0x0028, 0x1052} Pixel offset:** o rescale intercept, ovvero un valore numerico che definisce la relazione tra valori memorizzati e unità di output (in genere HU).
- **{0x7FE0, 0x0010} Pixel data:** stream di pixel che compongono l'immagine.

2 Direct Volume Rendering

L'idea alla base degli algoritmi che fanno parte di questa famiglia è quella di ottenere una rappresentazione tridimensionale dei dati volumetrici direttamente, senza la necessità di estrarre esplicitamente superfici geometriche dai dati.

Un algoritmo di rendering è tipicamente formato da due parti:

- un modello ottico;
- una tecnica per valutare questo modello in determinati punti della scena.

Utilizzando le proprietà dei voxel, gli algoritmi di rendering sfruttano il modello ottico per derivare colore e opacità in un punto della scena. A seguire, durante il processo di rendering colore e opacità vengono accumulati lungo ogni raggio per formare l'immagine che sarà in seguito visualizzata dall'utente: questo processo viene chiamato "compositing" o "blending".

I parametri ottici sono specificati direttamente dai valori dei dati, o vengono calcolati applicando una o più funzioni di trasferimento ai dati. L'obiettivo della funzione di trasferimento nelle applicazioni di visualizzazione è quello di enfatizzare o classificare le caratteristiche di interesse.

Tutti gli algoritmi di questa famiglia utilizzano strutture dati di ausilio e in base all'ordine in cui le strutture dati vengono attraversate possiamo dividere gli algoritmi in due gruppi:

- Image-order: lanciano uno o più raggi per ogni pixel e processano i voxel lungo ognuno. Gli algoritmi che rientrano in questo gruppo sono: Ray Marching, Volumetric Ray Tracing e Volumetric Path Tracing. Lo svantaggio di questo ordine di elaborazione è che la struttura dati deve essere attraversata una volta per ogni raggio, con alta probabilità di calcoli ridondanti.
- Object-order: operano mediante la proiezione dei voxel sul piano immagine mentre scorrono attraverso i dati volumetrici salvati in memoria rispettando l'ordine di archiviazione. Gli algoritmi che rientrano in questo gruppo sono: Texture-Based Volume Rendering e Splatting. Su questo tipo di ordine di elaborazione non è facile implementare metodi di ottimizzazione.

Nei paragrafi seguenti si approfondiranno il concetto di funzione di trasferimento e le possibili scelte di implementazione per il modello ottico. Nei prossimi Capi-

toli e nell'Appendice verranno invece affrontate in dettaglio le singole tecniche di rendering che rientrano in questa famiglia.

2.1 Funzione di Trasferimento

Una funzione di trasferimento è una mappa che associa, nel caso dell'imaging medico, un valore di intensità a un colore C e a un grado di opacità α .

$$T : \mathbb{R} \rightarrow [C, \alpha] \quad (2)$$

Nella maggior parte delle implementazioni il colore C è una tripla (rosso, verde, blu).

L'uso del valore dei dati come unica misura per controllare l'assegnazione del colore e dell'opacità può limitare l'efficacia della classificazione delle caratteristiche del volume. Per questo motivo, è bene applicare la funzione di trasferimento in concomitanza ad una trasformazione dei dati. Quest'ultima avviene attraverso una particolare look-up table chiamata "window". Il processo attraverso il quale si definisce la finestra è chiamato "windowing", esso consente di selezionare un intervallo di intensità all'interno dell'immagine e mapparlo su un intervallo di visualizzazione più limitato. La finestra è definita in base a due parametri, il suo centro ("center") e la sua larghezza ("width"), entrambe le quantità sono in Hounsfield units. Il valore di "center" deve cadere all'interno dell'intervallo compreso tra il valore minimo e il valore massimo di intensità dei dati, mentre il valore di "width" deve essere compreso tra 0 e la somma del valore minimo e il valore massimo di intensità.

La trasformazione applicata ai dati è descritta in dettaglio dallo pseudo-codice e dall'immagine qui di seguito:

```
1 // Valori definiti dai dati.  
2 data_max_intensity_value (HU)  
3 data_min_intensity_value (HU)  
4 sample_intensity_value (HU)  
5  
6 // Valori definiti dall'utente.  
7 window_width (HU)  
8 window_center (HU)  
9
```

```

10 normalized_window_center =
11   (window_center - data_min_intensity_value) /
12   (data_max_intensity_value - data_min_intensity_value)
13
14 normalized_window_width = window_width /
15   (data_max_intensity_value - data_min_intensity_value)
16
17 normalized_sample_intensity_value =
18   (sample_intensity_value - data_min_intensity_value) /
19   (data_max_intensity_value - data_min_intensity_value)
20
21 TF_position = (normalized_sample_intensity_value -
22   normalized_window_center + (normalized_window_width / 2)) /
23   normalized_window_width
24
25 rgba_values = submit_TF_position_to_TF_LUT(TF_position)

```

Dove la funzione `submit_TF_position_to_TF_LUT` rappresenta l'operazione di consultazione di una serie di valori da una look-up table:

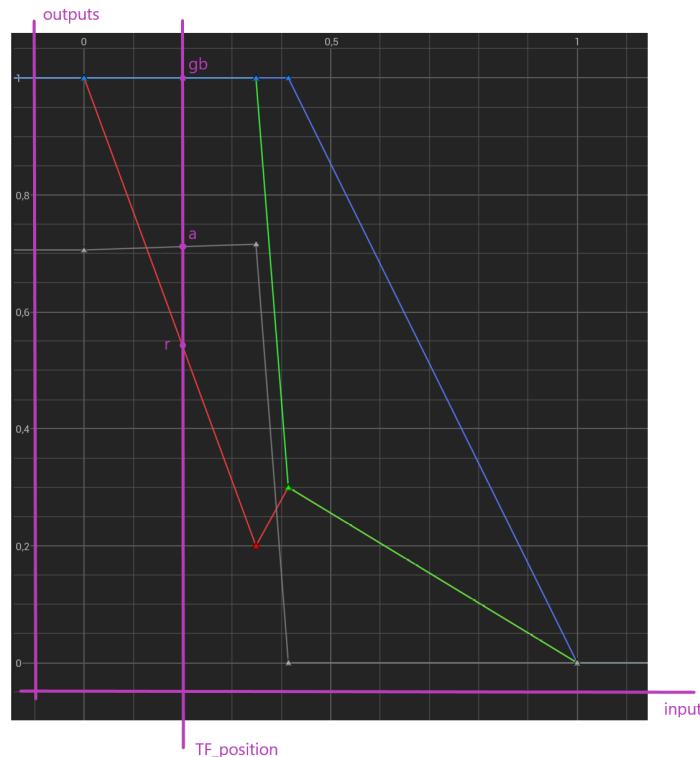


Figura 2: Funzione di trasferimento: look-up table.

Fonte: TBRaymarcherPlugin.

I valori di default che garantiscono un'efficace classificazione delle caratteristiche del volume sono:

Classificazione	Center (HU)	Width (HU)
Ossa	500	2000
Polmoni	-600	1600
Tessuti molli	50	350
Fegato	60	160
Aria	-1000	100

2.2 Calcolo dell'Illuminazione

Il calcolo dell'illuminazione, o semplicemente "lighting" in inglese, è il processo attraverso il quale si calcola l'interazione tra la luce e la superficie di un oggetto di scena: questo è il compito del modello ottico.

Le sorgenti di luce sono essenziali per il calcolo dell'illuminazione e possono essere suddivise come segue:

- Luce ambientale: luce che pervade la scena in modo uniforme, contribuendo a illuminare gli oggetti senza creare ombre nette.
- Luce direzionale: utilizzata per simulare la luce solare. Questa sorgente emette raggi luminosi in una direzione specifica e la sua posizione è considerata virtualmente infinita rispetto alla scena.
- Luce puntiforme: irradia la luce in tutte le direzioni da un punto specifico nello spazio.
- Spotlight (luce a faretto): simile a una luce puntiforme, ma con una direzione specifica. La luce si irradia in un cono, consentendo di concentrarla in una zona specifica.
- Luce ad area: questa sorgente di luce ha un'estensione fisica ed emette la luce in tutte le direzioni dalla sua superficie.

Le proprietà che definiscono come la superficie interagisce con la luce sono totalmente descritte dal materiale che la compone. L'unica decisione da prendere circa il calcolo dell'illuminazione è quale/i modello/i implementare. Si devono

distinguere quindi i modelli per illuminazione locale e modelli per illuminazione globale.

2.2.1 Illuminazione Locale

Quando si parla di illuminazione locale ci si riferisce a metodi che modellano solo la luce diretta proveniente dalle fonti di luce principali nella scena. Questo tipo di illuminazione ignora gli effetti di luce indiretta.

- La scelta più comune è il modello di Blinn-Phong [Bli77], che calcola l'intensità della luce riflessa in funzione:
 - della normale alla superficie \vec{n} ,
 - della direzione \vec{l} della luce incidente,
 - della direzione \vec{v} della vista,
 - dell'intensità I_L della luce incidente,
 - dei coefficienti ambientale (k_a), diffusivo (k_d) e speculare (k_s) del materiale

$$I = k_a I_a + k_d I_L (\vec{n} \cdot \vec{l}) + k_s I_L (\vec{h} \cdot \vec{n})^n \quad (3)$$

dove \vec{h} è l'halfway vector, calcolato come la media normalizzata fra direzione di vista e direzione della luce.

- Un'alternativa alla scelta appena descritta è il suo antenato: il modello di Phong [Pho75]. Esso utilizza il vettore di riflessione anziché l'halfway vector.
- Altre alternative sono specifiche per il tipo di materiale o superficie che si vuole renderizzare, ad esempio: per superfici opache e non riflettenti può essere efficacemente utilizzato il modello di Lambert [ON94], mentre per superfici ruvide e opache il modello di Oren-Nayar [ON94].

I modelli sopracitati sono irrealistici per il rendering di voxel in quanto sono tutti modelli basati sulla superficie. Per poterli applicare ai voxel si deve utilizzare un buffer di supporto per accumulare la quantità di luce attenuata dal punto di vista della sorgente luminosa.

Il primo passo consiste nel suddividere il volume in fette. Il taglio deve essere eseguito perpendicolarmente rispetto al vettore medio tra la direzione di vista e la direzione della luce, ciò consente di renderizzare una fetta sia dal punto di vista della luce che da quello dell'osservatore.

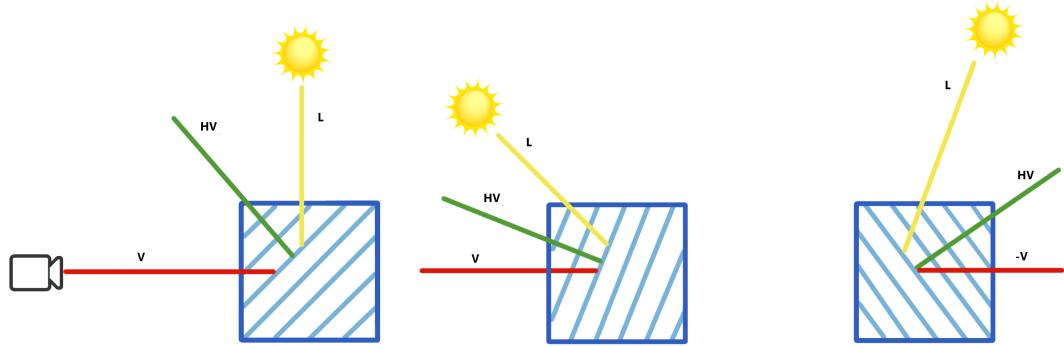


Figura 3: taglio perpendicolare all’halfway vector.

Fonte: immagine autoprodotta.

La quantità di luce che arriva ad una fetta è pari a 1 a cui viene sottratta l’opacità accumulata dalle fette precedentemente renderizzate. Ogni fetta del volume viene prima renderizzata dal punto di vista dell’osservatore. Per farlo, vengono utilizzati i risultati ottenuti dal rendering dal punto di vista della luce delle fette precedenti. La stessa fetta viene poi renderizzata dal punto di vista della luce ed i risultati verranno utilizzati per il rendering delle fette successive.

2.2.2 Illuminazione Globale

A differenza dell’illuminazione locale, che si concentra esclusivamente sulla modellazione della luce proveniente da sorgenti dirette, l’illuminazione globale si distingue per la sua capacità di plasmare sia la luce proveniente da fonti dirette sia quella che giunge in modo indiretto sulla superficie.

Storicamente, sono state proposte diverse formulazioni dell’equazione di rendering che modella l’illuminazione globale in scene 3D. Attualmente la formulazione più utilizzata è quella di David Immel et al. e James Kajiya, del 1986 [Kaj86] [ICG86]:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + L_r(\mathbf{x}, \omega_o, \lambda, t) = \\ L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i \quad (4)$$

dove:

- $L_o(\mathbf{x}, \omega_o, \lambda, t)$ è la radianza totale uscente dal punto \mathbf{x} , lungo la direzione ω_o , di lunghezza d'onda pari a λ , al tempo t .
- $L_e(\mathbf{x}, \omega_o, \lambda, t)$ è la radianza emessa.
- $L_r(\mathbf{x}, \omega_o, \lambda, t)$ è la radianza riflessa.
- Ω è l'emisfero unitario centrato in \mathbf{x} e contenente tutti i possibili valori per $\omega_i : \omega_i \cdot \mathbf{n} > 0$.
- $f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$ è la BDRF (bidirectional reflectance distribution function), una funzione che restituisce un valore che rappresenta la quantità di luce riflessa in uscita nella direzione ω_o rispetto alla quantità di luce incidente ω_i nel punto \mathbf{x} . Gli ultimi due parametri esprimono la lunghezza d'onda e il tempo.
- $L_i(\mathbf{x}, \omega_i, \lambda, t)$ è la radianza incidente.
- \mathbf{n} è la normale alla superficie nel punto \mathbf{x} .

Valutare la radianza riflessa attraverso l'integrale per ogni possibile direzione dell'emisfero è un compito molto costoso. Eric Lafourche, nel 1995, introduce un'approssimazione dell'equazione attraverso il metodo Monte Carlo [Laf95] in cui:

1. Vengono generati campioni casuali per le direzioni di luce ω_i nell'intervallo definito da Ω . I campioni devono essere generati in modo da seguire la distribuzione di probabilità della BRDF.
2. Per ogni valore generato viene valutata la BRDF: $f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t)$ e la radianza indicente: $L_i(\mathbf{x}, \omega_i, \lambda, t)$.
3. I contributi di tutti i campioni vengono accumulati:

$$\sum_{\omega_i} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}).$$
4. I contributi accumulati vengono mediati rispetto il numero di campioni. Più campioni si sono generati più il risultato approssimato si avvicina al valore reale.

L'equazione, come è stata presentata, copre la riflessione e l'emissione di luce, ma non include esplicitamente la rifrazione. Per includere la rifrazione nell'equazione di rendering bisogna considerare il trasferimento di luce attraverso la superficie: ciò può essere fatto a patto che la superficie sia trasparente o semi-trasparente. Per gli scopi di questa tesi, l'analisi della rifrazione non è essenziale, perciò, non verrà approfondita ulteriormente.

Anche in questo caso, come per l'illuminazione locale, la formulazione proposta da Kajiya non è adeguata per il rendering volumetrico perché presuppone che la luce interagisca con una superficie: riflessione, rifrazione ed emissione sono fenomeni caratteristici di questa interazione. Invece, un volume può emettere, assorbire e disperdere la luce al suo interno e/o esterno. Tutto questo deve essere considerato nell'equazione di rendering, che diventa:

$$L_o(\dots) = \text{emission} + \text{in-scattering} - \text{out-scattering} - \text{absorption} \quad (5)$$

dove:

- L'in-scattering si verifica quando la luce attraversa un volume e, anziché proseguire senza raggiungere l'occhio, viene deviata verso di esso a causa di un evento di dispersione.
- L'out-scattering si verifica quando un raggio luminoso, che tipicamente si dirigerebbe verso l'occhio, viene invece disperso in un'altra direzione.
- L'assorbimento si verifica quando una frazione dell'energia è assorbita dalle particelle del volume.

I termini di assorbimento e di out-scattering (ovvero quelli responsabili dell'attenuazione della luce che arriva all'occhio), sono modellati attraverso la legge di Beer-Lambert, la quale definisce l'esistenza di una dipendenza esponenziale tra la quantità di luce che attraversa il volume e il prodotto tra la distanza che la luce percorre all'interno del volume e la somma dei coefficienti di assorbimento e di scattering:

$$T = e^{-\text{thickness} * (\sigma_a + \sigma_s)} \quad (6)$$

Più elevati sono i coefficienti o la distanza, minore sarà il valore di T . L'equazione restituisce un valore compreso nell'intervallo $[0, 1]$.

Questa formulazione si adatta a volumi omogenei, ma non a volumi eterogenei, ovvero volumi medici. Si ricava la formulazione per i volumi eterogenei introducendo i seguenti termini:

- $\sigma_t = \sigma_a + \sigma_s$ è il coefficiente di estinzione.
- $\tau = \text{thickness} * \sigma_t$ è la profondità ottica per volumi omogenei.

Per volumi eterogenei il coefficiente di estinzione varia di voxel in voxel. Per calcolare la profondità ottica si deve integrare il coefficiente di estinzione lungo il raggio:

$$\tau = \int_{s=0}^d \sigma_t(v_s) ds \quad (7)$$

dove d è la distanza che il raggio percorre all'interno del volume e $\sigma_t(v_s)$ è il coefficiente di estinzione del voxel che si trova a distanza s dal punto di ingresso del raggio. A questo punto si può definire la legge di Beer-Lambert per volumi eterogenei:

$$T(d) = e^{-\int_{s=0}^d \sigma_t(v_s) ds} \quad (8)$$

Il fenomeno dell'in-scattering, responsabile dell'amplificazione luminosa, può essere modellato analizzando, lungo il raggio della camera, la quantità di luce deviata verso l'osservatore dalle particelle del volume.

La luce è composta da molteplici raggi e per calcolare la luce totale dispersa lungo un raggio di vista si deve tenere conto dell'effetto lungo tutto il raggio che, partendo dalla fotocamera, arriva al volume e lo attraversa. Si utilizza quindi un integrale:

$$L_{is}(\mathbf{x}, \omega_o, \lambda, t) = \int_{\Omega_{4\phi}} f_p(\mathbf{x}, \omega'_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega'_i, \lambda, t) d\omega'_i \quad (9)$$

dove:

- $\Omega_{4\phi}$ è la sfera unitaria centrata in \mathbf{x} e contenente tutti i possibili valori per ω'_i .

- $f_p(\mathbf{x}, \omega'_i, \omega_o, \lambda, t)$ è la funzione angolare di scattering e descrive la quantità di luce dispersa in una specifica direzione ω'_i . La funzione restituisce un valore nell'intervallo $[0, 1]$ ed è fondamentale tenerla in considerazione in quanto la dispersione, in realtà, non è mai isotropa.

Unendo i termini precedentemente descritti si ottiene l'equazione denominata "Volume Rendering Equation":

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = \int_{q=0}^d T(q) [\sigma_a(v_q)L_e(\mathbf{x}_q, \omega_o, \lambda, t) + \sigma_s(v_q)L_{is}(\mathbf{x}_q, \omega_o, \lambda, t)] dq + T(d)L_d(\mathbf{x}_d, \omega_o, \lambda, t) \quad (10)$$

dove:

- $L_d(\mathbf{x}_d, \omega_o, \lambda, t)$ è la radianza alla fine del raggio di vista. Può essere definita dallo sfondo o da un altro oggetto intersecato.

Tenere in considerazione il fenomeno dell'in-scattering durante il processo di rendering conferisce al volume un aspetto notevolmente più realistico. Tuttavia, risolvere l'integrale per ogni camera ray richiede la disponibilità di elevate risorse di calcolo. Nel contesto dei sistemi di rendering real-time l'obiettivo primario è mantenere l'interazione con la scena il più fluida possibile. Di conseguenza, questo fenomeno viene spesso ignorato. Il risultato che ne deriva è il seguente:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = \int_{q=0}^d T(q) [\sigma_a(v_q)L_e(\mathbf{x}_q, \omega_o, \lambda, t)] dq + T(d)L_d(\mathbf{x}_d, \omega_o, \lambda, t) \quad (11)$$

3 Ray Marching

Il Ray Marching è una famiglia di metodi di rendering in cui i raggi di vista vengono tracciati iterativamente. Ogni raggio è tracciato per segmenti e per ognuno di essi viene valutata una certa funzione.

L'algoritmo generale che descrive il Ray Marching è il seguente.

1. Bisogna scegliere preliminarmente:
 - un punto nello spazio da cui far partire i raggi, viene chiamato Ray Origin (*RO*);
 - un certo valore di accuratezza (ϵ);
 - una metrica di distanza D che sarà usata per calcolare la distanza tra l'origine del segmento e il punto di intersezione tra una figura costruita sul segmento e l'oggetto di scena. Le scelte più comuni sono:
 - Signed distance function (SDF): è la distanza ortogonale tra un punto e un confine. Nel caso si scegliesse questa metrica, la figura costruita sul segmento sarebbe una sfera e l'implementazione prenderebbe il nome di Sphere Tracing Ray Marching.
 - Manhattan distance: è la distanza tra due punti misurata come la somma del valore assoluto delle differenze delle loro coordinate. Nel caso si scegliesse questa metrica, la figura costruita sul segmento sarebbe un cubo e l'implementazione prenderebbe il nome di Cube Assisted Ray Marching.
 - un valore *MaxIteration* oltre il quale non prolungare un raggio. Questo valore è necessario per la convergenza dell'algoritmo.
2. Si traccia un segmento che congiunge *RO* al centro di un pixel. La punta del segmento appena tracciato diventa la coda di un secondo.
3. La punta del nuovo segmento viene fatta avanzare finché la figura costruita sul segmento non interseca un oggetto di scena.

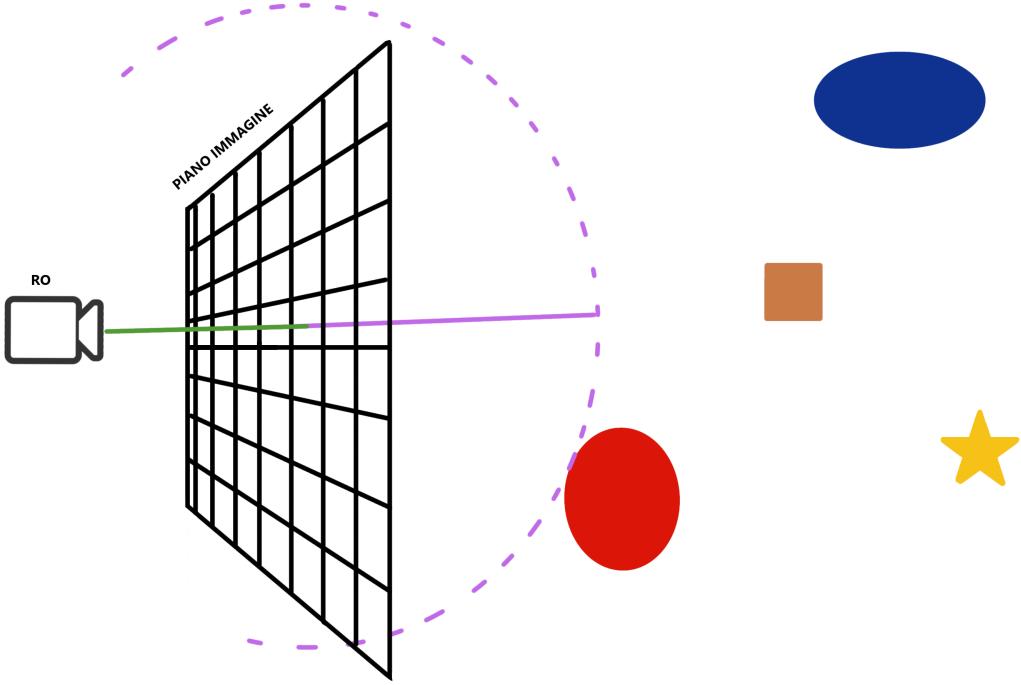


Figura 4: Sphere Tracing Ray Marching.

Step 2 in verde.

Step 3 in viola.

Fonte: immagine autoprodotta.

4. Se la distanza tra l'origine del segmento e il punto di intersezione è minore di ϵ ci si ferma. Il modello ottico verrà valutato nel punto in cui la figura costruita sul segmento interseca l'oggetto di scena. Da questa valutazione deriva il colore del pixel.
5. Se la distanza tra l'origine del segmento e il punto di intersezione è maggiore o uguale di ϵ , la punta del segmento diventa la coda di un nuovo segmento e si ripete il procedimento dal punto 3.
6. Se il numero di iterazioni eseguite è maggiore di $MaxIteration$, al pixel viene dato il colore dello sfondo.
7. Ripetere il procedimento dal punto 2 per ogni pixel. La prima iterazione terminerà in questa situazione:

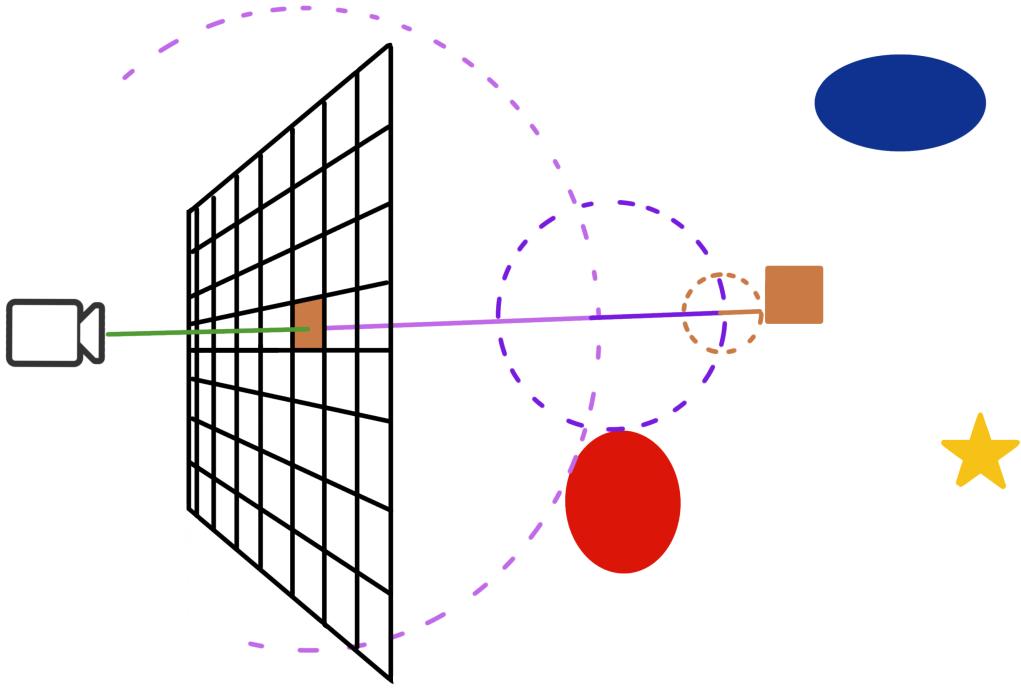


Figura 5: terminazione dell'algoritmo. Fonte: immagine autoprodotta.

Quanto descritto è il Ray Marching per rendering di superfici, tuttavia, l'obiettivo di questa tesi è presentare le soluzioni per il rendering di voxel. L'algoritmo deve essere esteso e verrà di conseguenza approfondito il Volumetric Ray Marching.

3.1 Volumetric Ray Marching

Il Volumetric Ray Marching calcola immagini bidimensionali partendo da set di dati volumetrici tridimensionali.

Nella variante volumetrica il calcolo non si ferma alla superficie ma continua all'interno del volume, campionandolo a determinati intervalli. Ad ogni intervallo il modello ottico viene valutato e successivamente i valori di opacità e colore vengono accumulati secondo l'over-operator:

$$\begin{aligned} C^{out} &= C^{in} + (1 - \alpha^{in})\alpha^{voxel}C^{voxel} \\ \alpha^{out} &= \alpha^{in} + (1 - \alpha^{in})\alpha^{voxel} \end{aligned} \tag{12}$$

Il processo può essere descritto come segue:

1. I raggi devono essere prolungati all'interno del volume in modo da trovare i punti di campionamento. Esistono tre alternative.

- Volumetric Ray Marching Plane Intersections: un volume composto da $N \times N \times N$ voxel può essere diviso utilizzando $N + 1$ piani rispetto ad ogni asse. Ogni punto di intersezione raggio-piano sarà considerato un punto di campionamento.

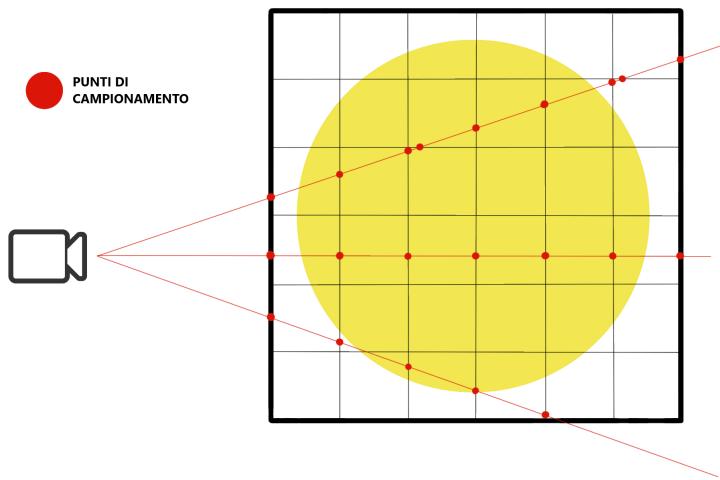


Figura 6: Volumetric Ray Marching Plane Intersections.

Fonte: immagine autoprodotta.

In questo modo il raggio marcia, all'interno del volume, di una quantità pari alla dimensione di un voxel. Esiste un caso degenere: un volume in cui tutti i voxel sono vuoti. Quest'ultimo scenario ha stimolato lo sviluppo di metodi di ottimizzazione noti come Space Leaping, che saranno affrontati nel paragrafo successivo.

- Sphere Assisted Volumetric Ray Marching: in ogni voxel vuoto viene salvato il raggio che rappresenta la sfera più grande in grado di riempire lo spazio vuoto intorno ad esso. Un raggio che interseca un voxel vuoto viene fatto marciare di una quantità pari all'informazione salvata nel voxel.

- Cube Assisted Volumetric Ray Marching: implementazione simile al metodo precedente ma che utilizza un cubo (anziché una sfera), il valore salvato nei voxel è pari alla metà del lato del cubo stesso.
2. Ogni punto di campionamento viene passato allo shader il quale valuterà il modello ottico, la funzione di trasferimento e deriverà colore e opacità. Successivamente, si campionerà back to front il volume, accumulando colore e opacità.

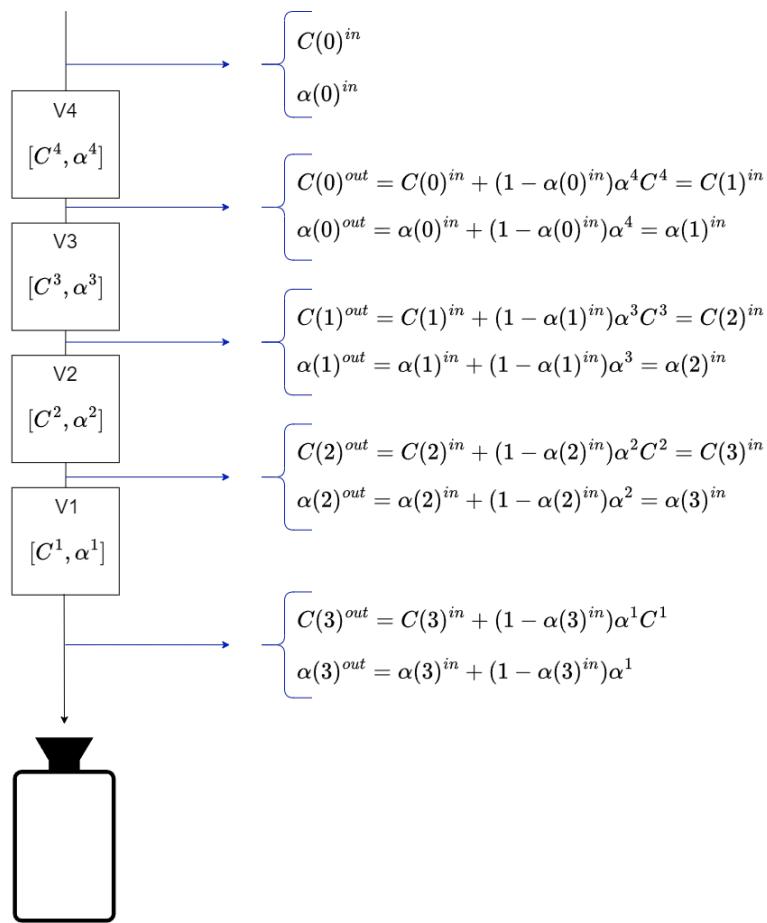


Figura 7: compositing con over-operator. Fonte: immagine autoprodotta.

Niente impedisce di utilizzare una combinazione di algoritmi per realizzare il rendering. Ad esempio, è possibile usare lo Sphere Tracing Ray Marching per cercare l’intersezione raggio-volume e poi il Volumetric Ray Marching Plane Intersections per campionare il suo interno.

3.1.1 Ottimizzazione del Volumetric Ray Marching

Il Volumetric Ray Marching soffre di un problema: è computazionalmente molto costoso. Per accelerare il processo di rendering bisogna trascurare alcune informazioni irrilevanti. I possibili miglioramenti che si possono attuare sono:

1. **Early Ray Termination**: raggiunto un certo valore di opacità diventa inutile continuare ad accumulare il colore. Le idee alla base dell'ottimizzazione sono terminare di attraversare il volume e accumulare informazione quando il contributo di un campione diventa irrilevante. Se si utilizza questo approccio bisogna campionare front-to-back.
2. **Space-Leaping**: risulta necessario utilizzare questa famiglia di miglioramenti solo se si è deciso di utilizzare il Plane Intersections per trovare i punti di campionamento. I voxel che non contengono porzioni di volume dovrebbero essere processati velocemente oppure del tutto ignorati nel processo di campionamento. Esistono diversi approcci per implementare questa ottimizzazione:
 - **Hierarchical spatial data structure**: i voxel che compongono il volume vengono divisi in regioni uniformi, ogni regione viene rappresentata da un nodo all'interno di una struttura gerarchica. Le informazioni della struttura vengono incluse all'interno dei voxel. Così, quando un raggio colpisce un voxel che appartiene ad una regione trasparente, può saltare direttamente al primo voxel al di fuori della regione sfruttando le informazioni della struttura di supporto.
 - **Homogeneity-acceleration**: i voxel che compongono il volume vengono divisi in regioni uniformi e vengono campionati solo i "voxel medi" di ogni regione.
 - **Proximity clouds**: attraverso uno step di pre-processing vengono individuati tutti i voxel trasparenti e per ognuno di essi viene calcolata la distanza rispetto al più vicino voxel pieno.
Quando un raggio incontra un voxel trasparente che dista n dal voxel pieno più vicino, il raggio può ignorare il voxel appena incontrato ed i successivi $n - 1$, in quanto si è sicuri che siano tutti trasparenti.

3. **Sfruttare la coerenza temporale nelle animazioni:** risulta necessario utilizzare questo approccio solo se si devono renderizzare una serie di immagini. Per ogni pixel, le coordinate del primo voxel non trasparente colpito dal raggio vengono memorizzate nel cosiddetto C-buffer (Coordinates-buffer). Le coordinate vengono calcolate dopo il rendering del primo fotogramma e poi aggiornate dopo ogni nuovo fotogramma renderizzato. Questi dati possono essere utilizzati per determinare da dove iniziare a campionare il volume per il rendering di fotogrammi successivi al primo. Aggiornare il C-buffer in maniera errata porta a risultati inaspettati.
4. **Campionamento dello schermo adattivo:** non tutti i pixel verranno campionati. Si decide se campionare un pixel in base al cambiamento di colore in un sottoinsieme di suoi vicini. Nelle aree con gradiente alto, si campioneranno tutti i pixel. In aree dove il gradiente è basso si useranno meno raggi. Per tutti i pixel non campionati, il colore verrà calcolato attraverso interpolazione. Questa ottimizzazione può essere utilizzata solo se si stanno renderizzando più fotogrammi in quanto è necessario avere almeno un'immagine di riferimento per calcolare i gradienti.

4 Progetto LunaVIEW

Negli ultimi anni, la visualizzazione immersiva e interattiva di dati medici ha acquisito una crescente importanza nel campo della diagnostica. Questa tecnologia non solo offre una migliore comprensione visiva dei volumi medici, ma permette anche ai professionisti della salute di interagire direttamente con i dati, migliorando l'accuracy e l'efficacia delle diagnosi. Il progetto LunaVIEW, sviluppato nel contesto della Borsa di Ricerca n. 27/2023 intitolata "Visualizzazione immersiva e interattiva di volumi medici (CT scan) a fini diagnostici", si colloca all'interno di questa cornice. Il progetto ha coinvolto l'adattamento e l'ottimizzazione del plugin per Unreal Engine, noto come TBRaymarcherPlugin [Bar], per soddisfare specifiche esigenze di visualizzazione. Un confronto iniziale avuto con il Dottor Giovanni Musella, esperto radiologo della Casa di Cura San Camillo di Cremona, ha guidato le iterazioni che hanno portato allo sviluppo del progetto. L'integrazione delle sue osservazioni, in aggiunta alla risoluzione di diverse problematiche legate alla logica dell'applicazione e all'uso di librerie deprecate da parte del plugin, hanno contribuito alla creazione di un'applicazione su misura per la visualizzazione immersiva e interattiva di volumi medici.

4.1 TBRaymarcherPlugin: Implementazione e Caso di Studio

Il plugin nasce nel 2018 come progetto di tesi magistrale di Tomas Bartipan [Bar]. Questo plugin offre diverse funzionalità per una visualizzazione realistica e interattiva di volumi medici.

- È stato sviluppato un Ray Marcher per il rendering di texture volumetriche ispirato alla guida e al codice di Ryan Brucks [Bru16]. Il modello realizzato è molto semplice e non tiene in considerazione l'assorbimento e la dispersione.
- È stata integrata l'illuminazione volumetrica utilizzando concetti tratti dall'articolo di Erik Sundén e Timo Ropinski dal titolo "Efficient volume illumination with multiple light sources through selective light updates" [SR15].
- È stato sviluppato un parser per ognuna delle seguenti tipologie di file: .dcm, .mhd, .raw e .png. In questo modo, è possibile convertire i volumi ricevuti in

output da qualsiasi tipologia di macchina in un oggetto della classe UVolumeTexture. Quest'ultima è la classe che Unreal Engine mette a disposizione per memorizzare informazioni volumetriche.

- È stato incluso il supporto per il windowing.
- Le funzioni di trasferimento sono state rappresentate attraverso curve di colore e sono stati forniti numerosi esempi predefiniti.
- È stato incluso il supporto ad un menu per modificare i parametri della finestra e la funzione di trasferimento da applicare al volume.

In questa tesi, ci si concentrerà esclusivamente sulla prima funzionalità. Verrà analizzato nel dettaglio il Ray Marcher e il modello ottico utilizzato per valutare la radianza nei punti di campionamento selezionati all'interno del volume. L'obiettivo è comprendere i punti deboli del modello e migliorarlo per ottenere una resa grafica superiore. Prima di tutto, è fondamentale analizzare i processi attraverso cui Unreal Engine esegue il rendering e le componenti coinvolte.

Nel processo di rendering sono coinvolti numerosi thread, quattro sono cruciali e degni di essere approfonditi.

- Il Game Thread, eseguito sulla CPU, gestisce tutte le trasformazioni degli oggetti nella scena, comprese animazioni, fisica, intelligenza artificiale e tutta la logica dell'applicazione, implementata sia tramite Blueprint che tramite codice *C++*.
- Il Draw Thread calcola quali elementi includere nel rendering. Quest'ultimo viene eseguito principalmente sulla CPU, ma alcune parti vengono eseguite anche sulla GPU. In questo thread avviene il processo di occultamento (occlusion), che attraverso quattro passaggi costruisce una lista di tutti gli oggetti visibili.
 1. Distance Culling: elimina dal processo di rendering tutti gli oggetti che si trovano oltre una determinata distanza dalla telecamera.
 2. View Frustum Culling: elimina dal processo di rendering tutti gli oggetti che si trovano al di fuori del view frustum della telecamera.

3. Precomputed Visibility: la scena viene suddivisa in celle, ciascuna delle quali memorizza i riferimenti agli oggetti potenzialmente visibili al suo interno.
 4. Occlusion Culling: controlla lo stato di visibilità di ogni oggetto nelle celle precedentemente calcolate. Gli oggetti non visibili vengono esclusi dal processo di rendering.
- Il Rendering Thread, a questo punto, sa quali oggetti devono essere rende- rizzati e dove si trovano, rimane da calcolare l'ordine in cui devono essere processati. Successivamente, il Rendering thread, si occupa del rendering vero e proprio. L'intero lavoro viene eseguito sfruttando la GPU.
 - L'RHI Thread (Render Hardware Interface) funge da interfaccia per la comu- nicazione tra il Rendering Thread e la GPU. In un ambiente come Windows, l'RHI Thread implementa OpenGL o Direct3D, a seconda della configura- zione e delle librerie utilizzate.

Il Draw Thread elabora gli output generati dal Game Thread, mentre il Rende- ring Thread elabora gli output del Draw Thread. Questo consente loro di operare in parallelo, ma comporta anche che il Draw Thread lavori su un fotogramma precedente rispetto al Game Thread, mentre il Rendering Thread lavora su un fotogramma precedente rispetto al Draw Thread. Questa configurazione può pre- sentare uno svantaggio: se uno dei thread è lento, tutti subiscono rallentamenti poiché operano simultaneamente. Il principale elemento su cui si focalizza il ren- derizzatore è il materiale che costituisce il volume. Quest'ultimo contiene tutte le informazioni necessarie al processo di rendering per derivare il colore e l'opacità da assegnare al pixel. Tomas Bartipan ha deciso di creare un materiale concepi- to come un semplice contenitore vuoto, la cui funzione principale è trasmettere informazioni ai metodi di shading definiti utilizzando il linguaggio HLSL. Questi metodi campionano ripetutamente una texture volumetrica lungo un raggio, tra- sformano i valori ottenuti attraverso una funzione di trasferimento, considerano l'illuminazione e infine accumulano campioni in un unico colore da assegnare al pixel. Il Rendering Thread richiama questi metodi durante la fase di pixel shading, che si occupa della resa del materiale per ciascun pixel del piano immagine.

Si esamineranno quindi più approfonditamente le implementazioni del materiale e degli shader.

4.1.1 Il Materiale: M_Raymarch

In Unreal Engine, un materiale è un elemento che si applica a una mesh per gestire l'aspetto visivo di un'oggetto all'interno della scena. Un materiale è espresso attraverso un grafo, che è una rappresentazione visuale delle istruzioni da eseguire per determinare l'aspetto finale della mesh a cui è assegnato. Come menzionato in precedenza, il materiale disponibile all'interno del plugin è un involucro vuoto che si interfaccia ai metodi definiti a livello più basso.

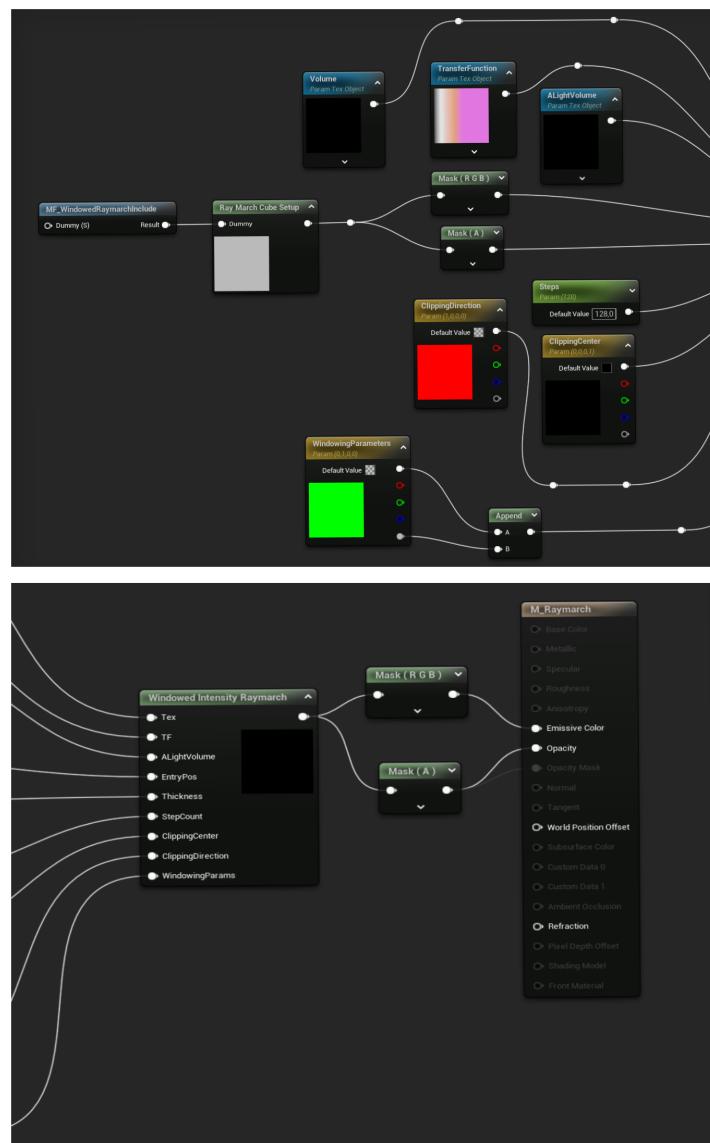


Figura 8: M_Raymarch: grafo del materiale.

Fonte: TBRaymarcherPlugin.

Il compito dei nodi è il seguente:

- Il nodo MF_WindowedRaymarchInclude importa tutte le funzioni di Ray Marching.
- Il nodo Ray March Cube Setup invoca la funzione di libreria PerformRaymarchCubeSetup, che calcola, per ciascun pixel, le coordinate del punto di ingresso del raggio nel volume e lo spessore del volume che il raggio deve attraversare. Si nota che il nodo accetta in input un cosiddetto "input dummy", un parametro non utilizzato all'interno della funzione ma necessario per definire il flusso di esecuzione. Diversamente dal flusso di esecuzione tradizionale dei Blueprint, questo grafo non consente la creazione di flussi contenenti funzioni senza input. Il vero input di questa funzione sono i MaterialParameters, calcolati dal pixel shader durante la sua inizializzazione e messi a disposizione ad ogni funzione che compone il flusso. L'output del nodo viene successivamente processato come un valore (R, G, B, A). Sebbene la sintassi utilizzata possa risultare fuorviante, questo metodo separa un array di quattro elementi in due array: il primo rappresenta le coordinate del punto di ingresso, mentre il secondo rappresenta lo spessore, ovvero una distanza.
- Il nodo Windowed Intensity Raymarch invoca la funzione di libreria PerformWindowedLitRaymarch, la quale prende in input:
 - la texture 3D che rappresenta il volume;
 - la texture 2D che rappresenta la funzione di trasferimento;
 - la texture 3D che rappresenta l'illuminazione all'interno della scena;
 - le coordinate del punto di ingresso del raggio nel volume;
 - lo spessore del volume;
 - il numero di passi di Ray Marching da eseguire;
 - la posizione del piano di clipping e l'orientamento della regione sezionata;
 - i parametri della finestra.

L'output di questo nodo sono due valori: un'opacità e un colore.

4.1.2 Gli Shader

In questo paragrafo si approfondiscono le funzioni di shading che i nodi del grafo del materiale invocano.

PerformRaymarchCubeSetup

```
1 // Tutti i valori restituiti sono codificati in coordinate UVW.
2 float4 PerformRaymarchCubeSetup
3     (FMaterialPixelParameters MaterialParameters)
4     // Parametri messi a disposizione dal pixel shader.
5 {
6     // Calcola la profondita' della scena (W, non Z).
7     // GetScreenPosition: il metodo restituisce la posizione
8     // post-proiezione dello schermo.
9     // ScreenAlignedPosition: il metodo calcola le coordinate UV
10    // corrispondenti alle coordinate dello schermo.
11    // CalcSceneDepth: il metodo restituisce la distanza della
12    // vista lungo l'asse W del view frustum.
13    float LocalSceneDepth = CalcSceneDepth(ScreenAlignedPosition(
14        GetScreenPosition(MaterialParameters)));
15
16    // Calcola il forward vector della camera in
17    // coordinate mondiali.
18    float3 CameraFWDVecWorld = mul(
19        float3(0.00000000, 0.00000000, 1.00000000),
20        ResolvedView.ViewToTranslatedWorld);
21
22    // Calcola il vettore che dal centro della camera punta verso
23    // la profondita' della scena.
24    float3 WorldDepthVec = normalize(
25        MaterialParameters.CameraVector) * LocalSceneDepth;
26
27    // Trasforma il vettore in coordinate locali del volume.
28    // GetPrimitiveData: restituisce un riferimento al volume.
29    // LWCHackToFloat: il metodo converte le coordinate da
30    // LargeWordCoordinate a Float.
31    WorldDepthVec = mul(WorldDepthVec,
32        LWCHackToFloat(GetPrimitiveData(
33            MaterialParameters.PrimitiveId).WorldToLocal));
34
35    // Calcola la profondita' della scena in coordinate locali.
36    LocalSceneDepth = length(WorldDepthVec);
```

```

37
38 // Si deve considerare la prondita' della camera
39 // rispetto al pixel.
40 LocalSceneDepth /= abs(dot(CameraFWDVecWorld,
41     MaterialParameters.CameraVector));
42
43 // Calcola la posizione e il vettore della camera in
44 // coordinate locali.
45 float3 LocalCamPos = mul(float4(LWCHackToFloat(
46     ResolvedView.WorldCameraOrigin), 1.00000000),
47     LWCHackToFloat(GetPrimitiveData(
48         MaterialParameters.PrimitiveId).WorldToLocal)).xyz;
49 float3 LocalCamVec = -normalize(mul(
50     MaterialParameters.CameraVector,
51     LWCHackToFloat(GetPrimitiveData(
52         MaterialParameters.PrimitiveId).WorldToLocal)));
53
54 // Trasforma la posizione della camera da coordinate
55 // locali a coordinate UVW (da +-0.5 a [0 - 1]).
56 LocalCamPos += 0.5;
57
58 // Calcola le distanze che il raggio percorre per entrare
59 // e per uscire dal volume.
60 float2 EntryExitTimes = RayAABBIntersection(
61     LocalCamPos, LocalCamVec, 0, 1);
62 // Ci si assicura che il punto d'ingresso non sia
63 // dietro la camera.
64 EntryExitTimes.x = max(0, EntryExitTimes.x);
65 // Ci si assicura che il punto d'uscita non sia occluso da
66 // altre geometrie. Puo' accadere quando due oggetti di
67 // scena si intersecano.
68 EntryExitTimes.y = min(LocalSceneDepth, EntryExitTimes.y);
69 // Calcola la thickness del volume.
70 float BoxThickness = max(0,
71     EntryExitTimes.y - EntryExitTimes.x);
72 // Calcola il punto d'ingresso nel volume in coordinate UVW.
73 float3 EntryPos = LocalCamPos +
74     (EntryExitTimes.x * LocalCamVec);
75
76 return float4(EntryPos, BoxThickness);
77 }

```

RayAABBIntersection

```
1 // Restituisce le distanze che il raggio percorre per
2 // entrare ed uscire dal volume. Se il raggio non
3 // interseca il volume, il metodo restituisce 0.0f per
4 // entrambe le distanze. Se RayOrigin e' interno al volume,
5 // il metodo restituisce 0.0f in result.x e restituisce
6 // la distanza che il raggio percorre per uscire dal
7 // volume in result.y.
8 float2 RayAABBIntersection(float3 RayOrigin, float3 RayDir,
9     float3 BoxMin, float3 BoxMax)
10 {
11     // Si ottiene l'inverso del vettore RayDir.
12     // E' necessario per ordinare gli assi intersecati.
13     float3 InverseRayDir = 1.0 / RayDir;
14
15     // Calcola la distanza percorsa dal raggio prima di
16     // raggiungere BoxMin per il rispettivo asse.
17     float3 TimeToAxisMin = (BoxMin - RayOrigin) * InverseRayDir;
18     // Calcola la distanza percorsa dal raggio prima di
19     // raggiungere BoxMax per il rispettivo asse.
20     float3 TimeToAxisMax = (BoxMax - RayOrigin) * InverseRayDir;
21     // Le distanze sono negative se l'intersezione con
22     // l'asse avviene dietro la camera.
23
24     // Calcola l'intersezione piu' vicina e piu' lontana.
25     float3 ClosestIntersections = min(TimeToAxisMax, TimeToAxisMin);
26     float3 FurthestIntersections = max(TimeToAxisMax,
27         TimeToAxisMin);
28
29     // t0 = ingresso = la piu' lontana tra le intersezioni
30     // piu' vicine.
31     const float t0 = max(ClosestIntersections.x, max(
32         ClosestIntersections.y, ClosestIntersections.z));
33     // t1 = uscita = la piu' vicina tra le intersezioni
34     // piu' lontane.
35     const float t1 = min(FurthestIntersections.x, min(
36         FurthestIntersections.y, FurthestIntersections.z));
37
38     return float2(t0, t1);
39 }
```

PerformWindowedLitRaymarch

```
1 float4 PerformWindowedLitRaymarch(
2     Texture3D DataVolume, // Texture del volume.
3     SamplerState DataVolumeSampler, // Impostazioni che indicano
4     // come il volume e' stato campionato.
5     Texture2D TF, // Texture della funzione di trasferimento.
6     Texture3D LightVolume, // Texture dell'illuminazione.
7     float3 CurPos, // CurPos = Entry Position.
8     float Thickness, // Spessore del volume che il raggio
9     // deve attraversare.
10    float StepCount, // StepCount * Thickness e' il
11    // numero di passi che l'algoritmo esegue.
12    float3 ClippingCenter, // Posizione del piano di clipping.
13    float3 ClippingDirection, // Orientamento del piano di clipping.
14    float4 WindowingParams, // WindowingParams.x == Center
15    // WindowingParams.y == Width.
16    // Gli altri due valori non sono utilizzati.
17    FMaterialPixelParameters MaterialParameters)
18    // Parametri messi a disposizione dal pixel shader.
19 {
20     // Calcola la grandezza dei passi nello spazio UVW.
21     float StepSize = 1 / StepCount;
22
23     // Numero di passi che l'algoritmo esegue.
24     float FloatActualSteps = StepCount * Thickness;
25
26     // Numero di passi interi da eseguire.
27     int MaxSteps = floor(FloatActualSteps);
28
29     // L'ultimo passo potrebbe essere una frazione
30     // di un passo intero.
31     float FinalStep = frac(FloatActualSteps);
32
33     // Calcola il vettore della camera in coordinate locali.
34     float3 LocalCamVec = -normalize(mul(
35         MaterialParameters.CameraVector,
36         LWCHackToFloat(GetPrimitiveData(
37             MaterialParameters.PrimitiveId).WorldToLocal)));
38     LocalCamVec = LocalCamVec * StepSize;
39
40     // Calcola la dimensione del passo in unita' locali. E'
```

```

41 // fondamentale affinche' l'opacita' calcolata dai
42 // compute shaders di UE sia consistente.
43 // VOLUME_DENSITY == 100.0f e' una macro definita
44 // dall'autore.
45 float StepSizeWorld = VOLUME_DENSITY * StepSize;
46
47 // Inizializzazione della variabile.
48 float4 LightEnergy = 0;
49
50 // CurPos viene spostata di una quantita' randomica verso la
51 // posizione della camera lungo la direzione definita da
52 // LocalCamVec. Omettendo questo passaggio e iniziando a
53 // campionare il volume in prossimita' del parallelepipedo
54 // contenente la texture volumetrica, si creano dei pattern
55 // di Moire' che rivelano la geometria.
56 JitterEntryPos(CurPos, LocalCamVec, MaterialParameters);
57
58 int i = 0;
59 for (i = 0; i < MaxSteps; i++)
60 {
61     CurPos += LocalCamVec;
62     // Ogni posizione esclusa dal piano di clipping
63     // viene ignorata.
64     if (!IsCurPosClipped(
65         CurPos, ClippingCenter, ClippingDirection))
66     {
67         // Esegue un passo di raymarching.
68         AccumulateWindowedRaymarchStep(LightEnergy, CurPos,
69             DataVolume, DataVolumeSampler, TF, LightVolume,
70             StepSizeWorld, WindowingParams);
71
72         // Il processo viene interrotto quando l'opacita'
73         // supera una certa soglia -> Early Ray Termination.
74         if (LightEnergy.a > 0.95f)
75         {
76             LightEnergy.a = 1.0f;
77             break;
78         };
79     }
80 }

```

```

82 // Esecuzione dell'ultimo passo.
83 // Viene eseguito solo se tutti i passi precedenti sono stati
84 // completati e se la sua dimensione e' maggiore di 0.
85 if (i == MaxSteps && FinalStep > 0.0f)
86 {
87     CurPos += LocalCamVec * (FinalStep);
88     // L'ultimo passo viene ignorato se la sua posizione e'
89     // esclusa dal piano di clipping.
90     if (!IsCurPosClipped(
91         CurPos, ClippingCenter, ClippingDirection))
92     {
93         AccumulateWindowedRaymarchStep(LightEnergy, CurPos,
94             DataVolume, DataVolumeSampler, TF, LightVolume,
95             VOLUME_DENSITY * FinalStep, WindowingParams);
96     }
97 }
98
99 return LightEnergy;
100}
101
102 // Sposta EntryPos di una quantita' randomica verso la posizione
103 // della camera lungo la direzione definita da LocalCamVec.
104 void JitterEntryPos(inout float3 EntryPos, float3 LocalCamVec,
105 FMaterialPixelParameters MaterialParameters)
106 {
107     int3 RandomPos = int3(MaterialParameters.SvPosition.xy,
108     View.StateFrameIndexMod8);
109     float rand = float(Rand3DPCG16(RandomPos).x) / 0xffff;
110     EntryPos -= LocalCamVec * rand;
111 }
112
113 // Restituisce True se CurPos e' esclusa dal piano di clipping
114 // definito da ClippingCenter e ClippingDirection.
115 bool IsCurPosClipped(float3 CurPos, float3 ClippingCenter,
116 float3 ClippingDirection)
117 {
118     return (dot(CurPos - ClippingCenter, ClippingDirection) <= 0.0);
119 }
```

AccumulateWindowedRaymarchStep

```
1 // Esegue un passo di raymarching e accumula l'energia
2 // stimata.
3 void AccumulateWindowedRaymarchStep(
4     inout float4 AccumulatedLightEnergy, float3 CurPos,
5     Texture3D DataVolume, SamplerState DataVolumeSampler,
6     Texture2D TF, Texture3D LightVolume, float StepSize,
7     float4 WindowingParams)
8 {
9     // Material.Clamp_WorldGroupSettings e' un sampler messo a
10    // disposizione dal pixel shader.
11    float4 ColorSample = SampleWindowedVolumeStep(CurPos,
12        StepSize, DataVolume, DataVolumeSampler, TF,
13        Material.Clamp_WorldGroupSettings, WindowingParams);
14
15    // Si deve regolare l'intensita' in accordo all'illuminazione
16    // della scena.
17    ColorSample.rgb = ColorSample.rgb *
18        LightVolume.SampleLevel(Material.Wrap_WorldGroupSettings,
19            saturate(CurPos), 0).r;
20
21    AccumulateLightEnergy(AccumulatedLightEnergy, ColorSample);
22 }
23
24 // Campiona il volume e lo trasforma in accordo ai parametri
25 // della finestra e alla funzione di trasferimento.
26 float4 SampleWindowedVolumeStep(float3 CurPos, float StepSize,
27     Texture3D Volume, SamplerState VolumeSampler, Texture2D TF,
28     SamplerState TFSampler, float4 WindowingParams)
29 {
30     const float DataValue = Volume.SampleLevel(VolumeSampler,
31         CurPos, 0).r;
32
33     // WindowingParams.x == Center, WindowingParams.y == Width.
34     float TFPoS = GetTransferFuncPosition(DataValue,
35         WindowingParams.x, WindowingParams.y);
36
37     // WindowingParams.z == high cutoff
38     // WindowingParams.w == low cutoff
39     // Se TFPoS e' al di fuori del range definito dalle
40     // impostazioni di cutoff, l'energia totale e' 0.
```

```

41     if ((TFPos < 0.0 && WindowingParams.z > 0.0) ||
42         (TFPos > 1.0 && WindowingParams.w > 0.0))
43     {
44         return float4(0, 0, 0, 0);
45     }
46
47     float4 ColorSample = TF.SampleLevel(TFSampler,
48                                         float2(TFPos, 0.5), 0);
49
50     // L'opacita' viene normalizzata.
51     ColorSample.a = saturate(ColorSample.a);
52
53     // La seguente trasformazione e' essenziale per regolare
54     // l'opacita' in accordo al numero di passi di raymarching
55     // che si eseguono. Se omessa, eseguendo troppi passi
56     // si otterebbe un volume completamente opaco. Al contrario,
57     // eseguendone pochi si otterebbe un volume completamente
58     // trasparente.
59     ColorSample.a = 1.0 - pow(1.0 - ColorSample.a, StepSize);
60
61     return ColorSample;
62 }
63
64 float GetTransferFuncPosition(float Value,
65                                float WindowCenter, float WindowWidth)
66 {
67     return (Value - WindowCenter + (WindowWidth / 2.0))
68     / WindowWidth;
69 }
70
71 // Accumulo con over-operator.
72 void AccumulateLightEnergy(inout float4 LightEnergy,
73                            in float4 CurrentSample)
74 {
75     LightEnergy.rgb = LightEnergy.rgb +
76                     (CurrentSample.rgb * CurrentSample.a *
77                      (1.0 - LightEnergy.a));
78
79     LightEnergy.a = LightEnergy.a +
80                     (CurrentSample.a * (1.0 - LightEnergy.a));
81 }
```


5 TBRaymarcherPlugin: un Nuovo Modello d’Illuminazione

Gli obiettivi principali nel campo della visualizzazione immersiva sono due: la fluidità della simulazione e ottenere un alto grado di realismo nei dettagli visualizzati. Attualmente, il primo obiettivo è soddisfatto dall’implementazione del sistema di rendering integrato nel plugin di Tomas Bartipan. Pertanto, in questo capitolo, l’attenzione verrà posta sul secondo obiettivo.

Come menzionato in precedenza, il modello ottico utilizzato dal Ray Marcher implementato da Tomas Bartipan in [Bar] è semplice e non tiene conto di due fenomeni cruciali legati all’interazione della luce con un volume: l’assorbimento e la dispersione. Questi due fenomeni, ampiamente spiegati nel paragrafo 2.2.2, sono stati tenuti in considerazione nel nuovo modello proposto in questo elaborato. Il modello presentato si basa fortemente sulle nozioni descritte all’interno della guida offerta da ScratchAPixel, intitolata "Volume Rendering" [Scr09a] [Scr09d] [Scr09c] [Scr09b].

È risultato necessario apportare modifiche sia al grafo del materiale M_Raymarch sia ad alcuni degli shader descritti nel paragrafo 4.1.2. In primo luogo, nel grafo dei materiali, è stata effettuata una modifica affinché il nodo Windowed Intensity Raymarch riceva in input le posizioni di tutte le sorgenti luminose presenti nella scena. Queste informazioni sono cruciali all’interno della funzione di libreria PerformWindowedLitRaymarch per valutare l’amplificazione luminosa dovuta all’in-scattering. Per quanto riguarda gli shader, molte funzioni non sono state modificate e sono state riutilizzate nel nuovo modello, tra cui:

- PerformRaymarchCubeSetup;
- RayAABBIntersection;
- JitterEntryPos;
- IsCurPosClipped;

Al contrario, le funzioni responsabili della valutazione dell’energia luminosa da assegnare ai pixel sono state nettamente modificate.

5.1 I Nuovi Shader

In questo paragrafo vengono approfondite le funzioni di shading riviste. Il nuovo modello ottico tiene conto degli effetti di assorbimento e dispersione.

NewPerformWindowedLitRaymarch

Il metodo, in primo luogo, calcola la direzione e l'entità dei passi di Ray Marching. Similmente al modello originale, sposta il punto di partenza del campionamento di una quantità casuale per evitare che la geometria contenente i dati generi artefatti visivi. Successivamente, per ogni passo, viene richiamata la funzione NewAccumulateWindowedRaymarchStep. Questa funzione aggiorna l'energia luminosa accumulata e la trasmissione (complemento rispetto a 1 dell'opacità). Il ciclo si interrompe se la trasmissione scende sotto una soglia predefinita, realizzando così la Early Ray Termination.

```
1 float4 NewPerformWindowedLitRaymarch(Texture3D DataVolume,
2     SamplerState DataVolumeSampler, Texture2D TF, Texture3D
3     LightVolume, float3 CurPos, float Thickness, float StepCount,
4     float3 ClippingCenter, float3 ClippingDirection,
5     float4 WindowingParams,
6     float3 LightPos1, // Posizione della sorgente luminosa.
7     float3 LightPos2, // Posizione della sorgente luminosa.
8     float3 LightPos3, // Posizione della sorgente luminosa.
9     float ShadowStepCount, // Massimo numero di passi che
// l'algoritmo esegue per calcolare l'amplificazione luminosa
// derivante dall'in-scattering.
10    FMaterialPixelParameters MaterialParameters)
11 {
12     // Calcola il numero e la grandezza dei passi di raymarching,
13     // nonche' la direzione di movimento. Queste istruzioni
14     // rimangono invariati rispetto alla versione proposta nel
15     // plugin.
16     float StepSize = 1 / StepCount;
17     float FloatActualSteps = StepCount * Thickness;
18     int MaxSteps = floor(FloatActualSteps);
19     float FinalStep = frac(FloatActualSteps);
20     float3 LocalCamVec = -normalize(mul(
21         MaterialParameters.CameraVector,
22         LWCHackToFloat(GetPrimitiveData(
23             MaterialParameters.PrimitiveId).WorldToLocal)));
24 }
```

```

26 LocalCamVec *= StepSize;
27 StepSize *= VOLUME_DENSITY;
28 // VOLUME_DENSITY == 1100.0f.
29 // La macro e' stata modificata rispetto la versione
30 // originale per ottenere una qualita' superiore.
31
32 // Inizializzazione delle variabili.
33 float3x3 LightsPos = float3x3(LightPos1, LightPos2,
34     LightPos3);
35 float3 LightEnergy = (0, 0, 0); // rgb
36 // Il modello proposto nel plugin lavora utilizzando
37 // l'opacita'. In questa variante si lavora con il suo
38 // complemento rispetto a 1.
39 float Transmission = 1; // = 1 - a
40 // Si suppone che il volume sia isotropo. E' una grandissima
41 // approssimazione dato che l'obiettivo e' renderizzare
42 // volumi medici.
43 float sigma_a = 0.05; // Coefficiente di assorbimento.
44 float sigma_s = 0.10; // Coefficiente di scattering.
45 // Coefficiente di estinzione.
46 float sigma_t = sigma_a + sigma_s;
47 // Aumentare i valori porta ad avere un volume piu' luminoso e
48 // rumoroso. Questo perche' si sta lavorando con il
49 // complemento rispetto a 1 dell'opacita'.
50 float G = 0; // Fattore di asimmetria di Henyey-Greenstein.
51 float ShadowOpacity = 0.5;
52
53 JitterEntryPos(CurPos, LocalCamVec, MaterialParameters);
54
55 // Calcola la grandezza dei passi di in-scattering
56 // nello spazio UVW.
57 float ShadowStepSize = 1 / ShadowStepCount;
58 ShadowStepSize *= VOLUME_DENSITY;
59
60 // Ciclo che esegue gli step di raymarching.
61 int i = 0;
62 for (i = 0; i < MaxSteps; i++)
63 {
64     CurPos += LocalCamVec;
65
66     // Ogni posizione esclusa dal piano di clipping

```

```

67     // viene ignorata.
68     if (!IsCurPosClipped(
69         CurPos, ClippingCenter, ClippingDirection))
70     {
71         // Esegue un passo di raymarching .
72         NewAccumulateWindowedRaymarchStep(LightEnergy, Transmission,
73             DataVolume, TF, LightVolume, DataVolumeSampler,
74             WindowingParams, LightsPos, CurPos, sigma_t,
75             StepSize, sigma_s, G, ShadowStepCount, ShadowStepSize,
76             ShadowOpacity, MaterialParameters);
77
78         // Il processo viene interrotto quando
79         // Transmission scende sotto una certa
80         // soglia -> Early Ray Termination.
81         if (Transmission < 0.01)
82         {
83             Transmission = 0;
84             break;
85         }
86     }
87
88
89 // Esecuzione dell'ultimo passo.
90 // Viene eseguito solo se tutti i passi precedenti sono stati
91 // completati e se la sua dimensione e' maggiore di 0.
92 if (i == MaxSteps && FinalStep > 0.0f)
93 {
94     CurPos += LocalCamVec * (FinalStep);
95
96     // L'ultimo passo viene ignorato se la sua posizione e'
97     // esclusa dal piano di clipping.
98     if (!IsCurPosClipped(
99         CurPos, ClippingCenter, ClippingDirection))
100    {
101        // Esegue un passo di raymarching .
102        NewAccumulateWindowedRaymarchStep(LightEnergy, Transmission,
103            DataVolume, TF, LightVolume, DataVolumeSampler,
104            WindowingParams, LightsPos, CurPos, sigma_t,
105            VOLUME_DENSITY * FinalStep, sigma_s, G, ShadowStepCount,
106            ShadowStepSize, ShadowOpacity, MaterialParameters);
107    }

```

```

108 }
109 return float4(LightEnergy, 1-Transmission);
110 }
```

NewAccumulateWindowedRaymarchStep

Dato un punto, il metodo campiona il dato volumetrico associato alla sua posizione. Utilizzando la funzione di trasferimento e le informazioni relative all'illuminazione della scena, determina il colore e l'opacità nel punto di campionamento. La trasmissione viene corretta secondo la legge di Beer-Lambert, descritta nel paragrafo 2.2.2, per tenere in considerazione i fenomeni di assorbimento e out-scattering. Attraverso un ciclo, si esegue un'iterazione per ogni sorgente luminosa al fine di calcolare l'amplificazione luminosa derivante dall'in-scattering. Infine, l'energia luminosa viene accumulata.

```

1 void NewAccumulateWindowedRaymarchStep(inout float3 LightEnergy,
2   inout float Transmission, Texture3D DataVolume, Texture2D TF,
3   Texture3D LightVolume, SamplerState DataVolumeSampler,
4   float4 WindowingParams, float3x3 LightsPos, float3
5   CurPos, float sigma_t, float StepSize, float sigma_s, float G,
6   float ShadowStepCount, float ShadowStepSize, float
7   ShadowOpacity, FMaterialPixelParameters MaterialParameters)
8 {
9   float4 ColorSample = SampleWindowedVolumeStep(CurPos,
10    StepSize, DataVolume, DataVolumeSampler, TF,
11    Material.Clamp_WorldGroupSettings, WindowingParams);
12
13 // Si deve regolare l'intensità in accordo
14 // all'illuminazione della scena.
15 ColorSample.rgb = ColorSample.rgb *
16   LightVolume.SampleLevel(Material.Wrap_WorldGroupSettings,
17   saturate(CurPos), 0).r;
18
19 // attenuazione dovuta ad assorbimento e out-scattering.
20 Transmission *=
21   exp(
22     -StepSize * saturate(ColorSample.a) * sigma_t
23   );
24
25 // Ciclo per calcolare l'amplificazione luminosa
26 // dovuta all'in-scattering.
27 float DensityLight = 0;
```

```

28     float LightAmp = 0;
29     for (int l = 0; l < 3; l++)
30     {
31         float3 LPos = CurPos;
32
33         // Calcola la posizione della camera nello
34         // spazio mondiale.
35         float3 CamPosWorld = LWCHackToFloat(
36             ResolvedView.WorldCameraOrigin);
37
38         // Calcola il vettore LPos-to-light in
39         // coordinate mondiali.
40         float3 PosToLightWorld = LightsPos[l] - CamPosWorld;
41
42         // Il vettore viene trasformato in coordinate locali.
43         float3 PosToLight = normalize(mul(PosToLightWorld,
44             LWCHackToFloat(GetPrimitiveData(
45                 MaterialParameters.PrimitiveId).WorldToLocal)));
46         PosToLight *= ShadowStepSize;
47
48         // Calcola la distanza che il raggio percorre per
49         // uscire dal volume.
50         float ExitTime = RayAABBIntersection(CurPos,
51             PosToLight, 0, 1).y;
52
53         // Ci si assicura che il punto d'uscita non sia occluso da
54         // altre geometrie. Puo' accadere quando due oggetti di
55         // scena si intersecano.
56         float LocalSceneDepth = CalcSceneDepth(ScreenAlignedPosition(
57             GetScreenPosition(MaterialParameters)));
58         float3 CameraFWDVecWorld = mul(
59             float3(0.00000000, 0.00000000, 1.00000000),
60             ResolvedView.ViewToTranslatedWorld);
61         float3 WorldDepthVec = normalize(
62             MaterialParameters.CameraVector) * LocalSceneDepth;
63         WorldDepthVec = mul(WorldDepthVec,
64             LWCHackToFloat(GetPrimitiveData(
65                 MaterialParameters.PrimitiveId).WorldToLocal));
66         LocalSceneDepth = length(WorldDepthVec);
67         LocalSceneDepth /= abs(dot(CameraFWDVecWorld,
68             MaterialParameters.CameraVector));

```

```

69     ExitTime = min(LocalSceneDepth, ExitTime);
70
71     // Calcola la thickness del volume.
72     float ThicknessForScattering = max(0, ExitTime);
73
74     // Numero di passi interi da eseguire per valutare
75     // l'in-scattering.
76     int MaxShadowStep = floor(ShadowStepCount *
77                               ThicknessForScattering);
78
79     // accumulo l'amplificazione luminosa portata dai
80     // campioni lungo lo shadow ray.
81     for (int s = 0; s < MaxShadowStep; s++)
82     {
83         LPos += PostToLight;
84
85         DensityLight += LightVolume.SampleLevel(
86             Material.Wrap_WorldGroupSettings, saturate(LPos), 0).r;
87     }
88     LightAmp +=
89         exp(
90             -DensityLight * ShadowStepSize * sigma_t * ShadowOpacity
91             ) *
92         PhaseHG(
93             -MaterialParameters.CameraVector, PostToLight, G
94             );
95
96     }
97     LightEnergy += ColorSample.rgb * LightAmp *
98     sigma_s * StepSize * ColorSample.a;
99 }
100
101 // Henyey-Greenstein Phase Function
102 float PhaseHG(float3 CameraVector, float3 CamToLightVector,
103                float G)
104 {
105     float CosTheta = dot(CameraVector, CamToLightVector);
106     return (1 / (4 * PI)) *
107         ((1 - G * G) / pow(1 + G * G - 2 * G * CosTheta, 1.5));
108 }
```


6 I due Modelli: Confronto Quantitativo

Il confronto quantitativo sarà effettuato analizzando diverse quantità che definiscono la complessità computazionale degli shader, la complessità del materiale e i tempi di esecuzione in interazioni tipiche.

La complessità di un materiale è definita dal numero di nodi che compongono il suo grafo e dalla quantità e complessità delle istruzioni matematiche che il grafo deve eseguire. Più istruzioni sono necessarie per la resa del materiale, maggiore sarà il tempo richiesto per calcolare il valore finale di un pixel. Purtroppo, i materiali traslucidi sono tra i tipi di materiali più onerosi da renderizzare. In Unreal Engine, la complessità viene misurata in istruzioni shader: affinché la scena possa essere eseguita senza problemi in tempo reale, il numero di istruzioni dovrebbe aggirarsi intorno a qualche centinaio. Il materiale offerto dal TBRaymarcherPlugin ha una complessità di 236 istruzioni shader, mentre il materiale proposto per la sua ottimizzazione ha una complessità di 325 istruzioni shader.

Per quanto riguarda l'occupazione di memoria del materiale stesso, il grafo di un materiale può gestire al massimo 16 texture sampler, di cui solo 13 sono disponibili all'utente (i restanti sono utilizzati da Unreal Engine). Questo vuol dire che in un grafo possono esserci al massimo 13 texture che non condividono un sampler tra di loro. Più texture possiede un materiale, maggiore sarà la memoria che esso occuperà. Dal punto di vista delle prestazioni, un numero elevato di texture potrebbe causare blocchi e ritardi quando il materiale viene caricato sulla GPU. Entrambi i materiali sono composti da 3 texture e dai loro rispettivi sampler. Queste texture sono: il nodo Volume, il nodo TransferFunction e il nodo ALightVolume.

Distaccandosi dalle quantità fornite da Unreal Engine, è possibile ragionare in termini di complessità computazionale del codice che definisce gli shader. Come osservabile dai metodi riportati nel paragrafo 4.1.2, il Ray Marcher di Tomas Bartipan esegue m passi di Ray Marching per ogni pixel che si sovrappone al volume. Durante questi passi, vengono eseguite istruzioni singole per calcolare il colore e l'opacità da assegnare al pixel. Questo si traduce in una complessità finale di $\mathcal{O}(n_pixels \cdot m_raymarching_steps)$ per renderizzare un fotogramma.

Nell'ottimizzazione proposta, per ogni pixel che si sovrappone al volume, con-

tinuano ad essere eseguiti m passi di Ray Marching. Tuttavia, per ogni passo, devono essere considerate le p sorgenti di illuminazione presenti nella scena, e per ciascuna di esse bisogna eseguire q iterazioni per calcolare l'amplificazione luminosa dovuta all'in-scattering. Questo porta ad una complessità finale di $\mathcal{O}(n_pixels \cdot m_raymarching_steps \cdot p_light_sources \cdot q_shadow_steps)$ per renderizzare un fotogramma.

Sono state ricavate diverse statistiche da due esecuzioni, durante le quali sono state simulate interazioni tipiche con il volume e gli oggetti di scena. Le esecuzioni, della durata di 3 minuti ciascuna, sono state effettuate su una macchina con le seguenti caratteristiche:

- RAM 32gb ddr4;
- Intel(R) Core(TM) i7-7820X CPU @ 3.60GHz 3.60 GHz;
- NVIDIA GeForce RTX 2080ti.

I parametri utilizzati per simulare le due interazioni sono i seguenti:

- 3 sorgenti d'illuminazione;
- 128 passi di Ray Marching;
- 32 passi per calcolare l'amplificazione luminosa dovuta all'in-scattering.

I dati riportati rappresentano i valori medi ottenuti durante le esecuzioni.

	TBRaymarcherPlugin	Ottimizzazione Proposta
FPS	36.92	34.94
Shader Compiling Thread	30.88 ms	33.60 ms
Game Thread	30.91 ms	33.59 ms
Rendering Thread	30.86 ms	33.54 ms
Pixel Shader Memory	9242.78 KB	9265.32 KB
Vertex Shader Memory	728.96 KB	766.63 KB

7 I due Modelli: Confronto Qualitativo

Riportare un confronto diretto tra modelli volumetrici è risultato impraticabile, poiché per una corretta comprensione è necessario osservarli direttamente tramite un visore dedicato. Pertanto, è stato effettuato un confronto qualitativo basato su diverse acquisizioni della resa dei due modelli.

Per effettuare un confronto qualitativo tra immagini è possibile utilizzare diverse tecniche. Queste, vengono suddivise principalmente in due categorie: metriche e valutazioni visive. Le metriche includono il calcolo della somiglianza, la stima della percentuale di rumore presente e la valutazione della precisione nella rappresentazione dei contorni. D'altra parte, le valutazioni visive si concentrano su aspetti come nitidezza, contrasto, qualità della luce e quantità di rumore percepito dall'occhio umano.

Nel contesto specifico di questo studio, l'applicazione di metriche quantitative non è risultata significativa, in quanto la nuova implementazione proposta ha modificato precisamente gli aspetti di nitidezza, contrasto e qualità della luce. Inoltre, il confronto è stato progettato per essere proiettato sui modelli 3D. Le metriche, al contrario, eseguono molto spesso confronti pixel a pixel, che non sono in grado di catturare le vere modifiche apportate dal modello proposto in questa tesi.

Per queste ragioni, il confronto delle immagini sarà condotto focalizzandosi esclusivamente sugli aspetti visivi tramite un confronto diretto. Questo confronto preliminare è svolto da un individuo privo di preparazione medica e le conclusioni riportate nei seguenti esempi sono presentate come oggettive. È possibile che vi siano opinioni divergenti riguardo ai confronti eseguiti, il che è comprensibile poiché la percezione dei dettagli può variare da persona a persona. Si prevede, in future iterazioni, di raccogliere le opinioni di esperti del settore per ottenere una valutazione più accurata.

Il primo esempio mira a dimostrare come la considerazione di due fenomeni fondamentali legati all'interazione della luce con un volume, quali l'assorbimento e la dispersione, migliori la percezione della tridimensionalità degli oggetti osservati dall'occhio umano.

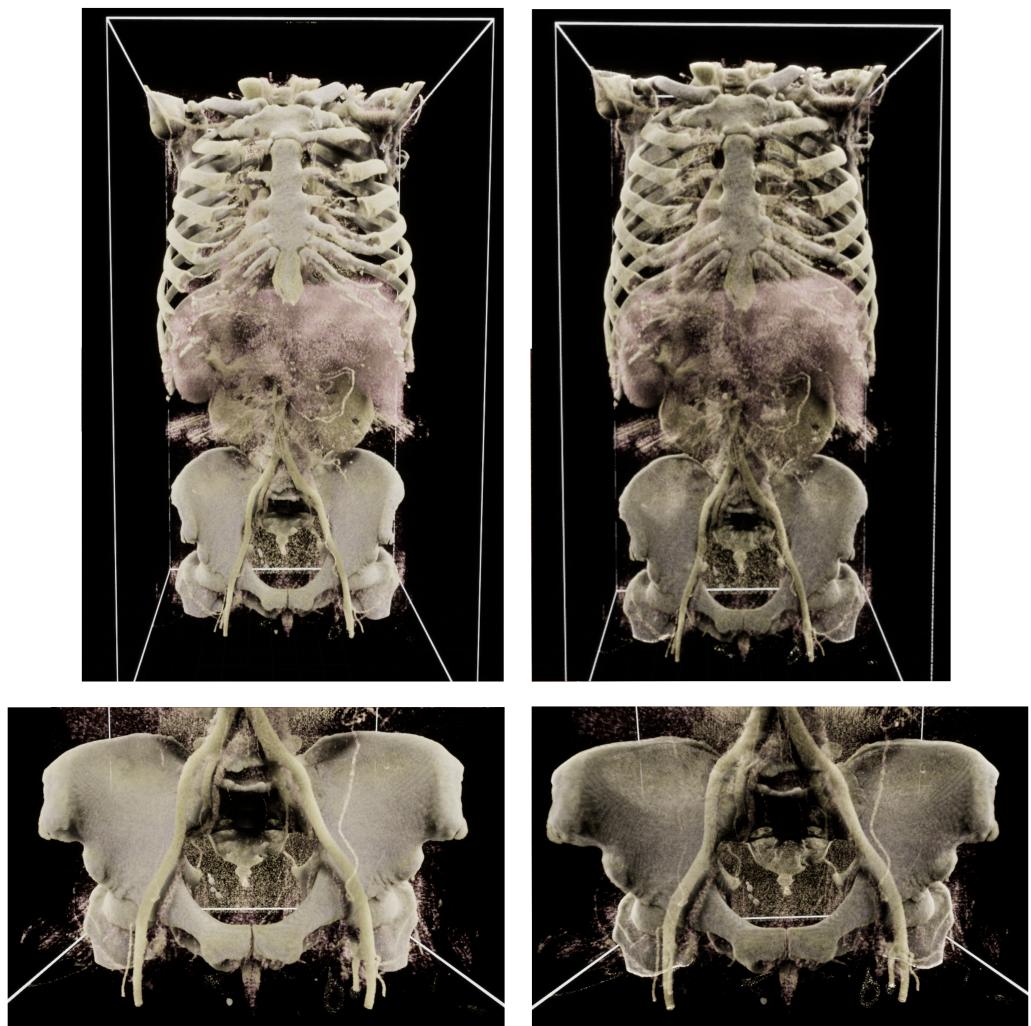


Figura 9: Confronto qualitativo: esempio 1. Fonte: immagine autoprodotta.

Le immagini mostrano uno scheletro renderizzato: a sinistra, utilizzando il modello di illuminazione integrato nel TBRaymarcherPlugin, a destra, attraverso il nuovo modello proposto. Nel dettaglio di sinistra, l'osso iliaco è completamente illuminato, risultando piatto. Nel dettaglio di destra, l'osso iliaco presenta ombre morbide che descrivono meglio la sua natura concava. Lo stesso ragionamento può essere applicato alla resa delle arterie iliache.

Il secondo esempio prende in esame un dettaglio del cuore.

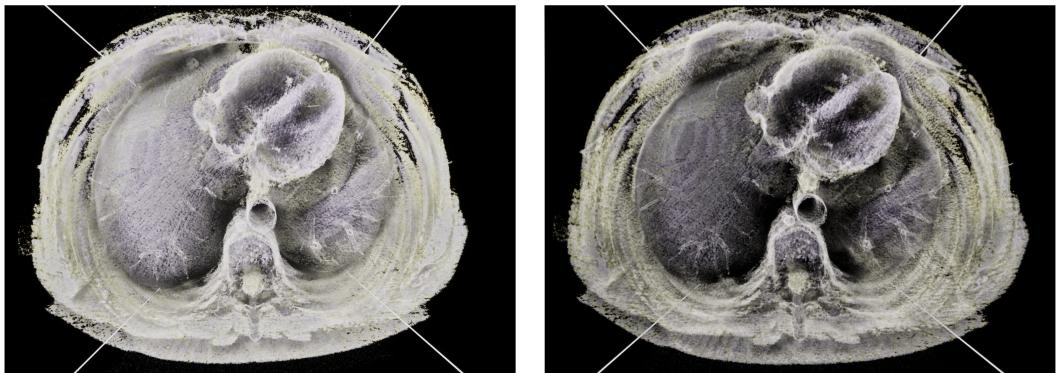


Figura 10: Confronto qualitativo: esempio 2. Fonte: immagine autoprodotta.

Nelle due immagini, renderizzate rispettivamente attraverso il TBRaymarcher-Plugin e attraverso il nuovo modello proposto, è possibile osservare una sezione dell'aorta discendente, il ventricolo destro ed il ventricolo sinistro. Nell'immagine di sinistra le ombre sono minime e la divisione dei ventricoli non è netta. Nell'immagine di destra, le ombre permettono una chiara divisione dei ventricoli e, soprattutto, come per l'esempio precedente, rendono più dettagliatamente la loro profondità.

Infine, viene riportato un esempio in cui il modello di illuminazione proposto risulta peggiore, in termini di confronto visivo diretto, rispetto al modello integrato nel TBRaymarcherPlugin.

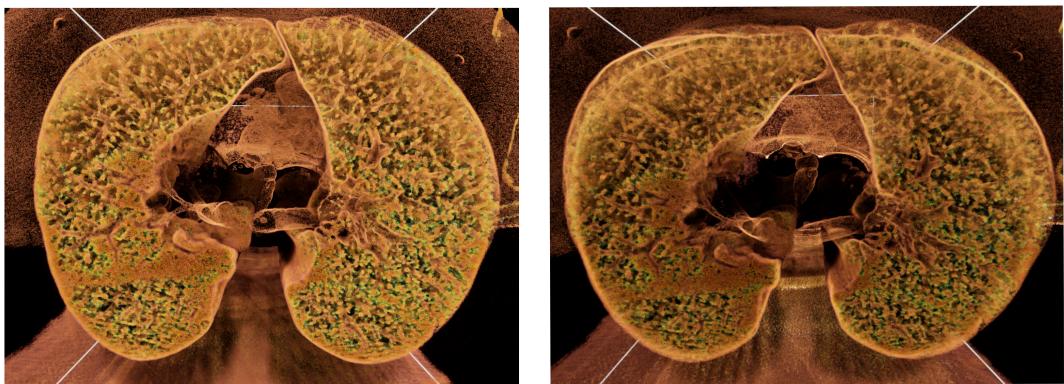


Figura 11: Confronto qualitativo: esempio 3. Fonte: immagine autoprodotta.

Le immagini mostrano un dettaglio della struttura polmonare, in particolare sono visibili i bronchi, i bronchioli e gli alveoli polmonari. L'immagine di sinistra è stata

renderizzata utilizzando il modello di illuminazione integrato nel TBRaymarcher-Plugin. L’immagine di destra è stata renderizzata utilizzando il nuovo modello proposto. È evidente come le ombre prodotte dal nuovo modello rendano difficile la visualizzazione e la comprensione della complessa struttura interna dei polmoni.

8 Conclusioni

Le statistiche riportate nel confronto quantitativo dimostrano che la soluzione ottimizzata non è inferiore, in termini di usabilità e prestazioni, rispetto alla soluzione proposta all'interno del TBRaymarcherPlugin. Un numero maggiore di passi porta a una rappresentazione più precisa, ma comporta anche un peggioramento delle prestazioni. Già con 256 passi di Ray Marching, l'interazione con il volume risulta pressoché impossibile.

Il confronto qualitativo ha messo in evidenza come il nuovo modello di illuminazione proposto possa migliorare la percezione della profondità e dei dettagli in specifici contesti volumetrici, come nel caso delle strutture ossee. Tuttavia, in situazioni complesse, le ombre prodotte dal nuovo modello possono oscurare dettagli fondamentali e rendere difficile la visualizzazione e la comprensione delle strutture. In sintesi, i risultati ottenuti dimostrano che il nuovo modello di illuminazione offre vantaggi significativi in termini di resa visiva in alcuni scenari, mentre in altri, la soluzione integrata nel TBRaymarcherPlugin rimane più efficace. Pertanto, è possibile concludere che l'approccio ottimale dipende fortemente dal contesto specifico dell'applicazione.

Per quanto riguarda le future direzioni di ricerca, ci sono diverse aree che potrebbero essere esplorate:

- Ottimizzazione delle prestazioni: è possibile implementare diverse tecniche di ottimizzazione per ridurre i passi di Ray Marching senza compromettere la qualità visiva. Queste sono state ampiamente descritte all'interno del Paragrafo 3.1.1. Un'alternativa è modificare l'intero Ray Marcher utilizzando una delle soluzioni descritte in Appendice.
- Miglioramento della qualità visiva: nel modello di illuminazione proposto, una semplificazione significativa adottata è stata quella di considerare il volume come isotropo, ovvero assumendo che ogni sua componente assorba e disperda la luce in maniera uniforme. Tuttavia, un volume medico è intrinsecamente anisotropo, poiché ogni tessuto, composto da specifici materiali, assorbe e disperde la luce in modo diverso.
- Integrazione del modello nel progetto LunaVIEW: il modello è stato sviluppato successivamente alla conclusione del progetto LunaVIEW e, pertanto,

non è stato integrato al suo interno. Il modello è stato utilizzato esclusivamente per questo studio. Tuttavia, l'integrazione del modello, insieme alle numerose funzionalità già aggiunte al TBRaymarcherPlugin per offrire un'esperienza di visualizzazione e manipolazione superiore, può solo migliorarne ulteriormente la qualità. Le funzionalità con cui il modello deve essere integrato sono:

- Evidenziazione dei noduli polmonari: attraverso il caricamento di una maschera di segmentazione binaria, insieme ai dati volumetrici in formato DICOM, il progetto è in grado di evidenziare, all'interno del volume, i noduli polmonari indicati dalla segmentazione.
- Segmentazione di dettagli anatomici: sebbene il windowing e le funzioni di trasferimento fossero già presenti per classificare e definire quali caratteristiche del volume visualizzare, è stata aggiunta una nuova caratteristica. Attraverso l'utilizzo di una rete convoluzionale [Was] è possibile segmentare i file DICOM per visualizzare esclusivamente determinati dettagli anatomici.
- Navigazione automatica ai noduli: i radiologi sono abituati a esaminare i dati medici attraverso sezioni, pertanto è stato implementato un sistema di navigazione automatica per mostrare le sezioni che includono il centro di ogni nodulo polmonare. Allo stesso modo, al fine di migliorare l'utilità dell'analisi, è stato integrato un sistema in grado di fornire le coordinate assolute in relazione a una vertebra di un nodulo.

Appendice A Volumetric Ray Tracing

È complesso attribuire l'invenzione del Ray Tracing a un singolo autore; nel corso di tre decenni diversi autori hanno concorso alla concezione dell'algoritmo come lo si conosce oggi.

Ad introdurlo per la prima volta è stato Arthur Appel che, nel 1968, in [App68] delinea un metodo per generare immagini bidimensionali di oggetti tridimensionali attraverso il tracciamento di raggi nella scena. Il desiderio di sviluppare un metodo per la generazione automatica di disegni in contesti organizzativi, architettonici e industriali ha stimolato la sua attività di ricerca.

Era ben chiaro che per simulare il realismo di una fotografia fosse necessario tenere in conto diversi aspetti, come: illuminazione diretta e diffusa, diffusione atmosferica, effetto della texture superficiale e trasparenza delle superfici. Purtroppo, la limitata potenza di calcolo degli hardware utilizzati nel 1968 non rendeva possibile tutto ciò. Appel si è quindi limitato al processo di ombreggiatura automatica dei disegni basati su linee. Lo scopo della sua ricerca era di valutare il costo della generazione di tali immagini e la qualità grafica risultante.

Il metodo illustrato da Appel, ai tempi non ancora denominato Ray Tracing ma "Point by Point Shading", prevedeva i seguenti passaggi:

- Determinare le coordinate delle proiezioni dei vertici della figura sul piano immagine, qui chiamato "Picture Plane".
- Generare un elenco di punti interni alle coordinate dei vertici precedentemente calcolati (P_{x_P}). Per ognuno di essi tracciare un raggio che, partendo dall'osservatore, attraversa la scena.
- Per ogni raggio calcolare il punto di intersezione con la figura (P_x).
- Per ogni punto di intersezione determinare se esso è nascosto alla sorgente di illuminazione. Per farlo, bisogna tracciare un raggio che collega il punto di intersezione alla sorgente luminosa: se il raggio non raggiunge la sorgente allora il punto è in ombra.

Schematicamente il processo può essere riassunto come segue:

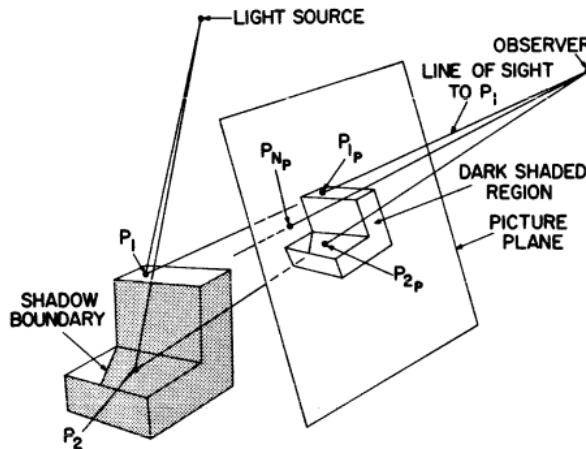


Figura 12: Point by Point Shading.

Fonte: Some techniques for shading machine renderings of solids.

Autore: Arthur Appel.

L'invenzione del Ray Tracing come lo si conosce oggi è comunemente attribuita a Turner Whitted. Nel 1980, nel suo articolo [Whi80] sottolineò che per rappresentare una scena di complessità elevata bisogna tenere conto che la luce, prima di arrivare all'osservatore, può essere riflessa da più superfici. La soluzione di Whitted, a differenza di quella proposta da Appel, non interrompe i calcoli appena un raggio interseca una superficie. Al contrario, a partire dal punto di intersezione, traccia un certo numero di nuovi raggi e l'algoritmo viene richiamato ricorsivamente su di essi. I raggi tracciati sono:

- uno shadow ray per ogni sorgente d'illuminazione che compone la scena. Se quest'ultimo raggiunge la sorgente allora essa contribuisce all'illuminazione del punto, diversamente il punto è in ombra rispetto a quella specifica sorgente;
- un raggio riflesso;
- un raggio rifratto, se la superficie è trasparente o semi-trasparente.

La procedura utilizza una struttura dati ad albero per tenere traccia dei punti di intersezione e stabilire l'ordine in cui devono essere processati:

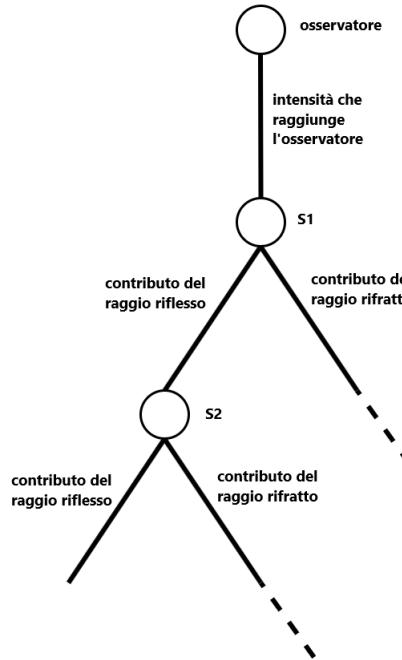


Figura 13: Ray Tracing: albero di ricorsione.

Fonte: immagine autoprodotta.

Creata l'albero, lo shader lo attraversa partendo dalle foglie e risalendo fino alla radice. Per ogni nodo, lo shader valuta l'equazione (3) (questo è quello che prevedeva l'algoritmo originale del 1980, le implementazioni odiere valutano l'equazione (4)).

Bisognerà aspettare il 1995 perché il Ray Tracing venga utilizzato per il rendering di volumi. In [AK95], Arie Kaufman e Lisa Avila illustrano una variante dell'approccio proposto da Whitted per il rendering di voxel.

La variante volumetrica si distingue dai metodi precedentemente descritti in questo capitolo poiché non calcola punti di intersezione tra raggi e superfici, ma segmenti di intersezione definiti da un punto di ingresso nel volume e un punto di uscita. Quando i dati sono ottenuti tramite voxelizzazione di un oggetto geometrico, la descrizione geometrica originale dell'oggetto può essere impiegata per calcolare in modo più preciso i due punti sopraccitati. Al contrario, a meno che non vengano utilizzate delle tecniche di ottimizzazione, l'intersezione di ingresso e di uscita si verifica in prossimità della geometria che contiene i dati. Durante il processo di lighting tutti i voxel vuoti all'interno della griglia dovranno essere processati. Purtroppo, questo è il caso delle scansioni CT, poiché nascono già espresse attraverso

voxel. Come nella soluzione di Whitted, ai punti di campionamento che cadono su un segmento di intersezione deve essere dato un ordine di processamento. Questi punti, rispettando l'ordine, vengono successivamente passati allo shader che valuta il modello ottico. Il processo viene concluso accumulando i valori per formare l'immagine finale.

Vengono proposte due ottimizzazioni per trovare i punti di ingresso e di uscita di un raggio in un volume che non deriva da una voxelizzazione di un oggetto geometrico. I casi sono dettati dal tipo di interpolazione che si è utilizzata per ricostruire il volume e permettono di ignorare i voxel trasparenti.

1. Se il volume è stato interpolato con una tecnica di ordine zero i voxel attraversati dal raggio vengono esaminati uno ad uno. Il punto di ingresso è definito dal primo voxel non trasparente che si incontra, il punto di uscita, invece, è definito dall'ultimo.
2. Se il volume è stato interpolato con una tecnica di ordine superiore esistono due alternative:
 - 2.1. Il raggio viene esaminato ad intervalli regolari e l'analisi è identica a quella precedente.
 - 2.2. Il volume viene diviso in cubi piccoli e discreti. Per ogni sezione del raggio che attraversa una cella cuboide formata da 8 cubi si valuta una funzione per stabilire la tipologia di area che la sezione attraversa e quindi l'azione da eseguire:
 - se la cella è composta da cubi trasparenti si estende la ricerca del punto di ingresso alla sezione di segmento successiva;
 - se la cella è composta da cubi non trasparenti allora il punto di ingresso è già stato trovato e si iniziano ad analizzare le sezioni per individuarne il punto di uscita;
 - se la cella è formata da cubi eterogenei ogni cubo della cella viene promosso e diviso in 8 cubi più piccoli. La ricerca prosegue in profondità finché non si troverà un'esatta suddivisione tra cubi trasparenti e opachi oppure finché non si saranno eseguite troppe iterazioni.

Infine, vengono descritte due alternative per selezionare i punti di campionamento sul segmento di intersezione:

1. Il segmento viene diviso in sezioni e di ognuna viene passato il valore medio allo shader. Si preferisce questa alternativa quando il volume è composto prevalentemente da aree omogenee.
2. Il segmento viene campionato ad intervalli regolari. Si preferisce questa soluzione quando il volume è composto da aree eterogenee e i punti da passare allo shader sono numerosi.

Per i due casi appena illustrati è possibile implementare la Early Ray Termination e passare un numero minore di punti allo shader.

L'equazione valutata dal modello ottico è quella indicata nel paragrafo 2.2.2 e denominata Volume Rendering Equation (10).

A.1 Volumetric Path Tracing

I termini Path Tracing e Ray Tracing vengono spesso confusi, tuttavia rappresentano concetti piuttosto distinti. Sebbene il Path Tracing si rifaccia ai principi del Ray Tracing, è importante notare che non sono processi analoghi. Il Path Tracing è stato introdotto nel 1986 da Jim Kajiya per risolvere alcune problematiche che limitavano l'efficacia del Ray Tracing, rendendolo nella maggioranza dei casi inutilizzabile.

Il problema principale che è stato riscontrato nella soluzione proposta da Whitted era la tendenza del processo di tracciamento dei raggi secondari di degenerare in modo esponenziale. Nel Ray Tracing, un singolo rimbalzo può generare non solo un raggio riflesso e uno rifratto, ma anche un certo numero di shadow ray. A loro volta, i raggi riflesso e rifratto potrebbero intercettare altre superfici, dando origine a ulteriori riflessi, rifrazioni e shadow ray. In una scena complessa, ciò potrebbe portare alla generazione di un albero descrittivo dei percorsi con decine di migliaia di foglie. Il risultato è la valutazione di sfumature di colore insignificanti per la determinazione del colore finale del pixel. Questo rappresenta un considerevole spreco di tempo computazionale. Il Path Tracing risolve questo problema randomizzando il lancio dei raggi secondari attraverso il metodo Monte Carlo, per questo motivo gli è stato affibbiato anche il nome di Monte Carlo Ray Tracing, da

cui deriva la confusione iniziale.

L'algoritmo utilizza il metodo Monte Carlo per scegliere casualmente un punto interno ad un pixel attraverso il quale tracciare il primo raggio. Se il raggio interseca una superficie, il metodo Monte Carlo viene nuovamente sfruttato per decidere in quale direzione, interna al dominio di integrazione, lanciare il raggio secondario. In caso di ulteriori intersezioni, il processo si ripete.

In scene che presentano condizioni di illuminazione particolarmente complesse, l'algoritmo di Path Tracing può produrre immagini caratterizzate da alta varianza. Un esempio di condizione critica è il seguente:

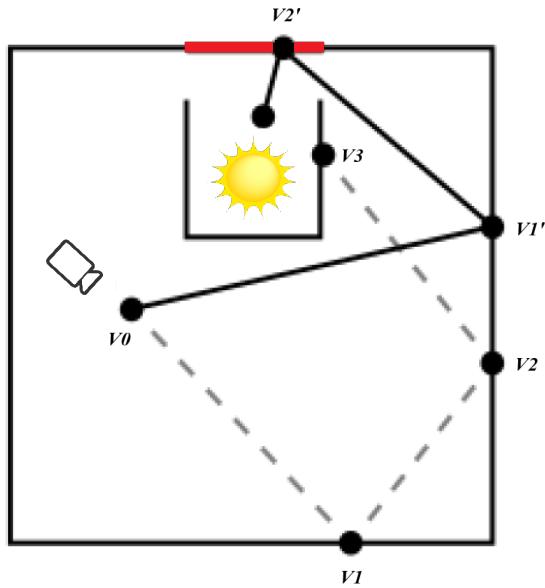


Figura 14: Condizione critica. Fonte: immagine autoprodotta.

Tracciando percorsi solo dal punto di vista dalla telecamera, difficilmente si cappionerà un vertice del percorso che è illuminato dalla sorgente. Gli shadow ray dei vertici V_0, V_1, V_2, V_3 e V_1' intersecano l'oggetto che ostruisce la sorgente. La maggior parte dei percorsi non avrà contributo. Allo stesso tempo, i pochi che colpiscono per caso la piccola regione sul soffitto, avranno un contributo significativo. Per risolvere le criticità, Eric Lafourte e Yves Willem, nel 1993, in [LW93] hanno descritto il Bidirectional Path Tracing. L'idea alla base della soluzione è quella di costruire percorsi che partano sia dall'osservatore che dalla sorgente d'illuminazione.

zione. Tutti i punti di intersezioni lungo i due percorsi vengono successivamente collegati con degli Shadow Ray e i relativi contributi vengono considerati per il calcolo del colore da attribuire al pixel. In questo modo, oltre al contributo della sorgente vengono considerati in modo probabilistico anche i contributi secondari, terziari, etc prodotti dalla stessa.

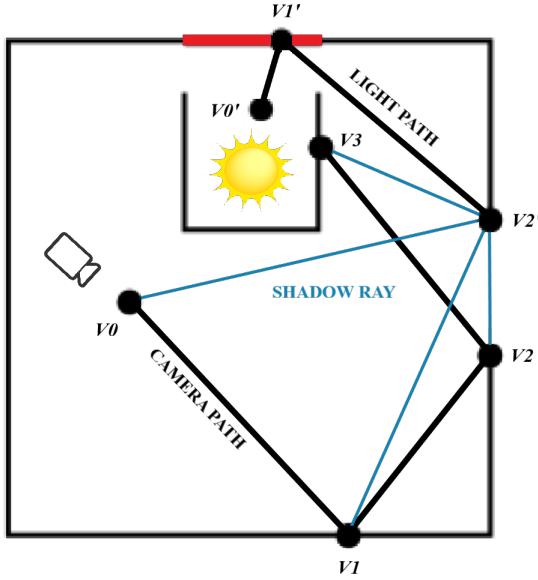


Figura 15: Bidirectional Path Tracing. Fonte: immagine autoprodotta.

Nella figura riportata non sono stati tracciati gli Shadow Ray dei vertici $V0'$ e $V1'$ in quanto occlusi.

Tre anni dopo l'introduzione del Bidirectional Path Tracing e solo un anno dopo la pubblicazione dell'articolo di Kaufman e Avila sul Volumetric Ray Tracing, Eric Lafortune e Yves Willems, in [LW96] attraverso il Bidirectional Path Tracing si concentrano sulla resa dei cosiddetti "Participating Media". Quest'ultimi si riferiscono a materiali che non solo trasmettono o riflettono la luce, ma possono anche assorbirla e disperderla sia all'interno che all'esterno, interagendo con la luce in modi non lineari o complessi. Alcuni esempi includono fumo, nebbia, vapore e altri fluidi. Benché l'articolo non parli in modo diretto di voxel l'idea alla base dell'algoritmo è applicabile ad essi.

Un raggio che attraversa un Participating Media ha una certa probabilità di essere

disperso. Questa probabilità è definita in accordo alla legge di Beer-Lambert (8). Campionando la funzione è possibile trovare la distanza che il raggio percorre dentro il volume prima di venire disperso. Una volta identificato il punto si utilizza il metodo Monte Carlo per definire la direzione di dispersione.

Questo metodo è computazionalmente costoso ma più versatile rispetto alle soluzioni precedentemente descritte, inoltre, gestisce correttamente lo scattering multiplo in mezzi non omogenei e anisotropo in situazioni di illuminazione complesse.

Appendice B Texture-Based Volume Rendering

Il Texture-Based Volume Rendering è la prima tecnica object-order che verrà approfondita. Questo significa che tutti i calcoli vengono svolti in coordinate nello spazio dell'oggetto e il rendering viene completato proiettando ogni elemento sul piano immagine e svolgendo un certo numero di operazioni di compositing.

L'algoritmo di rendering basato sulle texture si compone di 3 macro fasi: Inizializzazione, Aggiornamento dei dati e Disegno. La fase di Inizializzazione viene eseguita una sola volta, le due fasi successive vengono eseguite una prima volta per generare l'immagine e successivamente vengono rieseguite ogni qual volta l'utente modifica le impostazioni di rendering (come, ad esempio, i settaggi della camera o la funzione di trasferimento). La tecnica sfrutta direttamente alcune funzionalità della pipeline di rendering implementate sull'hardware grafico, come il Texture Mapping e il Blending.

Entrando nel dettaglio, le fasi sopracitate svolgono le seguenti operazioni:

- Inizializzazione: i dati vengono caricati dalla memoria e su di essi viene applicato lo Slicing: una trasformazione che taglia in sezioni sottili il volume lungo ogni direzione principale. Il risultato è il seguente:

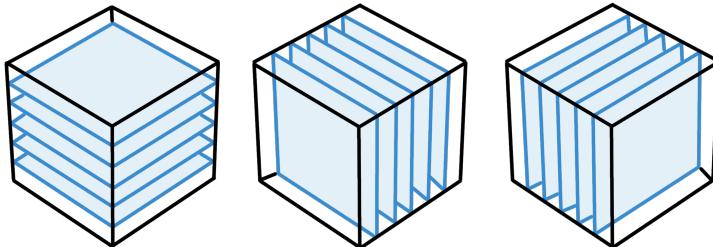


Figura 16: Slicing. Fonte: immagine autoprodotta.

I valori che cadono tra due sezioni vengono calcolati tramite interpolazione bilineare.

Creando 3 stack, uno per ogni direzione, si garantisce una corretta visualizzazione del volume da ogni possibile angolo con una sola esecuzione dello step.

Per ogni sezione, i dati volumetrici vengono approssimati attraverso primitive geometriche, in genere triangoli. Il risultato è un insieme di geometrie

cosiddette "proxy", ovvero approssimative. Il nome deriva dal fatto che l'hardware grafico non dispone di primitive volumetriche.

Infine, utilizzando le geometrie appena computate, si crea una texture map basata sui pixel per ogni sezione.

- Aggiornamento dei dati: questa fase prende in input i parametri della camera e la funzione di trasferimento. Con i primi si decide quale stack di geometrie proxy utilizzare per la visualizzazione. Con la funzione di trasferimento si generano le texture da applicare sulle sezioni. Quest'ultimo processo è totalmente svolto da funzionalità hardware.

L'utilità di aver pre-calcolato 3 stack di geometrie proxy diventa ora chiara: se l'angolo di visualizzazione supera i 45 gradi rispetto alla normale delle sezioni dello stack in visualizzazione è sufficiente scambiare lo stack in visualizzazione e aggiornare le texture. Le geometrie proxy e le texture map non andranno ricalcolate.

- Disegno: questo step si occupa di settare lo stato di rendering nella pipeline e di disegnare, infine, il volume sul piano di visualizzazione. Con "settare lo stato di rendering nella pipeline" si intende:

- disabilitare gli step di lighting e culling;
- abilitare l'alpha blending e depth testing;
- disabilitare la scrittura sul depth buffer.

Il metodo offre diversi vantaggi, ma presenta anche alcune limitazioni significative. Uno dei vantaggi principali del metodo è la sua velocità, che deriva dall'uso delle funzionalità hardware implementate direttamente sulla GPU. Un ulteriore vantaggio è la sua compatibilità con hardware vecchi che non sono in grado di supportare metodi più recenti. Basarsi totalmente su funzionalità hardware può anche rivelarsi uno svantaggio: il metodo utilizza poche ma grandi primitive, perciò, il numero di frammenti generato dal rasterizzatore per ognuna è elevato. La situazione è aggravata dal fatto che per produrre un rendering di qualità si debbano generare innumerevoli sezioni. Inoltre, non è semplice implementare ottimizzazioni.

I tre stack, essendo allineati alle facce del volume, causano la creazione di artefatti visivi se si osserva il volume ad un angolo di 45 gradi rispetto alla normale

ad una faccia. Quest'ultima problematica può essere risolta attraverso la variante descritta nel paragrafo successivo.

B.1 Variante con utilizzo di Texture 3D

In questa variante il volume viene suddiviso in fette perpendicolari alla direzione di vista. Questa alternativa offre un miglioramento grafico significativo, poiché non si osservano artefatti quando si guarda il volume con un'angolazione di 45 gradi rispetto alla normale ad una faccia. Tuttavia, è computazionalmente molto più costosa:

- Lo stack di geometrie proxy è solamente uno, quindi, in questa variante, gli step di Slicing del volume e di generazione delle geometrie proxy vengono ripetuti ogni volta che l'utente modifica i parametri della camera o la funzione di trasferimento.
- La variante utilizza l'interpolazione trilineare per calcolare i valori intermedi.

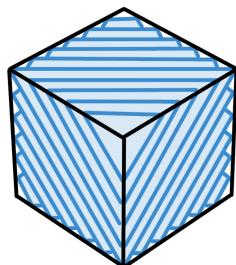


Figura 17: Slicing perpendicolare alla direzione di vista. Fonte: immagine autoprodotta.

Appendice C Splatting

Il nome scelto per il metodo vuole andare a descrive in maniera metaforica, ma tutt'altro che tecnica, il processo attraverso il quale lo Splatting renderizza dati volumetrici. La metafora usata dall'autore in [Wes91] è la seguente: si immagini il piano di visualizzazione come un muro di mattoni e i voxel come delle palle di neve. Il metodo "lancia" o, in linguaggio tecnico, proietta, le palle di neve sul muro di mattoni con lo scopo di lasciare un'impronta. Questa impronta si manifesta con un'intensità massima nel punto di impatto e gradualmente diminuisce con l'allontanarsi da esso. Il nome del metodo si ispira all'onomatopea che descrive il suono che produce una palla di neve che impatta contro il muro, ovvero, "splat". È inevitabile che le impronte delle palle di neve si sovrappongano sul muro, il colore di un mattone sarà quindi dato dalla combinazione delle palle di neve che si sovrappongono su di esso.

Nel paragrafo 1.1 è stato definito che un volume viene salvato nella memoria di un calcolatore come un insieme di valori discreti. Il volume stesso deve poi essere ricostruito e rappresentato attraverso un insieme di valori continui in fase di visualizzazione. La ricostruzione viene dunque svolta attraverso un processo di interpolazione (si veda paragrafo 1.1).

Lo Splatting non segue il convenzionale processo di ricostruzione: esso avviene attraverso la convoluzione del volume campionato con un kernel gaussiano. Siano:

- $\rho(i, j, k)$ la funzione che restituisce la densità del volume nel punto (i, j, k) , dove i, j e k sono definiti dagli intervalli di campionamento lungo le direzioni X, Y e Z ;
- $h(x, y, z) = e^{-(x^2/\sigma_x^2)-(y^2/\sigma_y^2)-(z^2/\sigma_z^2)}$ il kernel gaussiano usato per ricostruire il volume. Viene utilizzato un kernel gaussiano perché il metodo vuole rappresentare i voxel come "palle di neve", ovvero sferici e simmetrici. La forma è importantissima poiché qualsiasi altra potrebbe produrre artefatti guardando da diverse direzioni il volume. Per rendere il kernel isotropo deve valere $\sigma_x = \sigma_y = \sigma_z$.

Allora, la densità nel punto (x, y, z) , interno al volume, è data da:

$$\delta(x, y, z) = \sum_i \sum_j \sum_k \rho(i, j, k) h(x - i, y - j, z - k) \quad (13)$$

Partendo dalla formula appena indicata, l'autore, in [Wes91], dimostra i passaggi fondamentali per derivare l'equazione che descrive matematicamente l'impronta di un voxel sul piano immagine. Non è nell'interesse di questa tesi descrivere tutti i passaggi, perciò, si procederà riportando l'equazione appena menzionata ed in seguito verrà discusso come un voxel contribuisce all'immagine finale. L'equazione che descrive matematicamente l'impronta di un voxel sul piano immagine è:

$$F(x, y) = \int_{-\infty}^{\infty} h(x, y, w) dw \quad (14)$$

Dove:

- (x, y) è l'offset tra il centro di un pixel e il centro della proiezione del kernel sul piano immagine;
- w è la direzione del raggio che attraversa il kernel e colpisce il piano immagine perpendicolarmente.

Si evince che l'impronta è la proiezione sul piano immagine bidimensionale del kernel gaussiano di ricostruzione ed è indipendente rispetto ai valori dei campioni. Il contributo di un voxel v ad un pixel p di coordinate (x, y) è dato dall'equazione:

$$I_v(p) = I_v(x, y) = \rho(v) F(x - x_v, y - y_v)$$

Il colore finale di un pixel è ricavato sommando tutti i contributi delle impronte:

$$I(p) = \sum_v I_v(p) = \sum_v \rho(v) F(x - x_v, y - y_v) \quad (15)$$

Attraverso la matematica si sono descritte le idee alla base dell'algoritmo ed i calcoli che esso svolge. Con il seguente schema, invece, si mira a definire i passaggi chiave che lo Splatting esegue:

1. Il volume viene caricato dalla memoria e ricostruito attraverso la convoluzione con kernel gaussiano. Attraverso questo processo si ottiene una rappresentazione del volume nel continuo. Il volume viene tagliato in fette perpendicolari rispetto alla direzione di vista. Ogni fetta verrà processata singolarmente.

2. Si seleziona la prima fetta non ancora processata e ogni campione viene proiettato su un piano intermedio di supporto. Su quest'ultimo sarà necessario gestire le sovrapposizioni tra i campioni. Non verranno approfonditi i metodi per gestire le sovrapposizioni, per maggiori informazioni si faccia riferimento a [Wes91].
3. Si esegue uno step di compositing tra piano intermedio e piano immagine.
4. Si ripete il processo dallo step 2 per ogni fetta del volume.

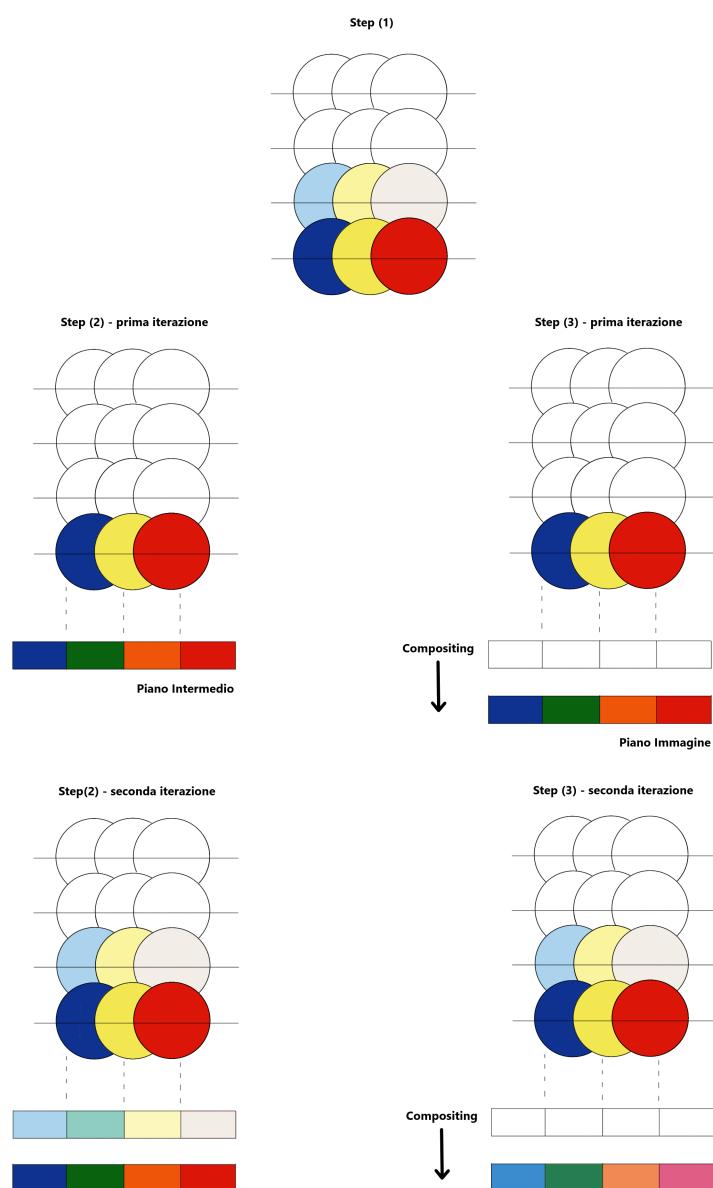


Figura 18: Splatting. Fonte: immagine autoprodotta.

Al fine di rendere più chiara e leggibile la rappresentazione dei passi dell'algoritmo si è optato per la saturazione dei colori nelle impronte. È importante ripetere quanto detto all'inizio del capitolo: un'impronta si manifesta con un'intensità massima nel punto di impatto e gradualmente diminuisce con l'allontanarsi da esso, questo effetto è dovuto alla natura del kernel di ricostruzione.

Un esempio di come dovrebbe essere rappresentata correttamente un'impronta è il seguente:

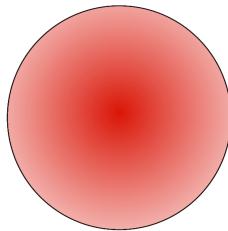


Figura 19: Impronta. Fonte: immagine autoprodotta.

Le fette possono essere processate back-to-front, se si vuole garantire un corretto overlay, oppure front-to-back, se si desidera ottimizzare e velocizzare il processo andando ad interrompere lo step di compositing per un determinato pixel appena esso diventa opaco.

Il metodo si rivelò particolarmente veloce: è stato dimostrato in [Wes91] che, sfruttando la proiezione ortografica e le proprietà di un volume campionato linearmente rispetto alle direzioni principali, la funzione $F(x, y)$ può essere computata una sola volta all'inizio del processo di rendering ed è corretta per ogni possibile punto da cui si osserva il volume.

Purtroppo, il metodo risente di due problematiche:

- I dettagli, se si osserva attentamente il volume attraverso viste ingrandite, sono sfocati. Questo è dovuto al fatto che il kernel gaussiano è un filtro passo basso, e quindi taglia le alte frequenze.
- Le ambiguità che si vengono a creare dalla sovrapposizione dei kernel richiedono complesse risoluzioni.

Appendice D Shear-Warp Factorization

Lo Shear-Warp Factorization [LL94] non è un algoritmo di rendering a sé stante, bensì è uno step di ottimizzazione applicabile ad un qualsiasi algoritmo di rendering.

L'obiettivo generale dello step è eliminare, o quantomeno diminuire, le costosissime e frequenti operazioni di interpolazione che si devono eseguire quando i raggi di visualizzazione campionano il volume ad un angolo che non è perpendicolare ad una faccia del volume. Per raggiungere lo scopo, il metodo esegue il processo di compositing con raggi di visualizzazione paralleli tra di loro e perpendicolari rispetto alle fette del volume.

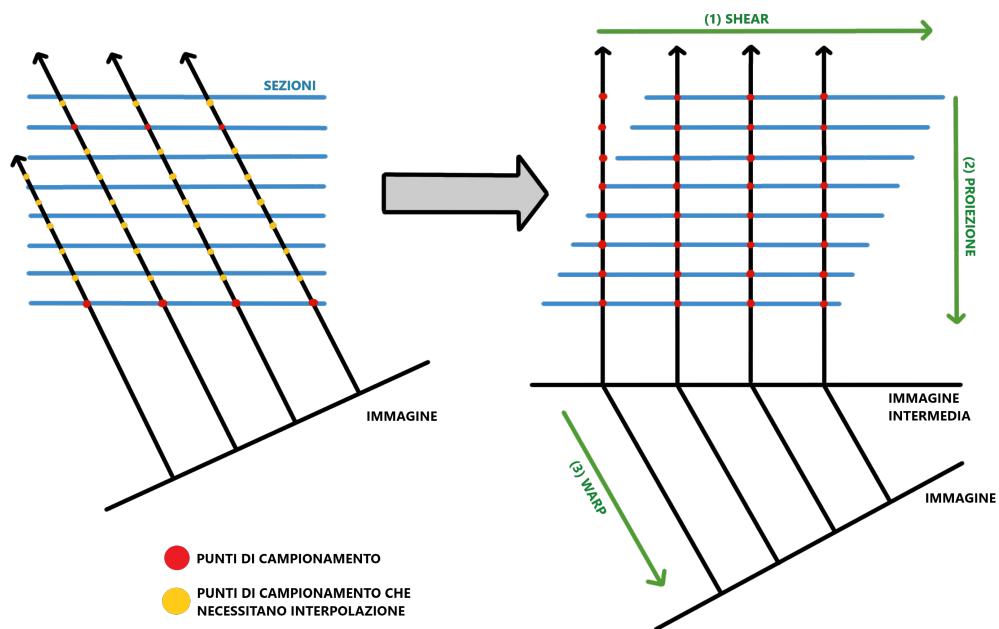


Figura 20: Shear-Warp Factorization. Fonte: immagine autoprodotta.

L'ottimizzazione, così come definita dal nome, si basa su una fattorizzazione. La fattorizzazione avviene sulla matrice di visualizzazione che rappresenta l'immagine finale e può essere descritta matematicamente come segue:

$$M_{view} = P \cdot S \cdot M_{warp} \quad (16)$$

dove:

- M_{view} è la matrice di visualizzazione (coordinate immagine);
- P è la matrice di permutazione, traspone il sistema per rendere l'asse z l'asse di visualizzazione principale. Si sceglie l'asse z per convenzione;
- S è la matrice di trasformazione, esegue lo shear sul volume.

$$- \text{ la matrice di shear per la proiezione parallela è: } S_{par} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s_x & s_y & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

dove s_x e s_y possono essere calcolati a partire dagli elementi di M_{view} .

$$- \text{ la matrice di shear per la proiezione prospettica è: } S_{persp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ s'_x & s'_y & 1 & s'_w \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

La matrice indica che, per trasformare una sezione di voxel, denominata v_0 , è necessario traslare la sezione di $(v_0 s'_x, v_0 s'_y)$ e successivamente scalarla uniformemente di $1/(1 + v_0 s'_w)$.

Benché questo tipo di proiezione sia stata storicamente utilizzata per risolvere le ambiguità legate alla profondità, non è l'opzione migliore in quanto risulta difficile campionare il volume in modo uniforme dato che i raggi divergono. Inoltre, le fette devono essere ridimensionate dopo lo shear:

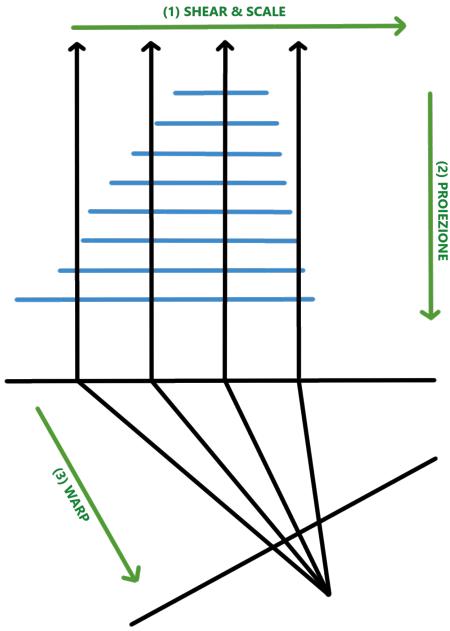


Figura 21: Shear-Warp Factorization: proiezione prospettica.

Fonte: immagine autoprodotta.

- M_{warp} è la matrice di trasformazione: coordinate oggetto → coordinate immagine. L'operazione di warp non è altro che un mapping tra pixel di due immagini diverse. Se l'immagine intermedia ha una risoluzione maggiore dell'immagine finale si eseguirà uno shrinking. Se l'immagine intermedia ha una risoluzione minore dell'immagine finale si eseguirà un'interpolazione bilineare.

D.1 Ottimizzazione dello Shear-Warp Factorization

La proprietà che caratterizza il metodo e da cui deriva la seguente ottimizzazione è: le scan line dei voxel nel volume sottoposto a trasformazione di taglio sono allineate alle scan line dei pixel nell'immagine intermedia. Ciò significa che le strutture dati che rappresentano le sezioni e l'immagine intermedia possono essere attraversate simultaneamente seguendo l'ordine delle scan line.

Utilizzando la Run-Length Encoding sulle scan line dei voxel si genera la prima struttura dati: un elenco che identifica regioni di voxel trasparenti.

La seconda struttura dati consiste in una matrice che indica per ogni pixel opaco dell'immagine intermedia l'offset al primo pixel non opaco che cade sulla stessa scan

line. Questa struttura verrà utilizzata in modo da ottenere un risultato identico a quello della Early Ray Termination.

Utilizzando la prima struttura dati per ignorare i voxel trasparenti e la seconda per ignorare i voxel occlusi, si eseguono i calcoli solo per i voxel non trasparenti e visibili. Il metodo richiede di eseguire l'operazione di compositing front-to-back.

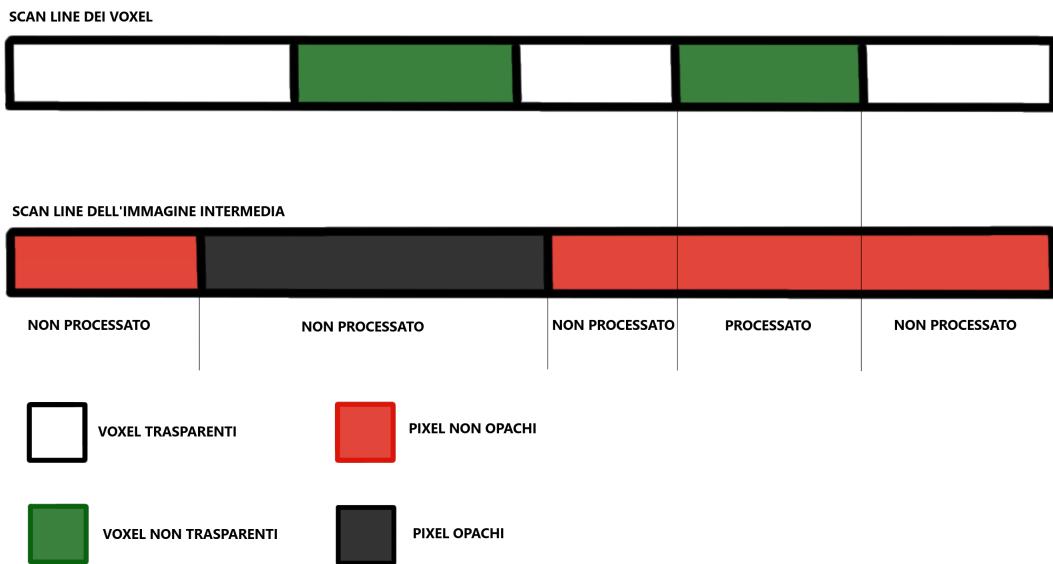


Figura 22: Shear-Warp Factorization: ottimizzazione con Run-Lenght Encoding.

Fonte: immagine autoprodotta.

L'ottimizzazione calcola 3 codifiche, una per ogni direzione principale del volume: questo evita la necessità di trasporre il volume per cambi di vista. Il vantaggio è rappresentato dal fatto che tutte le strutture dati sono calcolate con uno step di pre-elaborazione e sono indipendenti rispetto alla vista, ciò significa che il tempo speso per il calcolo può essere ammortizzato su più rendering.

Appendice E Marching Cubes

Il Marching Cubes è l'unico algoritmo approfondito che non fa parte della famiglia di algoritmi DVR e che, a differenza loro, estrae iso-superficie poligonali che seguono i contorni determinati da certi valori di soglia.

Se il volume non è rappresentato attraverso voxel, allora, la tecnica inizia suddividendo il volume in cubi piccoli e discreti. Diversamente, questo passaggio non è necessario. Questa suddivisione facilita l'analisi delle iso-superficie all'interno del volume.

Il metodo prende in input i valori della funzione F_{ijk} campionati nei punti (x_i, y_k, z_k) interni alla griglia di voxel. L'output dell'algoritmo consiste in una collezione di superfici, in ciascuna il valore di $F(x, y, z)$ è costante e il loro insieme descrive il contorno del volume. I vertici delle superfici giacciono sempre sul lato di un voxel. Il metodo considera un voxel alla volta e utilizzando i valori di F_{ijk} e l'interpolazione lineare è in grado di calcolare in che punto una superficie interseca un bordo. Calcolati tutti i punti di intersezione la superficie può essere disegnata all'interno del voxel. Per farlo, la versione originale dell'algoritmo si basa su una tabella con 15 configurazioni univoche. Sfruttando simmetria rotazionale, speculare e cambi di segno si riusciva a coprire quasi tutti i casi possibili.

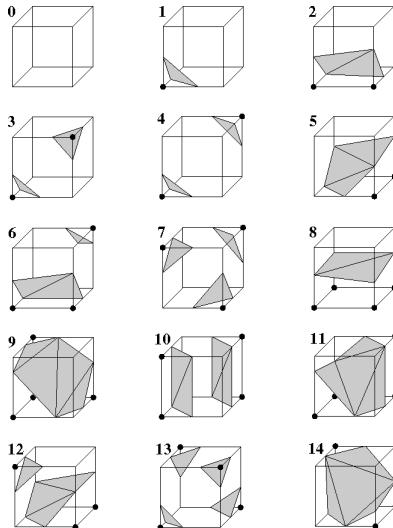


Figura 23: Marching cubes: configurazioni.

Fonte: Marching Cubes33: Construction of Topologically Correct.

Autori: IsosurfacesEvgeni V. ChernyaevInstitute

La configurazione da disegnare all'interno di un voxel viene selezionata basandosi su una denominazione dei vertici. Si immagini di scegliere la seguente:

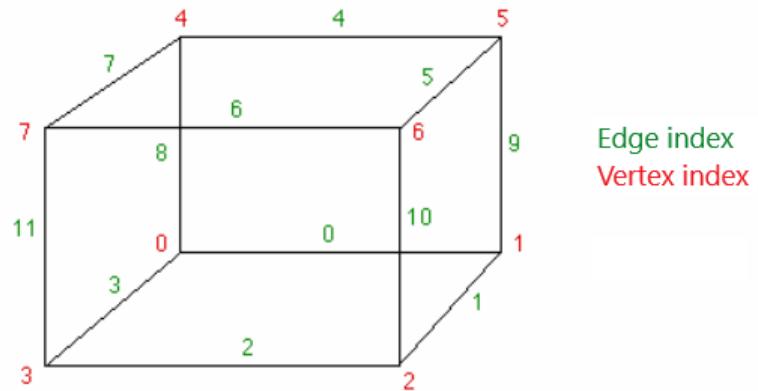


Figura 24: denominazione vertici. Fonte: immagine autoprodotta

Ad ogni possibile combinazione di marcatura dei vertici sotto illustrati:

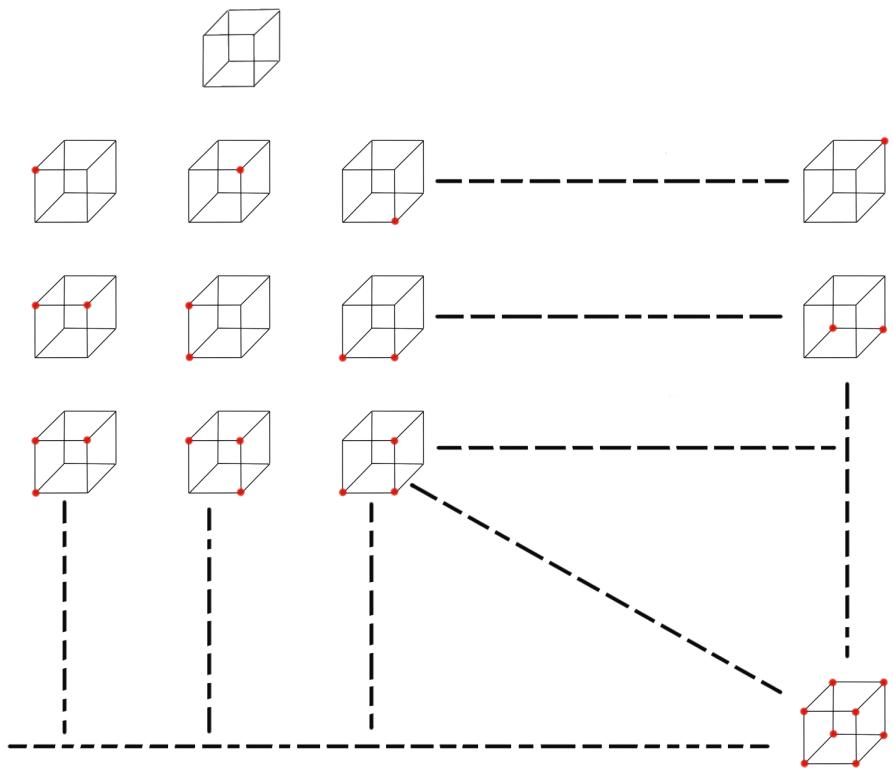


Figura 25: 256 marcature possibili. Fonte: immagine autoprodotta

si associa una configurazione univoca:

<i>index</i>	<i>e0</i>	<i>e1</i>	<i>e2</i>	<i>e3</i>	<i>e4</i>	<i>e5</i>	<i>e6</i>	<i>e7</i>	<i>configurazione</i>
00000000	0	0	0	0	0	0	0	0	conf1
00000001	0	0	0	0	0	0	0	1	conf2
00000010	0	0	0	0	0	0	1	0	conf2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
00000011	0	0	0	0	0	0	1	1	conf3
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
11111111	1	1	1	1	1	1	1	1	completamente renderizzato

EdgeTable =
Per ogni voxel si ricava il valore di F_{ijk} ai vertici e quest'ultimi si confrontano con l'iso-valore della superficie (α , da qui in avanti). Se $F_{ijk} > \alpha$ il vertice è esterno al volume e viene marcato con uno "0", se $F_{ijk} < \alpha$ il vertice è interno al volume e viene marcato con un "1". Facendo attenzione a seguire la stessa denominazione dei vertici precedentemente scelta, è facile ricavare un indice binario del voxel per capire quale configurazione disegnare al suo interno.

Nell'elaborazione delle superfici si poteva incorrere in casi ambigui dovuti all'interpolante. Il problema non risiedeva nella scelta della configurazione, quanto piuttosto nella scelta del profilo da disegnare. Le ambiguità sono state parzialmente eliminate attraverso lo sviluppo di nuovi algoritmi, lo sviluppo di test e la modifica dell'algoritmo originale. Particolare merito lo si deve a Chernyaev che portò la tabella delle configurazioni da 15 a 33:

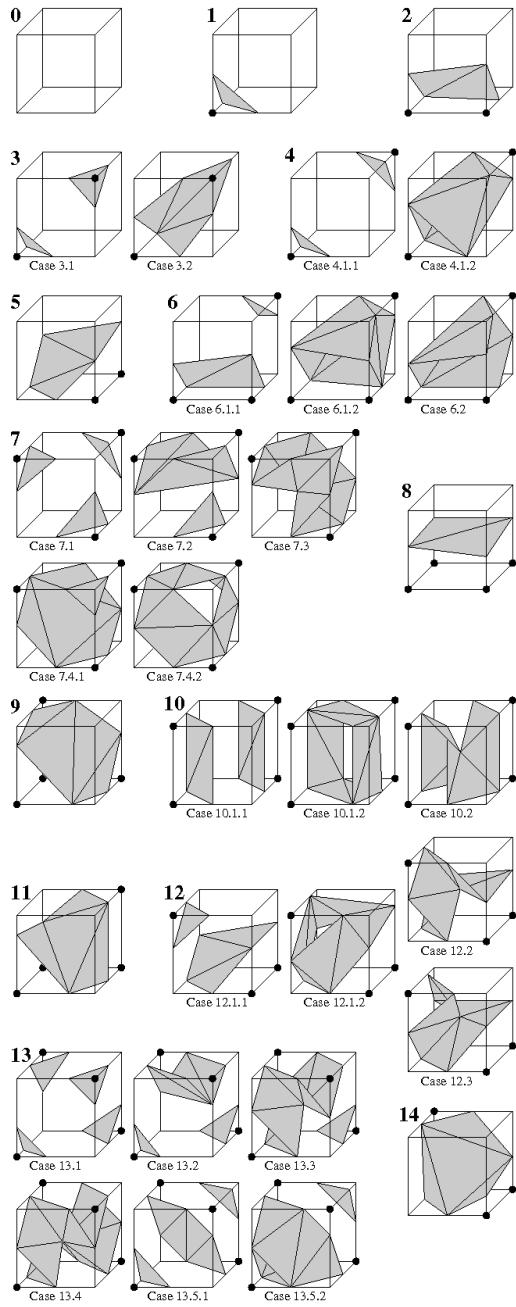


Figura 26: Marching cubes33: configurazioni.

Fonte: Marching Cubes33: Construction of Topologically Correct.

Autori: IsosurfacesEvgeni V. ChernyaevInstitute

Ulteriori anomalie sono state risolte grazie ad un test proposto da Nielson e Hamann nel 1991, l'Asymptotic Decider [NH91]. Molte altre problematiche topologiche sono state parzialmente risolte negli anni successivi grazie al lavoro di Custodio & al., del 2013.

E.1 Asymptotic Decider

Inizialmente è stato proposto come un miglioramento dell'algoritmo Marching Cubes, ma può essere considerato come un algoritmo autonomo.

Il metodo segue gli stessi passaggi dell'algoritmo originale con la differenza che si impegna a risolvere le ambiguità legate al collegamento degli iso-punti che cadono sui lati dei voxel.

Da qui in avanti, per semplicità di rappresentazione, si ragionerà su quadrati e non su cubi. L'equivalente dell'iso-superficie in 2D è l'iso-linea.

Supponendo che i vertici vengano marcati come positivi se il loro valore è maggiore di quello dell'iso-valore (esterni al volume), o negativi se è inferiore (interni al volume), si genera il caso ambiguo più semplice da rappresentare quando i vertici positivi sono separati da due iso-linee, o quando i vertici positivi si trovano nella sezione principale del quadrato e i vertici negativi sono separati da due iso-linee. Graficamente il caso è questo:

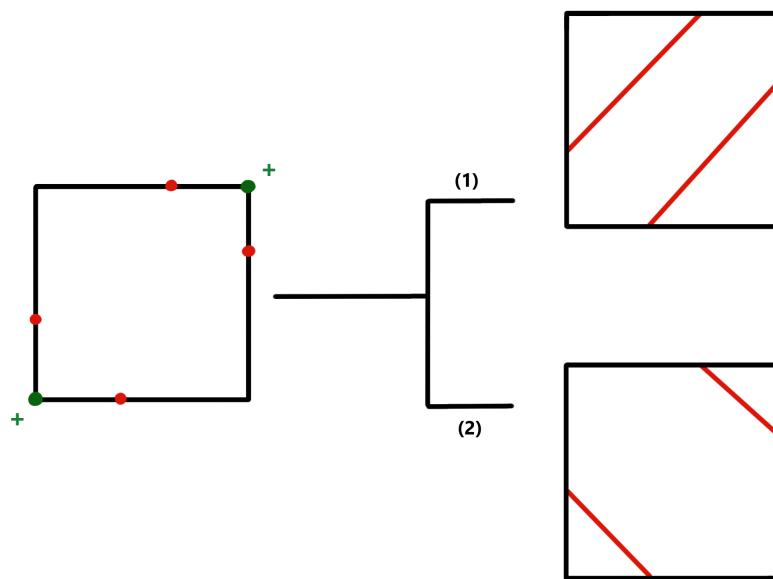


Figura 27: Marching Cubes: caso ambiguo. Fonte: immagine autoprodotta

Per risolvere l'ambiguità il metodo utilizza l'interpolazione bilineare. Dati:

- un quadrato di cui si conoscono i valori di F_{ijk} ai vertici : $F(x_0, y_0)$, $F(x_0, y_1)$, $F(x_1, y_0)$ e $F(x_1, y_1)$;
- un punto interno al quadrato di coordinate (s, t) ;

il valore $F(s, t)$ è dato dalla formula:

$$F(s, t) = (1 - s, s) \cdot \begin{bmatrix} F(x_0, y_0) & F(x_0, y_1) \\ F(x_1, y_0) & F(x_1, y_1) \end{bmatrix} \cdot \begin{bmatrix} 1 - t \\ t \end{bmatrix} \quad (17)$$

È stato dimostrato in [NH91] che la curva di livello di F , definita come $\{(s, t) : F(s, t) = \alpha\}$ è un'iperbole e, volendo dare una regola più formale di quella espressa per l'esempio precedente, un'ambiguità si crea quando entrambe le componenti iperboliche intersecano il voxel.

Confrontando il valore di α con il risultato dell'interpolazione bilineare nel punto di intersezione degli asintoti, le ambiguità possono essere risolte. Se $\alpha < F(S\alpha, T\alpha)$ il profilo corretto da disegnare nell'esempio precedente è il numero (1), se $\alpha > F(S\alpha, T\alpha)$ il profilo corretto da disegnare nell'esempio precedente è il numero (2).

Dove:

- $S\alpha = \frac{F(x_0, y_0) - F(x_0, y_1)}{F(x_0, y_0) + F(x_1, y_1) - F(x_0, y_1) - F(x_1, y_0)}$;
- $T\alpha = \frac{F(x_0, y_0) - F(x_1, y_0)}{F(x_0, y_0) + F(x_1, y_1) - F(x_0, y_1) - F(x_1, y_0)}$;
- $F(S\alpha, T\alpha) = \frac{F(x_0, y_0)F(x_1, y_0) + F(x_0, y_1)F(x_1, y_0)}{F(x_0, y_0) + F(x_1, y_1) - F(x_0, y_1) - F(x_1, y_0)}$

Appendice F Lavori Correlati

Nel capitolo vengono brevemente descritte le soluzioni esistenti nel campo del rendering di voxel. Come si è potuto evincere da questa tesi, le alternative sono numerose e per ciascuna è possibile trovare implementazioni open source in rete. Descriverle tutte non è nell'interesse di questo elaborato, ci si è quindi concentrati su quelle maggiormente degne di nota.

F.1 GVDB

GVDB, acronimo di GPU Voxel Databases [Hoe16], è una libreria open source sviluppata da NVIDIA per la gestione di dati volumetrici sparsi. Venne presentata per la prima volta alla GPU Technology Conference del 2017. Purtroppo, la libreria non viene aggiornata dal 2020, è tuttavia funzionante e disponibile. Di recente, progetti indipendenti hanno reso la libreria un plugin di Unreal Engine con l'obiettivo di renderizzare i fluidi.

Per effettuare il rendering la libreria si basa sul Ray-Tracing e su una particolare struttura dati che prende il nome dalla libreria stessa.

La struttura dati GVDB si ispira ai *B+* tree e consiste in un insieme di nodi a più livelli in una gerarchia a griglia. Ogni nodo contiene due informazioni: una bitmask e un elenco di puntatori ai nodi figli. L'elenco di puntatori considera solo i nodi attivi e i voxel vuoti vengono ignorati. La bitmask serve per individuare i figli attivi di un nodo data la sola posizione spaziale all'interno della griglia. Data una posizione spaziale, se il bit del figlio è attivo nella bitmask del nodo padre, il suo indice nell'elenco di puntatori ai nodi figli può essere rapidamente trovato utilizzando un conteggio a 64 bit sulla bitmask.

Comunemente, i sistemi basati su Ray Tracing utilizzano bounding box e algoritmi per il calcolo delle intersezioni per cercare il punto di intersezione tra un raggio ed un oggetto. In questo caso, si è deciso di utilizzare l'algoritmo DDA (Digital Differential Analyzer) per enumerare i voxel che un raggio attraversa.

F.2 Cinematic Rendering

Cinematic Rendering [CEGM16] è il termine che identifica una soluzione sviluppata da Siemens Healthineers per la visualizzazione tridimensionale di volumi medici.

Il progetto è stato presentato nel 2017 e continua ad essere attivamente sviluppato e supportato. Nel 2023 la soluzione è stata integrata su HoloLens2 consentendo agli utenti di interagire con i volumi in modo più interattivo. All'interno dell'ambiente di realtà aumentata è possibile ruotare, spostare, tagliare, ingrandire o rimpicciolire il volume con un semplice tocco.

Per ogni pixel, la soluzione utilizza il Path Tracing per simulare il trasporto della luce lungo i percorsi tracciati nella scena. Il calcolo dell'illuminazione si basa su un'approssimazione ottenuta mediante il metodo Monte Carlo dell'equazione di illuminazione globale denominata "Volume Rendering Equation" e illustrata nel paragrafo 2.2.2.

Invece delle classiche fonti di illuminazione, sono state utilizzate mappe HDR. Questa scelta permette di catturare la luce che arriva ai dati da qualsiasi direzione, portando ad una rappresentazione naturale del volume. Quest'ultima, in concomitanza alla scelta di simulare accuratamente la dispersione e l'assorbimento dei fotoni produce immagini fotorealistiche che presentano ombre morbide, occlusione ambientale, dispersione volumetrica e interazione fotonica sub-superficiale.

Tuttavia, benché la soluzione sia in grado di creare rappresentazioni fotorealistiche dei dettagli anatomici, non è ancora stata approvata per uso medico. Questo è dovuto alla natura dei dati volumetrici, che richiedono l'utilizzo di metodi di interpolazione durante il processo di ricostruzione del volume, comportando un certo grado di approssimazione. Una diagnosi non può basarsi su dati approssimati, da questo deriva la decisione di negare l'approvazione per uso medico. Al momento, Siemens Healthineers offre la soluzione per scopi accademici. L'utilizzo di ologrammi interattivi per l'insegnamento di procedure mediche rappresenta un'interessante evoluzione nell'ambito della formazione medica.

Riferimenti bibliografici

- [AK95] Lisa Avila and Arie Kaufman. Volumetric ray tracing. pages 11–18, 01 1995.
- [App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS ’68 (Spring)*, pages 37–45, New York, NY, USA, 1968. Association for Computing Machinery.
- [Bar] Tomas Bartipan. Volume Rendering (Raymarching) Plugin for Unreal Engine.
<https://github.com/tommybazar/TBRaymarcherPlugin?tab=readme-ov-file>.
- [Bli77] James F. Blinn. Models of light reflection for computer synthesized pictures. *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, 1977.
- [BMN96] Fatima Linda Boumghar, Serge Miguet, and Jean-Marc Nicod. Optimal subdivision and complexity of discrete surfaces in the dividing-cubes algorithm. 10 1996.
- [Bru16] Ryan Brucks. Creating a Volumetric Ray Marcher, November 2016.
<https://shaderbits.com/blog/creating-volumetric-ray-marcher>.
- [CEGM16] D Comaniciu, K Engel, B Georgescu, and T Mansi. Shaping the future through innovations: From medical imaging to precision medicine. *Medical Image Analysis*, 33:19–26, 2016.
- [DIC] DICOMLookup.
<http://dicomlookup.com/>.
- [FWKH17] Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. Production volume rendering: SIGGRAPH 2017 course. In *ACM SIGGRAPH 2017 Courses*, SIGGRAPH ’17, New York, NY, USA, 2017. Association for Computing Machinery.

- [Hoe16] Rama Hoetzlein. GVDB: raytracing sparse voxel database structures on the GPU. In *High Performance Graphics*, 2016.
- [ICG86] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 133–142, New York, NY, USA, 1986. Association for Computing Machinery.
- [IKLH04] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. Volume Rendering Techniques. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 39. Pearson Higher Education, 2004.
- [Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. Association for Computing Machinery.
- [Kau00] Arie E. Kaufman. Introduction to volume graphics. 2000.
- [Laf95] Eric P. Lafourte. *Mathematical Models and Monte Carlo Algorithms for Physically Based Rendering*. PhD thesis, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium, February 1995.
- [LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. Association for Computing Machinery.
- [LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, page 458, New York, NY, USA, 1994. Association for Computing Machinery.

- [LW93] Eric P. Lafourne and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, December 1993.
- [LW96] Eric P. Lafourne and Yves D. Willems. Rendering participating media with bidirectional path tracing. In Xavier Pueyo and Peter Schröder, editors, *Rendering Techniques '96*, pages 91–100, Vienna, 1996. Springer Vienna.
- [NH91] G.M. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceeding Visualization '91*, pages 83–91, 1991.
- [Nie08] Gregory M. Nielson. Dual marching tetrahedra: Contouring in the tetrahedral environment. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Fatih Porikli, Jörg Peters, James Klosowski, Laura Arns, Yu Ka Chun, Theresa-Marie Rhyne, and Laura Monroe, editors, *Advances in Visual Computing*, pages 183–194, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [ON94] Michael Oren and Shree K. Nayar. Generalization of lambert's reflectance model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 239–246, New York, NY, USA, 1994. Association for Computing Machinery.
- [Oss] OssoVR. The leading virtual reality surgical training and assessment platform.
<https://www.ossovr.com/>.
- [Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, jun 1975.
- [Pre] PrecisionOS. Vr medical education and training.
<https://www.precisionostech.com/vr-training/>.
- [RES] ANIMA RES. Insight heart - the human heart expedition.
<https://animares.com/portfolio/insight-heart>.

- [Res16] Goldman Sachs Global Investment Research. Virtual & Augmented Reality. *Profiles in Innovation*, pages 28–30, 2016.
- [Scr09a] Scratchapixel. Ray marching: Getting it right!, Since 2009.
<https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/ray-marching-get-it-right.html>.
- [Scr09b] Scratchapixel. raymarch-chap6.cpp, Since 2009.
<https://github.com/scratchapixel/scratchapixel-code/blob/main/volume-rendering-for-developers/raymarch-chap6.cpp>.
- [Scr09c] Scratchapixel. Volume rendering based on 3d voxel grids, Since 2009.
<https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/volume-rendering-voxel-grids.html>.
- [Scr09d] Scratchapixel. Volume rendering of a 3d density field, Since 2009.
<https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/volume-rendering-3D-density-field.html>.
- [Sol] SmartTek Solutions. Tavr surgery.
<https://smarttek.solutions/portfolio/tavr-surgery/>.
- [SR15] Erik Sundén and Timo Ropinski. Efficient volume illumination with multiple light sources through selective light updates. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 231–238, 2015.
- [Was] Jakob Wassermann. Totalsegmentator.
<https://github.com/wassermann/TotalSegmentator>.
- [Wes91] L.A. Westover. *Splatting: A Parallel, Feed-forward Volume Rendering Algorithm*. University of North Carolina at Chapel Hill, 1991.
- [Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, jun 1980.