

Lighting and Shading: il modello del Ray Tracing

Antonio Sirignano

Calcolo Scientifico per l'Innovazione Tecnologica - prof. Luisa D'Amore

Sommario—In questo progetto viene fatta un'introduzione generale al lighting e allo shading, con una presentazione ai modelli di riflessione e di shading che possono essere applicati alla grafica computazionale. Viene descritto nel dettaglio il modello del Ray Tracing con la presentazione e la codifica in Python di un algoritmo intento a simularlo.

Keywords—calcolo scientifico, lighting, shading, ray tracing.

Indice

1	Introduzione	1
2	Le sorgenti di luce	1
3	Modelli di riflessione	2
4	Modelli di shading	2
5	Il modello del Ray Tracing	3
6	L'equazione di rendering	3
7	Scanline rendering	3
7.1	Z-buffering	3
7.2	Algoritmo del pittore	4
8	Ray Tracing	4
9	Path Tracing	5
9.1	Differenze tra Path Tracing e Ray Tracing	5
9.2	Rumore nel Ray Tracing	5
10	Applicazione	5
10.1	Prerequisiti	5
11	Algoritmo	5
11.1	Configurazione della scena	6
11.2	Intersezione dei raggi	6
11.3	Definizione della sfera	6
11.4	L'intersezione con la sfera	7
11.5	Intersezione con l'oggetto più vicino	7
11.6	Punti d'intersezione	7
11.7	Intersezione della luce	8
11.8	Modello di riflessione: Blinn-Phong	8
11.9	Primo risultato	9
11.10	Finto piano	9
11.11	Riflessione	9
11.12	Generazione di un video	10
Riferimenti		11

1. Introduzione

La Computer Graphic Technology è l'abilità di produrre un effetto visivo realistico in un oggetto tridimensionale in un device di output bidimensionale, come un computer o un foglio stampato. Tutto ciò si ha grazie ai *metodi di rendering* nei quali è applicato lo *shading* per raggiungere il più possibile una rappresentazione di un oggetto vicina alla realtà.

Infatti lo shading computa quantità e colore della luce emessa da ogni punto della superficie.

Tali risultati dipendono dalle seguenti entità:

1. **La sorgente di luce.** Intensità, colore, forma, direzione e distanza della sorgente di luce devono essere prese in considerazione e inoltre possono essere sia puntiformi che di grandi dimensioni.

2. **La superficie dell'oggetto.** L'oggetto può essere lucido, liscio, ruvido, brillante o scuro. Può inoltre avere colori differenti quali opachi, trasparenti o traslucidi.
3. **L'ambiente.** Oggetti visti in uno spazio vuoto, senza un background che rifletta la luce su di essi, risultano duri (una navicella spaziale nello spazio profondo). Un modello realistico di shading deve tenere in considerazione la luce riflessa dagli altri oggetti (pareti vicine).

2. Le sorgenti di luce

1

2

3

3

3

3

4

5

5

5

5

6

6

6

7

7

8

8

9

9

9

9

10

11

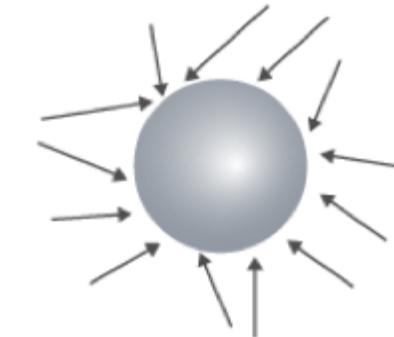


Figura 1. Luce ambientale

2. **Luce puntiforme.** È una sorgente di luce che non emette la stessa quantità di luce proveniente da tutte le direzioni, infatti un oggetto quanto è più vicino ad essa tanto è più luminoso. L'intensità della sorgente è quindi dipendente dalla distanza e dalla angolazione. È caratterizzata da colore, intensità, posizione e funzione di decadimento.

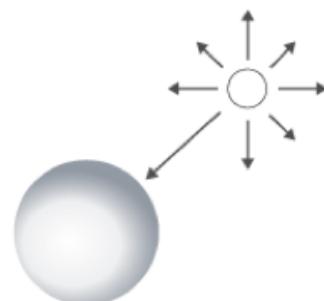


Figura 2. Luce puntiforme

3. **Luce direzionale.** Tale tipo è prodotta da una sorgente di luce da una distanza infinita dalla scena. Tutti i raggi di luce si espandono in una singola direzione e con la stessa intensità ovunque. È caratterizzata da colore, intensità e direzione.

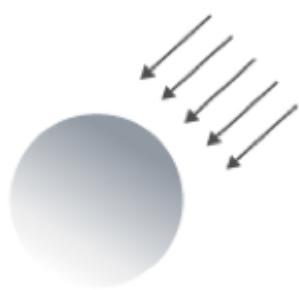


Figura 3. Luce direzionale

4. **Spotlight.** La luce si irradia in un cono con più luce al centro di esso. Questa luce è fissata all'asse primario di direzione con una restrizione su di essa. È caratterizzata come un punto di propagazione, un asse di direzione, un raggio intorno all'asse e la possibilità di una funzione di decadimento radiale.

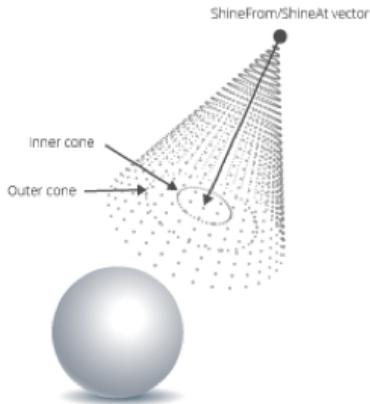


Figura 4. Spotlight

3. Modelli di riflessione

L'obiettivo principale dello shading è la produzione di un risultato accettabile quando la superficie dell'oggetto è affetta dai raggi di luce. I modelli di riflessione sono presentati di seguito.

1. **Riflessione diffusa.** Tale modello di riflessione diffonde la luce uniformemente in tutte le direzioni. In accordo con la legge di Lambert si ha che la diffusione del riflesso è proporzionale al coseno dell'angolo θ compreso tra la normale N e la direzione della sorgente L .

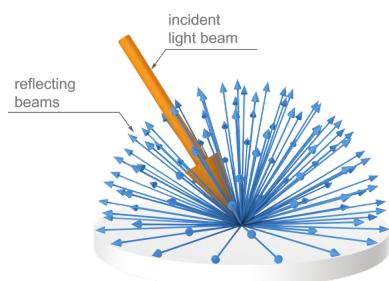


Figura 5. Riflessione diffusa

Da qui

$$\cos \theta = L \cdot N$$

E considerando il coefficiente di riflessione k_d , con $0 \leq k_d \leq 1$, si ha

$$R_d = k_d \cdot L \cdot N$$

Volendo considerare l'attenuazione della luce, si deve far riferimento alla distanze che essa percorre d . Quindi il termine di attenuazione quadratico è

$$R_d = \frac{k_d}{A + BD + C(D \cdot D)} (L \cdot N)$$

E quindi

$$I_d = \frac{k_d}{A + AD + A(D \cdot D)} (L \cdot N)$$

2. **Riflessione speculare.** Tale modello produce una riflessione luminosa sulla superficie dell'oggetto.

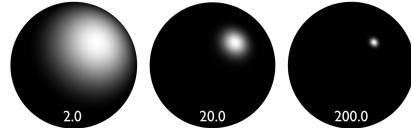


Figura 6. Riflessione speculare

L'angolo ϕ compreso tra la direzione della riflessione r e la direzione v dell'osservatore è affetto da una quantità di luce speculare. Il modello Phong stabilisce

$$R_s = k_s \cos^n \phi$$

per coefficiente k_s , con $0 \leq k_s \leq 1$. L'esponente n è il coefficiente di lucentezza. All'incrementare di n , la luce riflessa si concentra in una regione superficiale più piccola, centrata in r . Valori tra il 100 e il 500 corrispondono a superfici metalliche, valori più piccoli corrispondono a ad illuminazioni più ampie. Quindi

$$R_s = k_s (r \cdot v)^n$$

Una distanza viene moltiplicata similmente alle riflessioni diffuse.

La combinazione di tre tipi di riflessione danno

$$I = \frac{1}{A + BD + C(D \cdot D)} (K_d (L \cdot N) L_d + k_s (r \cdot v) n L_s) + K_a L_a$$

La direzione della riflessione r è semplice da n e I , assumendo che n sia un'unità di lunghezza, inoltre si vede che

$$\frac{L + r}{2} = (L \cdot n)n \implies r = 2(L \cdot n)n - L$$

4. Modelli di shading

I modelli di shading sono utilizzati per ottenere il modello di illuminazione desiderato. Modelli di shading efficienti per la superficie definita da un poligono si possono descrivere come seguono.

1. **Constant shading.** Questo modello è il più semplice modello di shading anche conosciuto come faceted shading o flat shading. Lo stesso colore è applicato su ogni intero poligono con un rendering veloce. L'equità della luce è usata una sola volta per poligono. Data una singola normale al piano, l'equazione della luce e le proprietà del materiale sono usate per generare un singolo colore. Il poligono avrà tale colore.
2. **Gouraud shading.** Questo modello è anche chiamato intensity interpolation shading o color interpolation shading. I colori sono interpolati attraverso il poligono e vi è la necessità di identificare ogni vertice. Il processo di rendering è più lento rispetto al modello flat. L'equità della luce è applicata ad ogni vertice e anche ogni colore è determinato dalla quantità di luce con le proprietà del materiale. La Scan-line interpolation è usata per assegnare un colore ad ogni punto di proiezione del poligono come la media pesata dei colori di dati vertici.

3. **Phong shading.** Tale modello è più realistico degli altri perché il suo algoritmo considera l'unione delle normali dei vertici ad ogni punto del poligono per avere una normale locale. Inoltre, il calcolo è applicato per avere un'illuminazione totale nel rendering.

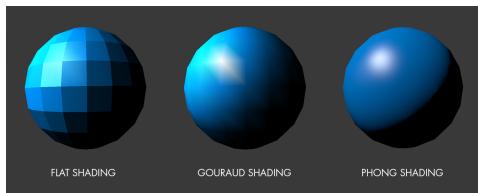


Figura 7. Flat, Gouraud e Phong shading

5. Il modello del Ray Tracing

L'ambizione degli artisti, dei designers, degli ingegneri è sempre stata la simulazione dell'immagine di una scena, una storia, un prodotto, un palazzo prima che siano effettivamente realizzati e costruiti - architetti devono costruire modelli in scala, produttori di film e video-game devono utilizzare storyboards per provare e trasmettere le sensazioni di un film o un video-game. E gli inserzionisti vogliono creare interpretazioni perfette dei loro prodotti nella migliore luce possibile. Inoltre i designer di prodotti e macchinari vogliono trovare le debolezze di un progetto prima che esso venga effettivamente costruito - direttori di film vogliono vedere il risultato finale prima dell'arduo lavoro di post-produzione, i produttori vogliono mostrare a potenziali clienti il prodotto finale per stimolarne la domanda. Il test di tali idee è chiamato virtual prototyping nella manifattura e pre-visualization nell'industria cinematografica. Vi è quindi la necessità di disporre di queste immagini, o video, a basso costo e in poco tempo.

6. L'equazione di rendering

Nella computer graphics, l'equazione di rendering è un'equazione integrale nella quale la radianza di equilibrio che lascia un punto è la somma della radianza emessa più la radianza riflessa sotto un'approssimazione dell'ottica geometrica.

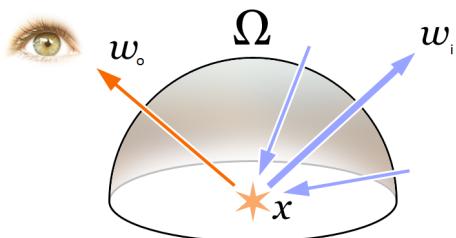


Figura 8. Equazione di rendering

Esistono svariate tecniche di rendering nella computer graphics che cercano di risolvere tale equazione. L'equazione di rendering descrive la quantità totale di luce emessa da un punto x lungo una particolare direzione di osservazione, data una funzione per la luce in entrata e una funzione di distribuzione bidirezionale di riflettenza (BRDF), dove,

- x è una posizione nello spazio
- w_o è la direzione della luce in uscita
- Ω è l'emisfero unitario centrato intorno $\{n\}$ contenente tutti i valori possibili di w_i
- w_i è il fattore di indebolimento dell'irraggiamento verso l'esterno a causa dell'angolo di incidenza, poiché il flusso luminoso si diffonde su una superficie la cui area è più grande dell'area proiettata perpendicolare al raggio

Risolvere l'equazione di rendering per ogni scena è la prima sfida per ottenere un rendering realistico.

Uno degli approcci è basato sui metodi degli elementi finiti, che ha portato all'algoritmo di radiosità. Un altro approccio utilizzando il metodo Monte Carlo ha portato a diversi algoritmi inclusi Path Tracing, photon mapping, il Metropolis light transport e molti altri. Monte Carlo ray tracing richiede come input la descrizione di una scena altamente dettagliata e basata sulla fisica delle cose. L'algoritmo applica le leggi fisiche per simulare la propagazione della luce attraverso la scena, piuttosto che approssimazioni ad hoc per i fenomeni visivi. Questo tipo di simulazione richiede modelli geometrici estremamente dettagliati. Inoltre, rendering fotorealistici richiedono che le proprietà fisiche del materiale superficiale siano modellate correttamente, in modo da descrivere come la luce si disperde quando la colpisce.

7. Scanline rendering

Lo Scanline rendering è un algoritmo per la determinazione della superficie visibile che lavora riga per riga rispetto a poligoni per poligono o pixel per pixel.

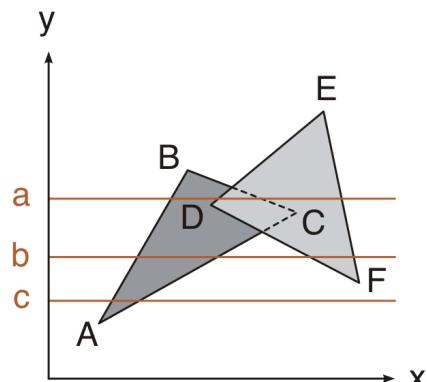


Figura 9. Algoritmo Scanline

Tutti i poligoni da renderizzare sono ordinati dall'alto sulla base delle ordinate in ordine di apparizione. Poi, ogni riga o scanline dell'immagine è calcolata utilizzando l'intersezione di una scanline con i poligoni nella parte anteriore della lista ordinata, mentre quest'ultima viene aggiornata scartando i poligoni non più visibili mano a mano che la scanline attiva avanza verso il basso dell'immagine. Il vantaggio principale di questo metodo è che l'ordine dei vertici lungo la normale del piano di scansione riduce il numero di comparazioni tra i bordi. Un ulteriore vantaggio rispetto allo z-buffering è che il processo dei pixel visibili è mantenuto al minimo assoluto. Attraverso l'ordinamento front-to-back utilizzato nei moderni sistemi z-buffer, si possono realizzare simili benefici.

7.1. Z-buffering

Lo Z-buffering, anche noto come depth buffering, è la gestione delle coordinate di profondità delle immagini nella 3D graphics, fatta di solito a livello hardware. Quando si proietta un oggetto sullo schermo con un motore di 3D-rendering, la profondità (z-value) del pixel generato nell'immagine proiettata viene memorizzata in un buffer (z-buffer). Uno z-value corrisponde alla misura della distanza perpendicolare da un pixel sul piano di proiezione alla sua corrispondente coordinata 3D su un poligono nello spazio. Lo Z-buffer ha la stessa struttura di un'immagine, ovvero un 2D-array, con la sola differenza che memorizza uno z-value per ogni pixel dello schermo invece che dei pixel dei dati. Ha le stesse dimensioni di uno screen buffer, eccetto quando sono utilizzati Z-buffer multipli.

Quando vediamo un'immagine contenente oggetti o superfici parzialmente o totalmente opachi, non è possibile vedere completamente questi oggetti sia che siano lontani dall'osservatore sia che siano dietro altri oggetti. L'identificazione e la rimozione di tali oggetti è chiamato problema delle superfici nascoste (hidden surface problem). Per migliorare il tempo di rendering, le superfici nascoste vengono rimosse prima che l'immagine sia memorizzata nel z-buffer, in modo che quest'ultimo calcoli lo z-value del pixel corrispondente al primo oggetto che compare e lo compara con lo z-value del pixel nella stessa posizione nello z-buffer corrispondente all'oggetto che è più vicino all'osservatore per trovare la sovrapposizione.

7.2. Algoritmo del pittore

L'algoritmo del pittore è una delle soluzioni più semplici al problema della visibilità nella 3D computer graphics. Quando si prietta una scena 3D in un piano 2D, è necessario decidere quale tra i poligoni è visibile.

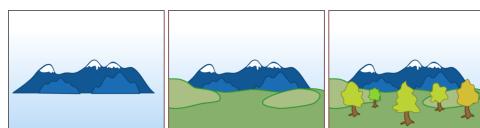


Figura 10. Algoritmo del pittore

L'algoritmo del pittore ordina tutti i poligoni nella scena partendo dalla profondità e poi li disegna in questo ordine, dal più lontano al più vicino. Si disegna sulle parti non visibili, al costo di avere oggetti già disegnati nascosti. L'ordine usato dall'algoritmo è chiamato depth order: se un oggetto oscura la parte di un altro, allora il primo oggetto disegnato dopo il secondo viene oscurato.

8. Ray Tracing

Negli ultimi anni, la qualità delle immagini generate computazionalmente hanno raggiunto un livello di realismo tale che i rendering sono indistinti dalle fotografie.

Nella computer graphics, il ray tracing è una tecnica di rendering per generare un'immagine tracciando il percorso della luce come pixel in un piano e simulando gli effetti dati dall'intersezione di tale percorso con gli oggetti virtuali. La tecnica può produrre un grado di realismo visivo molto alto, anche più alto a volte dei tipici metodi di rendering scanline ma con un alto costo computazionale.

Avanzati effetti di shading possono rendere la visualizzazione più efficace. Il vantaggio, l'opportunità e l'obiettivo del ray tracing è quello di fornire ulteriori segnali visivi per una migliore visualizzazione degli oggetti 3D.

Ci sono almeno quattro diversi raggi coinvolti nel ray tracing:

- **Raggi oculari** che hanno origine dagli occhi.
- **Raggi d'ombra**: dal punto di superficie alla sorgente di luce.
- **Raggi di riflessione**: dal punto di superficie in direzione speculare.
- **Raggi di trasmissione**: dal punto di superficie nella direzione di rifrazione.

L'algoritmo di ray tracing calcola il raggio dell'occhio dell'osservatore attraverso ogni pixel, calcola il punto di intersezione più vicino ad una superficie di scena, quindi ombreggia quel punto calcolando i raggi d'ombra e genera i raggi riflessi e rifratti.

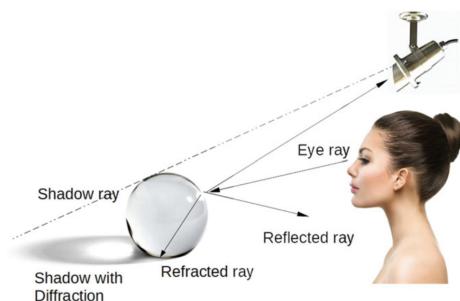


Figura 11. Raggi nel ray tracing

Il ray tracing ricorsivo simula la riflessione speculare e le ombre attraverso e fuori le superfici trasparenti. Può usare o impiegare l'illuminazione indiretta, a volte aree di sorgenti luminose o altre influenze caustiche.

Crea riflessi accurati, rifrazioni, ombre e altre caratteristiche che possano far sembrare la scena reale.

Il ray tracing in grandi linee può essere diviso in tre generali categorie:

1. Off-line
2. Interactive
3. Real-time

Off-line è usato ampiamente dagli studi cinematografici, pubblicitari e di design. Ha la qualità maggiore.

Interactive ray tracing riduce il numero di raggi in modo da avere comunque una buona immagine e allo stesso tempo di offrire all'utente la possibilità di manipolare il modello.

Real-time ray tracing può essere realizzato, con alcune restrizioni, dall'assistenza di piccoli supercomputer.

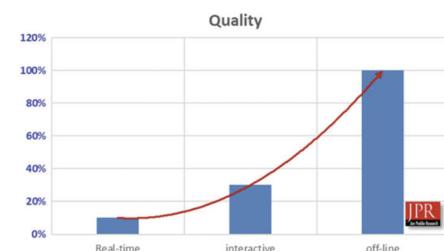


Figura 12. Performance e qualità nei vari modi di ray tracing

Uno dei trucchi del ray tracing è quello di tracciare solo determinati elementi o oggetti all'interno di un'immagine ma, se fatto in modo giusto, si ha un'immagine fisicamente corretta.

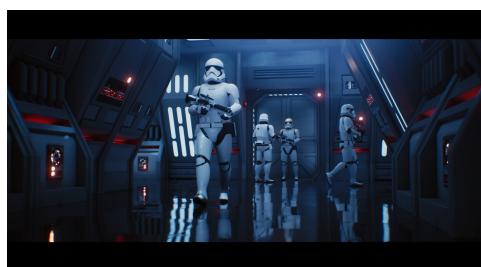


Figura 13. Stormtroopers (Star Wars) renderizzati con real-time ray tracing (Nvidia): il background non ha applicato il ray tracing, ne tanto meno lo necessita, ma il pavimento potrebbe dipendere dal materiale che il'autore decida di utilizzare

Il ray tracing è capace di simulare una vasta varietà di effetti ottici, ad esempio riflessione, rifrazione e fenomeni di dispersione, ma non è necessariamente il metodo più realistico: metodi che includono

tecniche addizionali posso dare simulazioni più accurate.

Il ray tracing è imperfetto: immagini con un ray tracing basico sono molto pulite, quindi l'allineamento degli oggetti e il campionamento possono portare a pattern non intenzionali noti come pattern di moiré e aliasing.

Il ray tracing dà colore per ogni possibile punto dell'immagine. Tuttavia un pixel quadrato contiene un numero infinito di punti e quei punti potrebbero non avere tutti lo stesso colore. Il campionamento viene utilizzato appunto per scegliere il colore di tale pixel, ma come è ben noto, il campionamento implica aliasing.

9. Path Tracing

Path Tracing è un'estensione dell'algoritmo del ray tracing. Esso simula molti percorsi di luce per ogni pixel e ne valuta la media per calcolare il colore finale per ognuno di essi.

Ogni volta che un raggio colpisce una superficie, un nuovo raggio viene tracciato a partire dal punto di collisione in direzione casuale fino a quando raggiunge una massima profondità di percorso (maximum path depth) o finché un meccanismo "uccide" il raggio (simile ad una roulette russa). Di conseguenza, il Path Tracing produce effetti come mescolamento dei colori diffuso (diffuse color bleeding), riflessi lucidi (sfocati), ombre morbide, reali luci di area, vera profondità di campo.

Path Tracing usa il campionamento casuale (metodo Monte Carlo) per incrementare il calcolo dell'immagine finale: il processo di campionamento casuale consente di renderizzare alcuni fenomeni complessi che non vengono gestiti nel ray tracing regolare, ma generalmente ci vuole più tempo per produrre un'immagine tracciata di alta qualità. Tale campionamento introduce del rumore nell'immagine renderizzata. Tale problema si risolve lasciando che l'algoritmo generi più campioni.

9.1. Differenze tra Path Tracing e Ray Tracing

Path Tracing si basa fisicamente sulla simulazione della luce che consente un rendering altamente realistico. È un algoritmo elegante che può simulare molti complessi modi di determinare il percorso e la dispersione della luce nelle scene virtuali. Il Path Tracing usa il tracciamento dei raggi per determinare la visibilità tra gli eventi di scattering. Ray Tracing è una operazione di base che può essere usata per molteplici cose. Quindi, il tracciamento dei raggi da solo non produce in maniera automatica immagini realistiche. Per questo si utilizzano algoritmi del trasporto della luce come il Path Tracing. Tuttavia, anche se elegante e molto potente, il Path Tracing è molto costoso e impiega parecchio tempo per produrre immagini stabili. A tal caso sono stati proposti dei filtri adattivi che riutilizzano più informazioni possibili su molti fotogrammi e pixel al fine di produrre immagini robuste e stabili.

9.2. Rumore nel Ray Tracing

I rendering in real-time tramite GPU utilizzano la rasterizzazione o ciò che è noto come rendering scanline. Per fornire un rendering in tempo reale, i motori di gioco e altri render usano tecniche intelligenti, ma sono falsi o approssimazioni: chiunque usi tali strumenti sa immediatamente quali sono queste limitazioni e il lavoro extra necessario per cercare di approssimare gli effetti desiderati. Il rendering real-time non può creare reali riflessi, rifrazioni, o rimbalzi di luce. Per generare tali effetti di luce, vi è la necessità di approssimare o imitare tali effetti.

Invece di tracciare da centinaia a migliaia di raggi per pixel, per minimizzare i tempi, è stata sviluppata una tecnica chiamata denoising: il concetto è quello di seguire pochi raggi per diminuire il rumore dell'immagine. Ci sono due metodi di denoising:

- **Temporal denoising** renderizza alcuni raggi per ogni frame; essi sono diversi, e nel tempo, la loro media da un risultato fluido. Per un'immagine ferma, il temporal denoising funziona

bene, ma per un'immagine in movimento ci sono problemi con il ghosting.

- **Spatial denoising** viene applicato alle aree rumorose e utilizza un filtro di levigatura su di esse come una sfocatura del bordo. Non funziona bene con l'animazione e crea macchie in movimento. Rimuove anche la nitidezza dagli oggetti che dovrebbero essere nitidi, pertanto viene utilizzato il denoising spaziale. Il denoising spaziale combina i risultati di più pixel vicini come un filtro di sfocatura. Tuttavia, crea luccichii sulle immagini in movimento.

10. Applicazione

Di seguito viene presentata una semplice implementazione del ray tracing in Python.

10.1. Prerequisiti

Quello che necessita sono basiche nozioni di geometria vettoriale:

- Dati due punti A e B , qualunque sia la dimensione ($1, 2, 3, \dots, n$), il vettore che va da A a B può essere calcolato con $B - A$.
- La lunghezza del vettore (norma), che si valuta calcolando la radice quadrata della somma delle componenti vettoriali al quadrato. Si denota con $\|v\|$.
- Versore, vettore di lunghezza 1: $\|v\| = 1$. Il versore di un vettore può essere calcolato effettuando una normalizzazione: $u = \frac{v}{\|v\|}$.
- Prodotto tra vettori.
- Risoluzione di equazioni di secondo grado

11. Algoritmo

Per dare una spiegazione dell'algoritmo vi è la necessità di impostare la scena:

- Si impostino oggetti nello spazio 3D
- Si impostino una sorgente di luce
- Vi è la necessità di impostare un punto (o camera) per osservare la scena
- Affinché la camera osservi, è necessario uno schermo, attraverso il quale osservare gli oggetti

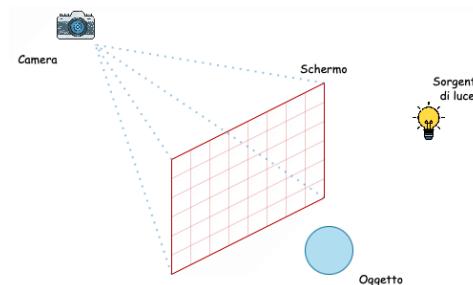


Figura 14

Dato la scena, l'algoritmo del ray tracing si rappresenta come segue:

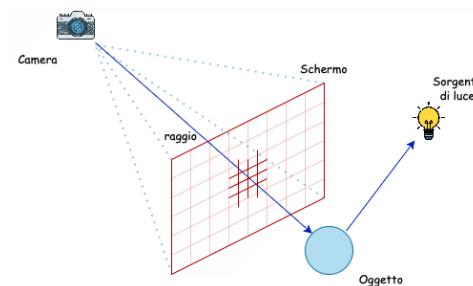


Figura 15

Algorithm 1 Algoritmo del ray tracing

```

for all pixel  $p(x, y, z)$  dello schermo do
     $p \leftarrow$  il colore nero
    if il raggio che parte dalla camera e  $p$  interseca un oggetto della
        scena then
            Calcola il punto d'intersezione con l'oggetto più vicino
            if non ci sono oggetti tra il punto d'intersezione e la luce
                then
                    calcola il colore del punto d'intersezione
                     $p \leftarrow$  colore del punto d'intersezione
                end if
            end if
        end for

```

Alla fine dell'algoritmo lo schermo si riempierà con il corretto colore e l'immagine potrà essere salvata.

11.1. Configurazione della scena

Prima di tutto è necessario configurare la scena. Supponiamo che camera e schermo siano posizionati come in figura.

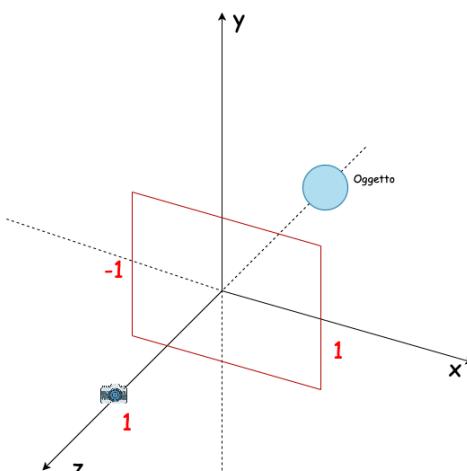


Figura 16. Scena

Da qui, la camera si trova in $(x = 0, y = 0, z = 1)$ e lo schermo è parte del piano formato dall'asse x e y .
Può essere qui scritto uno scheletro del codice.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 width = 500
5 height = 300
6
7 camera = np.array([0, 0, 1])
8 ratio = float(width) / height
9 screen = (-1, 1/ratio, 1, -1/ratio)
10
11 image = np.zeros((height, width, 3))
12 for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
13     for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
14         # image[i, j] = ...
15         print("progress: %d/%d" % (i+1, height))
16
17 plt.imsave('image.png', image)

```

- La camera è solo una posizione (3 coordinate)
- Lo schermo è definito da quattro numeri: sinistra, sopra, destra, sotto. È compreso tra -1 e 1 sull'asse delle x (valore arbitrario), ed è compreso da $\frac{-1}{ratio}$ e $\frac{1}{ratio}$ sull'asse delle y , dove il $ratio = \frac{image_width}{image_height}$. Si fa ciò poiché si vuole l' aspect ratio del

nostro schermo corrisponda con quella dell'immagine che si vuole produrre.

- Il loop divide lo schermo lungo altezze e larghezza in punti, appartenenti al piano $x - y$, per poi calcolare il colore di ogni singolo pixel.

Facendo eseguire tale scheletro l'immagine di output è completamente nera.

11.2. Intersezione dei raggi

Visto che il raggio inizia dalla fotocamera e va nella direzione del pixel mirato in quel momento, si può definire un versore che punta in una direzione simile. Pertanto, si definisce un raggio che partendo dalla camera e va verso il pixel con la seguente equazione:

$$ray(t) = camera + \frac{pixel - camera}{\|pixel - camera\|} t$$

Si ricordi, la fotocamera e il pixel sono punti nello spazio tridimensionale. Per $t = 0$ si ha la posizione della camera, mentre all'aumentare di t più ci si allontana dalla camera nella direzione del pixel.

Possiamo generalizzare l'equazione precedente indicando con O l'origine e D la destinazione:

$$ray(t) = O + \frac{D - O}{\|D - O\|} t = O + d \cdot t$$

con d il vettore di direzione.

Possiamo quindi aggiungere al codice precedente il codice relativo al calcolo del raggio.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def normalization(vector):
5     return vector / np.linalg.norm(vector)
6
7 width = 500
8 height = 300
9
10 camera = np.array([0, 0, 1])
11 ratio = float(width) / height
12 screen = (-1, 1/ratio, 1, -1/ratio)
13
14 image = np.zeros((height, width, 3))
15 for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
16     for j, x in enumerate(np.linspace(screen[0], screen[2], width)):
17         pixel = np.array([x, y, 0])
18         origin = camera
19         direction = normalization(pixel - origin)
20
21         # image[i, j] = ...
22
23         print("progress: %d/%d" % (i+1, height))
24
25 plt.imsave('image.png', image)

```

- È stata aggiunta una funzione chiamata `normalization(vector)` per normalizzare il vettore
- È stato aggiunto il calcolo dell'origine e della direzione; il pixel ha $z = 0$ poiché si trova sullo schermo che si trova su piano $x - y$.

11.3. Definizione della sfera

La parte più complessa è l'intersezione con gli oggetti di scena, poiché essa differisce in base alla tipologia di oggetto. Per semplicità, in tale esempio si prende in considerazione una sfera.

Una sfera è un oggetto molto semplice da definire matematicamente: essa è definita come un insieme di punti che sono tutti alla stessa distanza r (raggio) da un dato punto (centro). Quindi, dato un centro C di una sfera, e il suo raggio r , un arbitrario punto X si trova sulla sfera se e solo se:

$$\|X - C\|^2 = r^2$$

Possiamo quindi già definire nel codice la nostra sfera come segue (la inseriamo dopo la dichiarazione dello schermo)

```
1 objects = [
2     {'center': np.array([-0.2, 0, -1]), 'radius':0.7},
3     {'center': np.array([0.1, -0.3, 0]), 'radius':0.1},
4     {'center': np.array([-0.3, 0, 0]), 'radius':0.15}
5 ]
```

11.4. L'intersezione con la sfera

Conoscendo l'equazione dei raggi e sapendo la condizione che va soddisfatta affinché un punto si trovi sulla superficie sferica, sostituendo la prima di queste nella seconda, si ha:

$$\begin{aligned} \|ray(t) - C\|^2 &= r^2 \\ \Rightarrow \|O + d \cdot t - C\|^2 &= r^2 \\ \Rightarrow \langle d \cdot t + O - C, d \cdot t + O - C \rangle &= r^2 \\ \Rightarrow \langle d, d \rangle t^2 + 2t \langle d, O - C \rangle + \langle O - C, O - C \rangle &= r^2 \\ \Rightarrow \|d\|^2 t^2 + 2t \langle d, O - C \rangle + \|O - C\|^2 - r^2 &= 0 \end{aligned}$$

Questa è un'equazione di secondo grado che può essere risolta per t . Il discriminante è:

$$\begin{aligned} a &= \|d\|^2 = 1 \\ b &= 2\langle d, O - C \rangle \\ c &= \|O - C\|^2 - r^2 \\ \Delta &= b^2 - 4ac \end{aligned}$$

Una volta calcolato il discriminante, ci sono tre possibilità:

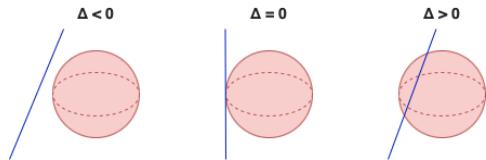


Figura 17

Viene utilizzato il terzo caso ($\Delta > 0$), ovvero l'unico caso in cui si verificano intersezioni. La funzione, che ritorna il valore t , distanza tra l'origine del raggio e il punto d'intersezione, se si verifica $\Delta > 0$, mentre nel caso contrario $\Delta < 0$, il valore None, è così definita:

```
1 def sphere_intersect(center, radius, ray_origin,
2     ray_direction):
3     b = 2* np.dot(ray_direction, ray_origin - center)
4     c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
5     delta = b**2 - 4*c
6     if delta > 0:
7         t1 = (-b + np.sqrt(delta)) / 2
8         t2 = (-b - np.sqrt(delta)) / 2
9         if t1 > 0 and t2 > 0:
10            return min(t1, t2)
11    return None
```

Va notato che si ritorna solo la soluzione più piccola tra le due, che indica l'intersezione più vicina, se ovviamente entrambe le soluzioni sono positive.

11.5. Intersezione con l'oggetto più vicino

Può essere creata facilmente, utilizzando la funzione precedente, un funzione che cerca l'oggetto più vicino che viene intercettato dal raggio, se esiste. Semplicemente, si valutano tutte le sfere in cerca di intersezioni, memorizzando la più vicina.

```
1 def nearest_intersected_object(objects, ray_origin,
2     ray_direction):
3     distances = [sphere_intersect(obj['center'], obj['radius'],
4         ray_origin, ray_direction) for obj in objects]
5     nearest_object = None
6     min_distance = np.inf
7     for index, distance in enumerate(distances):
8         if distance and distance < min_distance:
9             min_distance = distance
10            nearest_object = objects[index]
11    return nearest_object, min_distance
```

Alla chiamata di tale funzione, se l'oggetto più vicino non esiste, non c'è nessun oggetto che viene intersecato dal raggio, mentre in caso contrario viene ritornato l'oggetto più vicino con la rispettiva distanza.

11.6. Punti d'intersezione

Utilizzando le funzioni mostrate finora, si possono calcolare i punti d'intersezione.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def normalization(vector):
5     return vector / np.linalg.norm(vector)
6
7 def sphere_intersect(center, radius, ray_origin,
8     ray_direction):
9     b = 2* np.dot(ray_direction, ray_origin - center)
10    c = np.linalg.norm(ray_origin - center) ** 2 - radius ** 2
11    delta = b**2 - 4*c
12    if delta > 0:
13        t1 = (-b + np.sqrt(delta)) / 2
14        t2 = (-b - np.sqrt(delta)) / 2
15        if t1 > 0 and t2 > 0:
16            return min(t1, t2)
17    return None
18
19 def nearest_intersected_object(objects, ray_origin,
20     ray_direction):
21     distances = [sphere_intersect(obj['center'], obj['radius'],
22         ray_origin, ray_direction) for obj in objects]
23     nearest_object = None
24     min_distance = np.inf
25     for index, distance in enumerate(distances):
26         if distance and distance < min_distance:
27             min_distance = distance
28             nearest_object = objects[index]
29     return nearest_object, min_distance
30
31 width = 500
32 height = 300
33
34 camera = np.array([0, 0, 1])
35 ratio = float(width) / height
36 # sinistra, sopra, destra, sotto
37 screen = (-1, 1/ratio, 1, -1/ratio)
38
39 objects = [
40     {'center': np.array([-0.2, 0, -1]), 'radius':0.7},
41     {'center': np.array([0.1, -0.3, 0]), 'radius':0.1},
42     {'center': np.array([-0.3, 0, 0]), 'radius':0.15}
43 ]
44
45 image = np.zeros((height, width, 3))
46 for i, y in enumerate(np.linspace(screen[1], screen[3],
47     height)):
48     for j, x in enumerate(np.linspace(screen[0], screen[2],
49         width)):
50         pixel = np.array([x, y, 0])
51         origin = camera
52         direction = normalization(pixel - origin)
53
54         # check for intersections
55         nearest_object, min_distance =
56         nearest_intersected_object(objects, origin,
57             direction)
58         if nearest_object is None:
59             continue
60
61         # compute intersection point between ray and
62         # nearest object
63         intersection = origin + min_distance*direction
```

```

57     # image[i, j] = ...
58     print("progress: %d/%d" % (i+1, height))
59
60 plt.imsave('image.png', image)

```

11.7. Intersezione della luce

Finora si è visto che se c'è una linea che parte dalla camera verso un oggetto, si sa qual è e che parte di esso si sta osservando. Non si sa se quell'oggetto sia illuminato o meno. Potrebbe capitare che la luce non colpisca quello specifico punto e quindi non può essere visto. Va quindi controllato se non c'è alcun oggetto della scena tra il punto d'intersezione e la luce.

Si vuole sapere se il raggio che parte dal punto d'intersezione e va verso la luce interseca un oggetto della scena prima di attraversare la luce (utilizziamo la funzione `nearest_intersected_object()`).

Si dichiari la luce come segue:

```

1 light = {'position': np.array([5, 5, 5])}

```

Per verificare se un oggetto sta ombreggiando il punto d'intersezione, va fatto passare il raggio che inizia dal punto d'intersezione e va verso la luce, e vedere se l'oggetto più vicino restituito sia effettivamente più vicino della luce rispetto al punto d'intersezione, in altre parole nel mezzo.

```

1 # ...
2 intersection = origin + min_distance*direction
3
4 intersection_to_light = normalization(light['position']
5                               - intersection)
6 _, min_distance = nearest_intersected_object(objects,
7                                               intersection, intersection_to_light)
8 intersection_to_light_distance = np.linalg.norm(light[
9                               'position'] - intersection)
10 is_shadowed = min_distance <
11      intersection_to_light_distance

```

Questo non funzionerà perché c'è bisogno di una piccola modifica: se si usasse il punto di intersezione come origine, potrebbe essere rilevata la sfera come oggetto tra il punto d'intersezione e la luce. Una soluzione è quella di allontanarsi leggermente dalla sfera nella direzione del vettore normale della superficie.

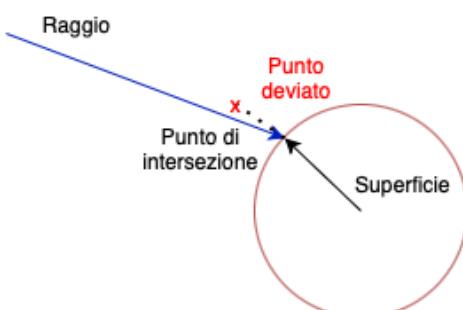


Figura 18

Questa procedura può essere utilizzata per ogni tipo di oggetto. Il codice corretto sarà quindi:

```

1 # ...
2 intersection = origin + min_distance*direction
3
4 normal_to_surface = normalization(intersection -
5                               - nearest_object['center'])
5 shifted_point = intersection + 1e-6*normal_to_surface
6 intersection_to_light = normalization(light['position']
7                               - shifted_point)
8 _, min_distance = nearest_intersected_object(objects,
9                                               shifted_point, intersection_to_light)

```

```

9 intersection_to_light_distance = np.linalg.norm(light[
10                               'position'] - intersection)
11 is_shadowed = min_distance <
12      intersection_to_light_distance
13 if is_shadowed:
14     continue

```

11.8. Modello di riflessione: Blinn-Phong

Finora si è visto che un fascio di luce colpisce un oggetto e il riflesso viene intravisto va verso la camera. Ma cosa vede la camera? Per rispondere a tale domanda interviene il modello di Blinn-Phong. Il modello di Blinn-Phong è un'approssimazione del modello Phong che è computazionalmente più leggero.

In accordo con tale modello, ogni materiale ha 4 proprietà:

- **Colore ambientale:** colore dell'oggetto in assenza di luce;
- **Colore diffuso:** colore percepito come risultato della riflessione diffusa della luce;
- **Colore speculare:** colore di una parte lucida di un oggetto quando è colpito dalla luce;
- **Lucentezza:** coefficiente che indica quanto lucida è una superficie.

Quindi ogni oggetto deve avere queste proprietà, che aggiungiamo al codice

```

1 objects = [
2     {'center': np.array([-0.2, 0, -1]), 'radius':0.7, 'ambient': np.
array([0.1, 0, 0]), 'diffuse': np.
array([0.7, 0, 0]), 'specular': np.array([1, 1,
1]), 'shininess':100},
3     {'center': np.array([0.1, -0.3, 0]), 'radius':0.1, 'ambient': np.
array([0.1, 0, 0.1]), 'diffuse': np.
array([0.7, 0, 0]), 'specular': np.array([1, 1,
1]), 'shininess':100},
4     {'center': np.array([-0.3, 0, 0]), 'radius':0.15, 'ambient': np.
array([0, 0.1, 0]), 'diffuse': np.
array([0, 0.6, 0]), 'specular': np.array([1, 1,
1]), 'shininess':100}
5 ]

```

Tale modello impone queste proprietà anche alla luce, eccetto per la lucentezza.

```

1 light = {'position': np.array([5, 5, 5]), 'ambient': np.
array([1, 1, 1]), 'diffuse': np.array([1, 1, 1]),
'specular':np.array[1, 1, 1]}

```

Date queste proprietà, il modello Blinn-Phong calcola l'illuminazione di un singolo pixel come segue:

$$I_p = k_a \cdot i_a + k_d \cdot i_d \cdot L \cdot N + k_s \cdot i_s \cdot \left(N \cdot \frac{L + V}{\|L + V\|} \right)^{\frac{\alpha}{4}}$$

dove

1. k_a, k_d, k_s sono le proprietà ambientali, di diffusione, speculari di un oggetto;
2. i_a, i_d, i_s sono le proprietà ambientali, di diffusione, speculari della luce;
3. L è la direzione del versore che parte dal punto d'intersezione verso la luce;
4. N è il versore della normale della superficie nel punto d'intersezione;
5. V il versore che va dal punto d'intersezione verso la camera
6. α è la lucentezza dell'oggetto

```

1 if is_shadowed:
2     break
3
4 # RGB
5 illumination = np.zeros((3))

```

```

6 # ambient
7 illumination += nearest_object['ambient'] * light['ambient']
8
9 # diffuse
10 illumination += nearest_object['diffuse'] * light['diffuse'] * np.dot(intersection_to_light,
11     normal_to_surface)
12
13 # specular
14 intersection_to_camera = normalization(camera -
15     intersection)
15 H = normalization(intersection_to_light +
16     intersection_to_camera)
16 illumination += nearest_object['specular'] * light['specular'] * np.dot(normal_to_surface, H) ** (
17     nearest_object['shininess'])/4
18 image[i, j] = np.clip(illumination, 0, 1)

```

Nell'ultima linea di codice è stato limitato il colore tra 0 e 1 per non avere problemi.

11.9. Primo risultato

Eseguendo il codice, incrementando altezza e larghezza dello schermo (comporta un aumento del tempo di calcolo), quello che si ottiene è la seguente immagine

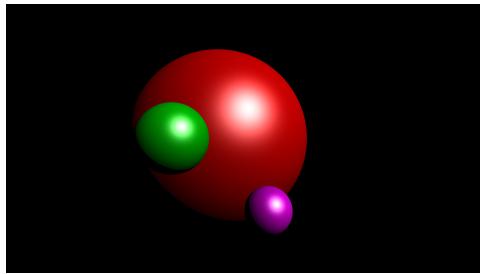


Figura 19

Vedendo il risultato, è possibile aggiungere il riflesso alle sfere e un piano.

11.10. Finto piano

Un finto piano può essere renderizzato utilizzando una quarta sfera, impostando un raggio quasi infinito. Questo permette di percepire la sua superficie come un piano d'appoggio.

Quindi la soluzione è aggiungere alle sfere

```

1 { 'center': np.array([0, -9000, 0]), 'radius':
2     90000-0.7, 'ambient': np.array([0.1, 0.1, 0.1]), 'diffuse':
3     np.array([0.6, 0.6, 0.6]), 'specular':
4     np.array([1, 1, 1]), 'shininess': 100, 'reflection':
5     0.5 }

```

11.11. Riflessione

Il raggio che è renderizzato finora, parte dalla sorgente luminosa, colpisce la superficie di un oggetto e rimbalza verso la camera. Potrebbe capitare che prima di arrivare alla camera colpisca altri oggetti prima: questo è il fenomeno della riflessione. Il raggio accumula differenti colori e quando colpirà la camera si vedrà il riflesso.

Ogni oggetto deve avere un coefficiente di riflessione compreso tra 0 e 1, dove con 0 si indica che non riflette luce, mentre 1 è una superficie specchio. Si aggiunga tale coefficiente alle sfere:

```

1 objects = [
2     { 'center': ... , 'reflection': 0.5 },
3     { 'center': ... , 'reflection': 0.5 },
4     { 'center': ... , 'reflection': 0.5 },
5     { 'center': ... , 'reflection': 0.5 }
6 ]

```

Per includere la riflessione, vi è la necessità di tracciare il raggio riflesso dopo che vi è un'intersezione e di includere il contributo del colore per ogni punto di intersezione. Tale operazione va ripetuta un certo numero di volte che va definito.

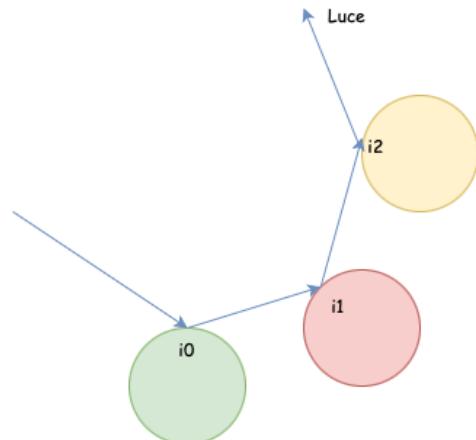


Figura 20

Per il colore di ogni pixel, bisogna sommare il contributo per ogni punto d'intersezione del raggio:

$$c_p = i_0 + r_0 i_1 + r_0 r_1 i_2 + r_0 r_1 r_2 i_3 + \dots$$

dove

- c è il colore finale del pixel;
- i l'illuminazione calcolata; attraverso il modello Blinn-Phong nell' i -esimo punto d'intersezione
- r è la riflessione dell' i -esimo oggetto intersecato.

Si sceglie arbitrariamente quando fermare la somma.

La direzione del raggio può essere calcolata come segue:

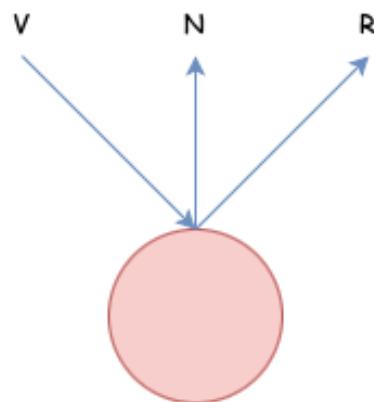


Figura 21

$$R = V - 2(V \cdot N)N$$

dove,

- R è il versore del raggio riflesso
- V è il versore del raggio che colpisce la superficie
- N è il versore della normale della superficie nel punto in cui viene colpita dal raggio

Si aggiunge tale funzione

```

1 def reflected(vector, axis):
2     return vector - 2 * np.dot(vector, axis) * axis

```

Il main modificato è quindi:

```

1  for i, y in tqdm(enumerate(np.linspace(screen[1],
2    screen[3], height)), position=0):
3    for j, x in enumerate(np.linspace(screen[0], screen
4      [2], width)):
5      pixel = np.array([x, y, 0])
6      origin = camera
7      direction = normalization(pixel - origin)
8
9      color = np.zeros((3))
10     reflection = 1
11
12     for k in range(max_depth):
13       # check for intersections
14       nearest_object, min_distance =
15       nearest_intersected_object(objects, origin,
16         direction)
16       if nearest_object is None:
17         break
18
19       # compute intersection point between ray
20       and nearest object
21       intersection = origin + min_distance *
22       direction
23
24       normal_to_surface = normalization(
25         intersection - nearest_object['center'])
26       shifted_point = intersection + 1e-5 *
27       normal_to_surface
28       intersection_to_light = normalization(light
29         ['position'] - shifted_point)
30
31       _, min_distance =
32       nearest_intersected_object(objects, shifted_point,
33         intersection_to_light)
34       intersection_to_light_distance = np.linalg.
35       norm(light['position'] - intersection)
36       is_shadowed = min_distance <
37       intersection_to_light_distance
38
39       if is_shadowed:
40         break
41
42       # RGB
43       illumination = np.zeros((3))
44
45       # ambient
46       illumination += nearest_object['ambient'] *
47       light['ambient']
48
49       # diffuse
50       illumination += nearest_object['diffuse'] *
51       light['diffuse'] * np.dot(intersection_to_light,
52       normal_to_surface)
53
54       # specular
55       intersection_to_camera = normalization(
56         camera - intersection)
57       H = normalization(intersection_to_light +
58         intersection_to_camera)
59       illumination += nearest_object['specular'] *
60       light['specular'] * np.dot(normal_to_surface, H)
61       ** (nearest_object['shininess'] / 4)
62
63       # reflection
64       color += reflection * illumination
65       reflection *= nearest_object['reflection']
66
67       # new ray origin and direction
68       origin = shifted_point
69       direction = reflected(direction,
70       normal_to_surface)
71
72     image[i, j] = np.clip(color, 0, 1)

```

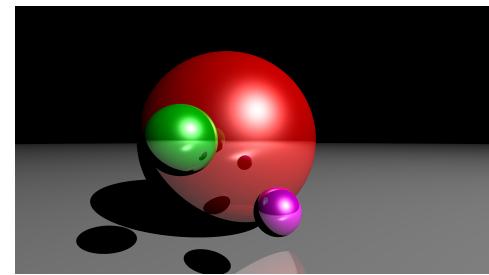


Figura 22

Il tempo impiegato per rendering è stato di 21 : 21 minuti con una risoluzione 3840×2160

11.12. Generazione di un video

Volendo generare un video, è possibile generare i diversi frame facendo variare la posizione della camera.

È stata inserita la classe `tqdm` per avere più informazioni durante il processo Eseguendo il codice completo, l'immagine di output è

Riferimenti bibliografici

- [1] B. Saleh, *Computer Graphics Fundamental: Lighting and shading*, mag. 2017. indirizzo: https://www.academia.edu/33137083/Computer_Graphics_Fundamental_Lighting_and_Shading.
- [2] J. Peddie, *Ray Tracing: A tool for all*. Springer, 2019.
- [3] A. Omar, *Ray Tracing From Scratch in Python*, lug. 2020. indirizzo: <https://omaraflak.medium.com/ray-tracing-from-scratch-in-python-41670e6a96f9>.