

Lighting and Shading: il modello del Ray Tracing

Antonio Sirignano

Università degli Studi di Napoli Federico II

Calcolo Scientifico per l'Innovazione Tecnologica

7 giugno 2024

Introduzione

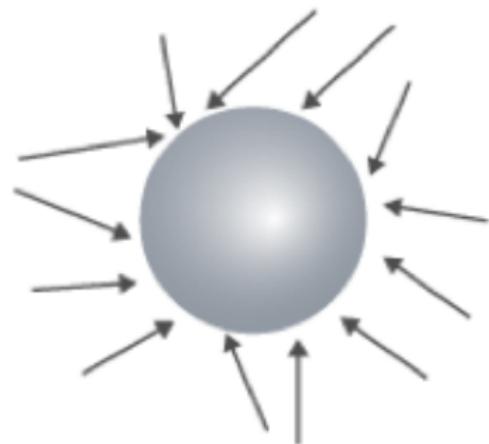
- ▶ **Computer Graphic Technology** è l'abilità di produrre un effetto visivo realistico in un oggetto tridimensionale in un device di output bidimensionale (computer o foglio stampato)
- ▶ Metodi di rendering
 - ▶ Shading
- ▶ Entità:
 - ▶ **Sorgenti di luce:** intensità, colore, forma, direzione, distanza e dimensione.
 - ▶ **Superficie dell'oggetto:** lucida, liscia, ruvida, brillante e scura. Colore opaco, trasparente e traslucido.
 - ▶ **Ambiente:** spazio vuoto o spazio contenente altri oggetti.

Sorgenti di luce

Un oggetto illuminato dalla luce è colpito da raggi luminosi proiettati sulla sua superficie da un emittente chiamato **sorgente di luce**.

Sorgenti di luce

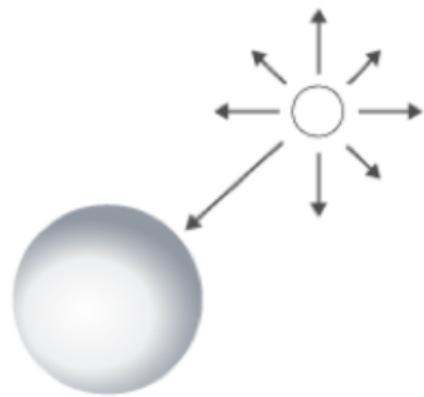
Luce ambientale



1. Sorgente di luce non direzionale.
2. Luce emessa in ogni direzione.
3. Intensità indipendente da tutte le sue caratteristiche (posizione e orientamento).

Sorgenti di luce

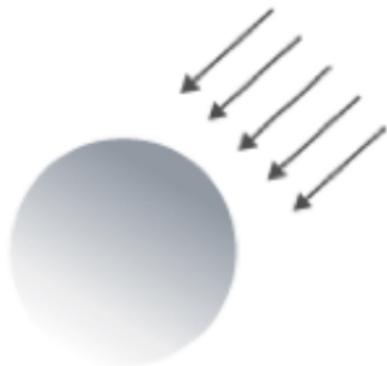
Luce puntiforme



1. Sorgente di luce non direzionale.
2. Luce emessa in ogni direzione.
3. Intensità indipendente da tutte le sue caratteristiche (posizione e orientamento).

Sorgenti di luce

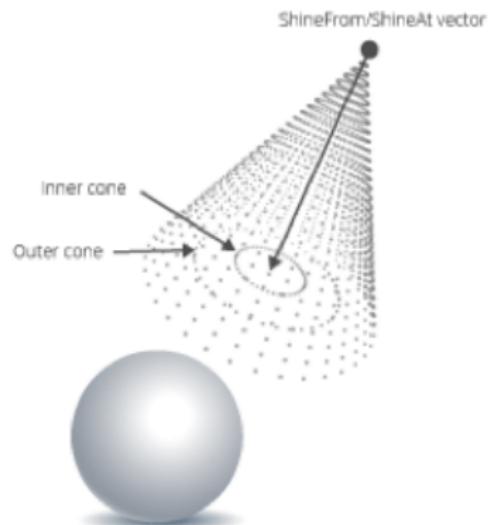
Luce direzionale



1. Sorgente di luce posizionata ad una distanza infinita dalla scena.
2. Tutti i raggi di luce si espandono in una singola direzione e tutti con la stessa intensità.
3. Caratterizzata da colore, intensità e direzione.

Spotlight

Luce ambientale



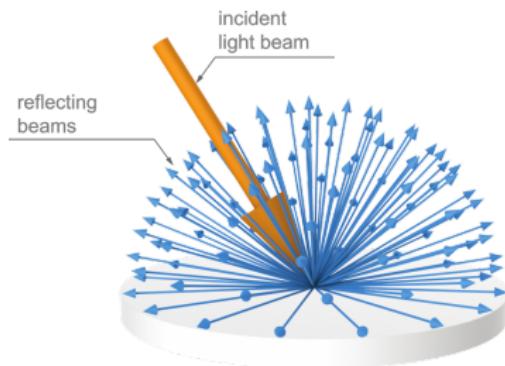
1. Luce irradiata in un cono con più luce al centro.
2. Punto di propagazione, asse di direzione, raggio intorno all'asse e funzione di decadimento radiale.

Modelli di riflessioni

L'obiettivo principale dello shading è la produzione di un risultato accettabile quando la superficie è affetta dai raggi di luce.

Modelli di riflessioni

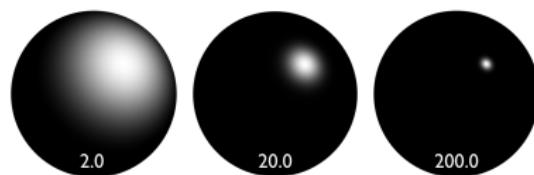
Riflessione diffusa



1. Diffusione della luce uniformemente in tutte le direzioni.
2. Vale la legge di Lambert: la diffusione del riflesso è proporzionale al coseno dell'angolo θ compreso tra la normale N e la direzione della sorgente L .

Modelli di riflessioni

Riflessione speculare



1. Produce una riflessione luminosa sulla superficie dell'oggetto
2. Il modello Phong stabilisce che

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

con k_s coefficiente di riflessione, $0 \leq k \leq 1$, e ϕ angolo compreso tra la direzione della riflessione r e la direzione dell'osservatore v .

3. All'incrementare di n , la luce riflessa si concentra in una regione superficiale più piccola.

Modelli di shading

I modelli di shading sono utilizzati per ottenere il modello di illuminazione desiderato.

Modelli di shading

Constant shading

1. Modello più semplice di shading
2. Lo stesso colore è applicato su un intero poligono con un rendering veloce.
3. L'equità della luce è utilizzata una sola volta per poligono
4. Data una singola normale sul piano, l'equazione della luce e le proprietà del materiale sono usate per generare un singolo colore

Modelli di shading

Gouraud shading

1. I colori sono interpolati attraverso il poligono e vi è la necessità di identificare ogni vertice.
2. L'equità della luce è applicata ad ogni vertice.
3. Ogni colore è determinato dalla quantità di luce e dalla proprietà del materiale.

Modelli di shading

Phong shading

1. Modello più realistico rispetto ai precedenti.
2. Considera l'unione delle normali dei vertici ad ogni punto del poligono per avere una normale locale.
3. Il calcolo è applicato per avere un'illuminazione totale nel rendering.

Modelli di shading

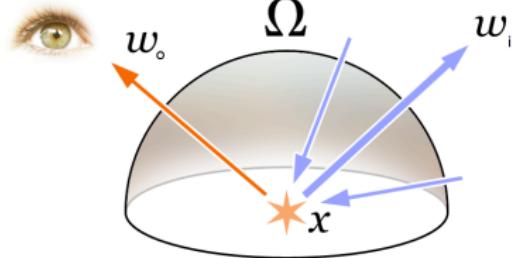


- ▶ L'ambizione di artisti, designer, ingegneri è sempre stata la simulazione dell'immagine di una scena, una storia, un prodotto, un palazzo, prima che siano effettivamente costruiti e realizzati.
- ▶ Vi è quindi la necessità di disporre di immagini, o video, realistiche a basso costo e in poco tempo

L'equazione del rendering

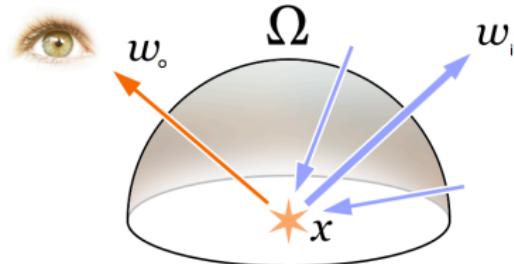
- ▶ Nella computer graphics, l'**equazione del rendering** è un'equazione integrale nella quale la radianza di equilibrio che lascia un punto è la somma della radianza emessa più la radianza riflessa sotto un'approssimazione dell'ottica geometrica.
- ▶ Esistono svariate tecniche per risolvere quest'equazione.

L'equazione del rendering



- ▶ Tale equazione 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 riflettanza, dove:

L'equazione del rendering

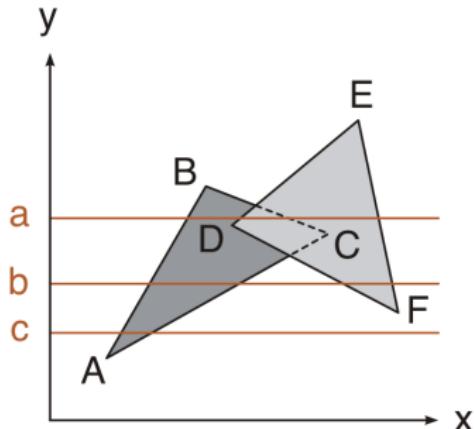


1. x è una posizione nello spazio
2. w_o è la direzione della luce in uscita
3. Ω è l'emisfero unitario centrato intorno $\{n\}$ contenente tutti i valori possibili di w_i
4. w_i è il fattore di indebolimento dell'irraggiamento verso l'esterno a causa dell'angolo d'incidenza.

Scanline rendering

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

Scanline rendering



1. Tutti i poligoni da renderizzare sono ordinati dall'alto sull'asse delle ordinate in ordine di apparizione.
2. Ogni riga (scanline) dell'immagine è calcolata utilizzando l'intersezione di una scanline con i poligoni nella parte interiore della lista ordinata.
3. La lista viene aggiornata scartando i poligoni non più visibili man mano che si avanza.
4. L'ordine dei vertici lungo la normale del piano di scansione, riduce il numero di comparazioni tra i bordi.

Scanline rendering

Z-buffering

- ▶ Lo **Z-buffering** è la gestione delle coordinate di profondità delle immagini nella grafica 3D, fatta a livello hardware.
- ▶ Quando un oggetto viene proiettato sullo schermo un oggetto con un motore di rendering 3D, la profondità (*z-value*) del pixel generato viene memorizzato 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* memorizza una *z-value* per ogni pixel.

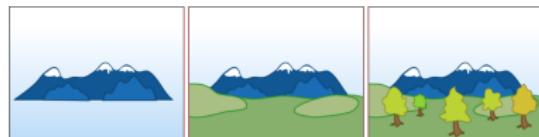
Scanline rendering

Z-buffering

- ▶ Non è possibile vedere completamente oggetti o superfici che siano parzialmente o totalmente opachi, indipendentemente dalla distanza dell'osservatore.
- ▶ 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 venga memorizzata nello *z-buffer*.

Scanline rendering

Algoritmo del pittore

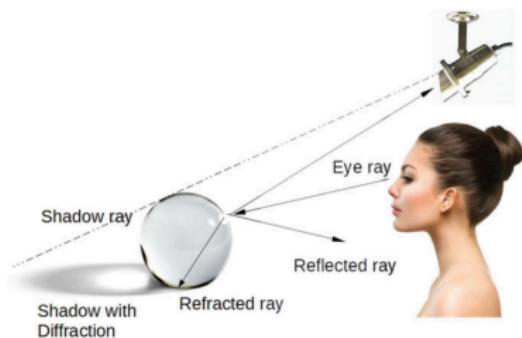


- ▶ L'algoritmo del pittore è una delle soluzioni più semplici al problema della visibilità nella grafica 3D.
- ▶ Ordina tutti i poligoni nella scena partendo dalla profondità e poi li disegna in questo ordine: dal più lontano al più vicino.
- ▶ Si disegna sulle parti non visibili, al costo di avere 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 tra i due oggetti disegnati, viene oscurato

Ray Tracing

- ▶ Negli ultimi anni, la qualità delle immagini generate computazionalmente hanno aggiunto un livello di realismo tale che i rendering sono indistinguibili 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.
- ▶ Tale tecnica può produrre un grado di realismo visivo molto alto ma con un alto costo computazionale.

Ray Tracing



► Ci sono almeno 4 raggi coinvolti nel ray tracing:

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

Ray Tracing

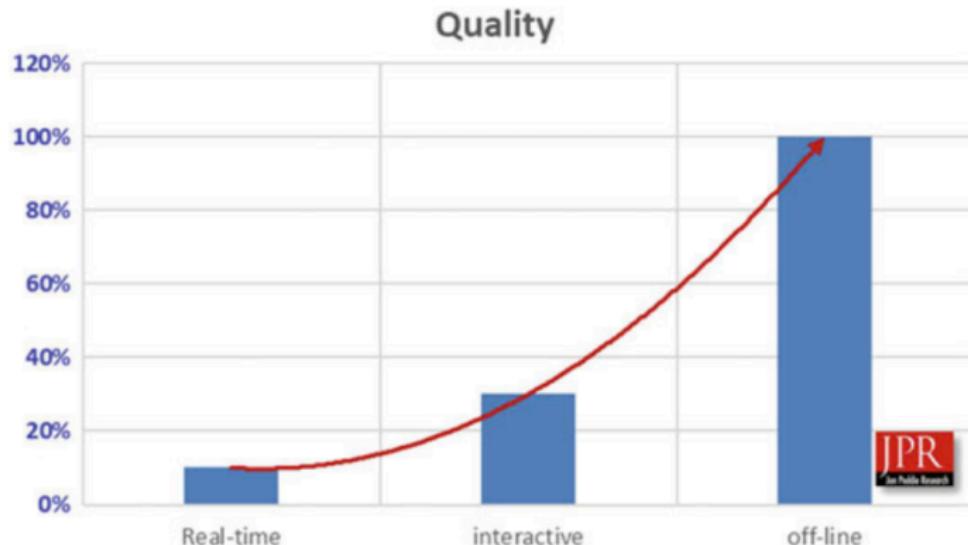
- ▶ Calcola il raggio dell'occhio dell'osservatore attraverso ogni pixel, calcola il punto di intersezione più vicino ad una superficie di scena, quindi ombreggia quel punto calcolando i raggi d'ombra e genera i raggi riflessi rifratti.
- ▶ Simula la riflessione speculare e le ombre attraverso e fuori le superfici trasparenti.
- ▶ Può usare o impiegare l'illuminazione indiretta, a volte aree di sorgenti luminose o altre influenze caustiche.

Ray Tracing

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

- ▶ **Off-line:** usato ampiamente dagli studi cinematografici, pubblicitari e di design. Ha la qualità maggiore
- ▶ **Interactive:** riduce il numero dei raggi, in modo da avere comunque una buona immagine e allo stesso tempo di offrire all'utente la possibilità di manipolare l'utente.
- ▶ **Real-Time:** può essere utilizzato, con alcune restrizioni, dall'assistenza cdi piccoli supercomputer.

Ray Tracing



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
- ▶ Ha la capacità di simulare una vasta varietà di effetti ottici: riflessione, rifrazione e fenomeni di dispersione.
- ▶ Non è necessariamente il metodo più realistico: usando tecniche addizionali si possono avere simulazioni più accurate.

Ray Tracing

Il ray tracing è imperfetto

- ▶ Immagini con un ray tracing basico sono molto pulite, quindi l'allineamento degli oggetti e il campionamento posso portare a pattern non intenzionali,
 1. Pattern di moiré
 2. Aliasing
- ▶ Dando colore per ogni possibile punto dell'immagine e sapendo che un pixel contiene un numero infinito di punti al suo interno che potrebbero avere uno stesso colore, si utilizza il campionamento per scegliere il colore da dare a tale pixel. Ma campionamento è sinonimo di aliasing.

Path Tracing

- ▶ Il Path Tracing è un'estensione dell'algoritmo del ray tracing.
- ▶ Simula molti percorsi di luce per ogni pixel e ne valuta la media per calcolare il colore finale di 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* o finché un meccanismo non *uccide* il raggio (simile ad un roulette russa).

Path Tracing

- ▶ Path Tracing produce effetti come mescolamento dei colori diffuso (*diffuse color bleeding*), riflessi lucidi, ombre morbide, reali luci di area, vera profondità di campo.
- ▶ Usa il campionamento casuale per incrementare il calcolo dell'immagine finale.
 - ▶ Consente il rendering di alcuni fenomeni complessi che non vengono gestiti dal ray tracing, ma generalmente ci vuole più tempo per produrre un'immagine tracciata di alta qualità.
 - ▶ Il problema è l'introduzione del rumore nell'immagine che si risolve lasciando che l'algoritmo generi più campioni.

Path Tracing

Differenze tra Ray Tracing e Path Tracing

- ▶ Path Tracing si basa fisicamente sulla simulazione della luce che consente un rendering altamente realistico; è un algoritmo elegante che può simulare molti diversi modi di determinare un percorso e la dispersione della luce nelle scene virtuali.
- ▶ Il Path Tracing usa il tracciamento dei raggi per determinare la visibilità degli oggetti
- ▶ Il Ray Tracing è un'operazione di base che può essere usata per molteplici cose; ma il tracciamento dei raggi da solo non produce in maniera automatica immagini realistiche.
- ▶ Il Path Tracing è molto costoso e impiega parecchio tempo per produrre immagini stabili.

Applicazione

Implementazione del Ray Tracing tramite il linguaggio Python

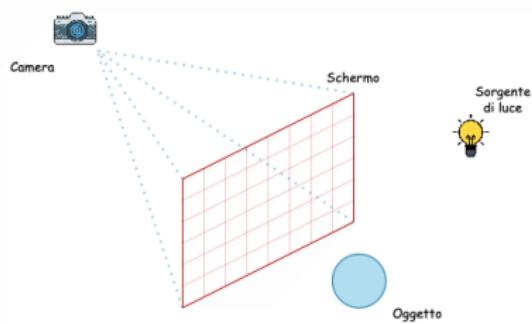
Applicazione

Prerequisiti

Ciò che è sono basiche nozioni di geometria vettoriale

1. Dati due punti A e B . qualunque sia la loro dimensione, il vettore che va da A a B si calcola come $B - A$.
2. $\|A\|$ denota la lunghezza del vettore (norma).
3. Versore, ovvero vettore di lunghezza 1. Il versore di un generico vettore si calcola tramite normalizzazione: $u = \frac{v}{\|v\|}$
4. Prodotto tra vettori.
5. Risoluzioni equazioni di secondo grado.

Algoritmo

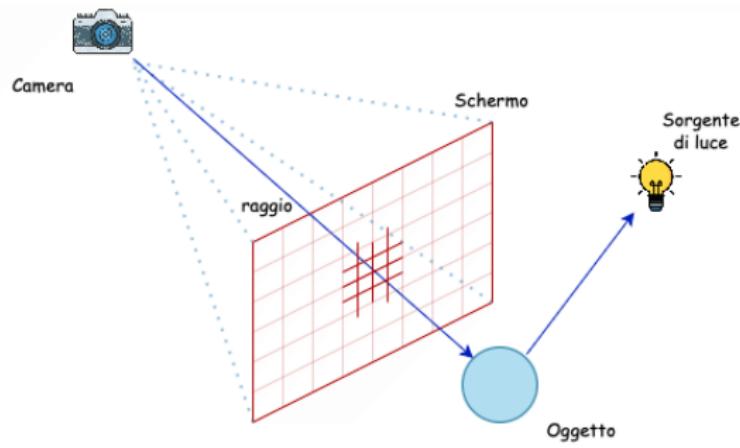


► Per dare una spiegazione dell'algoritmo vi è la necessità di impostare la scena:

1. Si impostino oggetti nello spazio 3D.
2. Si imposti una sorgente di luce.
3. Si imposti un punto per osservare la scena.
4. Affinché la camera osservi, è necessario uno schermo, attraverso il quale osservare gli oggetti.

Algoritmo

- ▶ Dato la scena, l'algoritmo del Ray Tracing può essere rappresentato come segue:



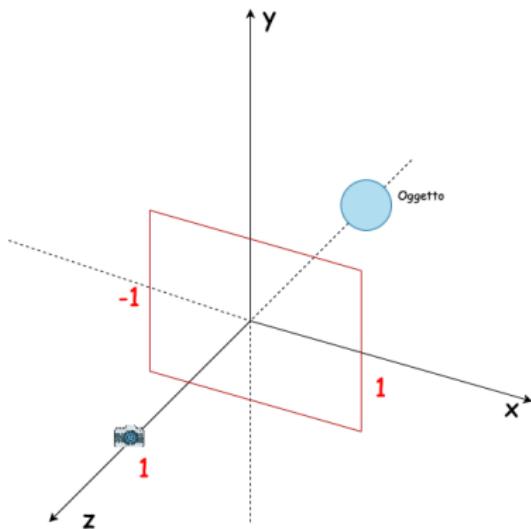
- ▶ Alla fine dell'algoritmo lo schermo si riempirà con il corretto colore e l'immagine potrà essere salvata.

Algoritmo I

```
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
```

Algoritmo

Configurazione della scena



- ▶ È necessario configurare la scena:
 1. Supponiamo che la camera e lo schermo siano posizionati come in figura.
 2. La camera si trova in $(x = 0, y = 0, z = 1)$ e lo schermo è parte del piano formato da x e y .

Algoritmo

Configurazione della scena

```
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],
13     height)):
13     for j, x in enumerate(np.linspace(screen[0], screen
14         [2], width)):
15         # image[i, j] = ...
16     print("progress: %d/%d" % (i+1, height))
17 plt.imsave('image.png', image)
```

Algoritmo

Configurazione della scena

- ▶ La camera è solo una posizione
- ▶ Lo schermo è definito da quattro valori, corrispondenti a sinistra, sopra, destra, sotto. È compreso tra 0 e 1 sull'asse delle x (valori arbitrari), ed è compreso da $\frac{-1}{ratio}$ e $\frac{1}{ratio}$ sull'asse delle y , dove $ratio = \frac{image_width}{image_height}$. Si fa ciò perché si vuole che l'aspect ratio del nostro schermo corrisponda con quella dell'immagine che si vuole riprodurre.
- ▶ Il loop divide lo schermo lungo altezza e lunghezza in punti, appartenenti al piano $x-y$, per poi calcolare il colore di ogni singolo pixel.

Algoritmo

Intersezione dei raggi

- ▶ Si definisce un versore che indichi la direzione di un raggio partente dalla camera in direzione di un pixel mirato.
- ▶ L'equazione di tale raggio è

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

- ▶ Per $t = 0$ si ha la posizione della camera; all'aumentare di t ci si allontana sempre di più dalla camera rispetto al pixel.
- ▶ Una generalizzazione di tale equazione, indicando con O l'origine e D la destinazione è:

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

con d il versore.

Algoritmo

Intersezione dei raggi

```
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],
16                                 height)):
16     for j, x in enumerate(np.linspace(screen[0], screen
17                                     [2], width)):
18         pixel = np.array([x, y, 0])
19         origin = camera
20         direction = normalization(pixel - origin)
21
22         # image[i, j] = ...
23
24         print("progress: %d/%d" % (i+1, height))
25 plt.imsave('image.png', image)
```

Algoritmo

Intersezione dei raggi

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

Algoritmo

Definizione della sfera

- ▶ La parte più complessa è l'intersezione con gli oggetti di scena, in quanto differisce in base all'oggetto.
- ▶ Per semplicità in tale applicazione utilizziamo la **sfera**.
- ▶ Una sfera è definita come un insieme di punti che sono tutti alla stessa distanza r , raggio, da un dato punto, centro.
- ▶ Dato un centro C e il raggio r , un punto X si trova sulla sfera se:

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

Algoritmo

Definizione della sfera

```
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 ]
```

Algoritmo

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 questa nella seconda si ha:

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

- ▶ Un'equazione di secondo grado che può essere risolta per t

Algoritmo

L'intersezione con la sfera

- ▶ Il discriminante è:

$$a = \|d\|^2 = 1$$

$$b = 2\langle d, O - C \rangle$$

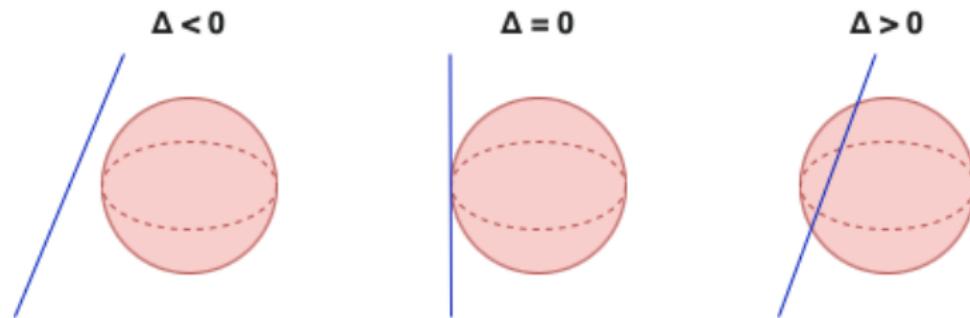
$$c = \|O - C\|^2 - r^2$$

$$\Delta = b^2 - 4ac$$

Algoritmo

L'intersezione con la sfera

- ▶ Una volta calcolato il discriminante, ci sono tre possibilità:



- ▶ Viene utilizzato il caso $\Delta > 0$, ovvero l'unico caso in cui vi sono intersezioni.
- ▶ La funzione che ritorna il valore t , distanza tra l'origine del raggio e il punto d'intersezione, o valore None se si è verificato $\Delta < 0$, è definita come segue:

Algoritmo

Intersezione con la sfera

```
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 -
5         radius ** 2
6     delta = b**2 - 4*c
7     if delta > 0:
8         t1 = (-b + np.sqrt(delta)) / 2
9         t2 = (-b - np.sqrt(delta)) / 2
10        if t1 > 0 and t2 > 0:
11            return min(t1, t2)
12    return None
```

Algoritmo

Intersezione con la sfera

- ▶ Può essere facilmente creata una funzione che ritorna l'oggetto più vicino che viene intercettato dal raggio, se esiste, valutando semplicemente 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[‘
4         radius'], ray_origin, ray_direction) for obj in
5                   objects]
6     nearest_object = None
7     min_distance = np.inf
8     for index, distance in enumerate(distances):
9         if distance and distance < min_distance:
10             min_distance = distance
11             nearest_object = objects[index]
12
13 return nearest_object, min_distance
```

Algoritmo

Punti d'intersezione

- ▶ Utilizzando le funzioni mostrate finora possiamo calcolare i punti d'intersezione.

```
43 for i, y in enumerate(np.linspace(screen[1], screen[3],
        height)):
44     for j, x in enumerate(np.linspace(screen[0], screen
        [2], width)):
45         pixel = np.array([x, y, 0])
46         origin = camera
47         direction = normalization(pixel - origin)
48
49         # check for intersections
50         nearest_object, min_distance =
51             nearest_intersected_object(objects, origin,
52                 direction)
53         if nearest_object is None:
54             continue
55
56         # compute intersection point between ray and
57         # nearest object
58         intersection = origin + min_distance*direction
```

Algoritmo

Intersezione della luce

- ▶ A questo punto bisogna capire se l'oggetto che viene colpito dal raggio uscente della camera sia illuminato o meno.
- ▶ Potrebbe capitare che l'oggetto non sia colpito dalla luce o che ci sia un altro oggetto nel mezzo.
- ▶ Si vuole quindi sapere se il raggio che parte dal punto d'intersezione e va verso la luce interseca prima un oggetto della scena prima di attraversare la luce.
- ▶ La luce viene dichiarata come segue:

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

Algoritmo

Intersezione della luce

- ▶ 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 alla luce rispetto al punto d'intersezione:

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

Algoritmo

Intersezione della luce

- ▶ Il codice precedente non funzionerà!
- ▶ Se si usasse il punto d'intersezione come origine, potrebbe essere rilevata la sfera come oggetto tra il punto d'intersezione e la luce.
- ▶ Una soluzione è allontanarsi leggermente dalla sfera nella direzione del vettore normale della superficie.



Algoritmo

Intersezione della luce

```
1 # ...
2 intersection = origin + min_distance*direction
3
4 normal_to_surface = normalization(intersection -
5     nearest_object['center'])
6 shifted_point = intersection + 1e-6*normal_to_surface
7 intersection_to_light = normalization(light['position'] -
8     shifted_point)
9 _, min_distance = nearest_intersected_object(objects,
10     shifted_point, intersection_to_light)
11 intersection_to_light_distance = np.linalg.norm(light[,
12         'position'] - intersection)
13 is_shadowed = min_distance <
14     intersection_to_light_distance
15
16 if is_showed:
17     continue
```

Algoritmo

Modello di riflessione: Blinn-Phong

- ▶ Il modello di Blinn-Phong è un'approssimazione del modello di Phong, computazionalmente più leggero.
- ▶ In accordo con tale modello, ogni materiale ha 4 proprietà:
 1. **Colore ambientale**: colore dell'oggetto in assenza di luce.
 2. **Colore diffuso**: colore percepito come risultato della riflessione diffusa della luce.
 3. **Colore speculare**: colore di una parte lucida di un oggetto quando è colpito dalla luce.
 4. **Lucentezza**: coefficiente che indica quanto lucida è una superficie.

Algoritmo

Modello di riflessione: Blinn-Phong

```
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, 0.7]), '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 ]
6
7 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]}
```

Algoritmo

Modello di riflessione: Blinn-Phong

- ▶ 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

Algoritmo

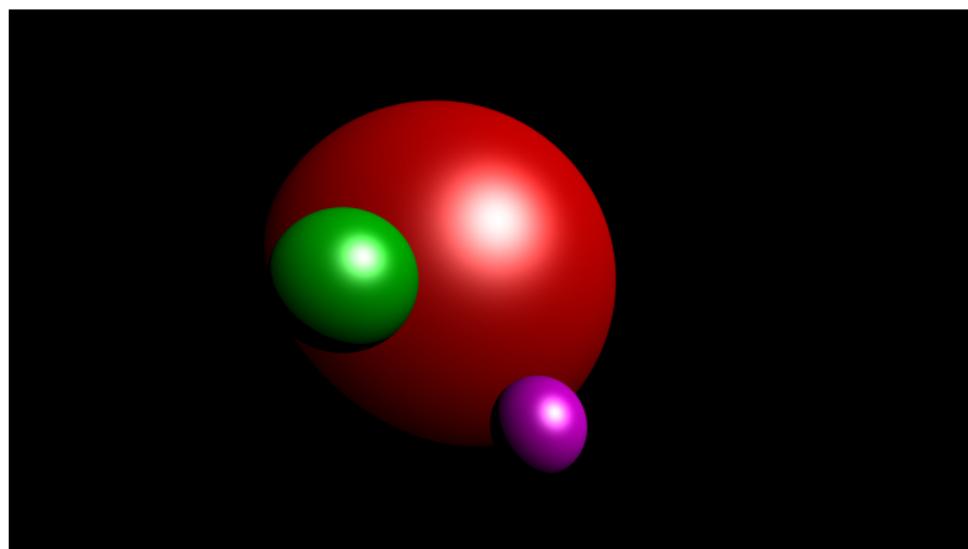
Modello di riflessione: Blinn-Phong

```
1 if is_shadowed:
2     break
3
4 # RGB
5 illumination = np.zeros((3))
6
7 # ambient
8 illumination += nearest_object['ambient'] * light['ambient']
9
10 # diffuse
11 illumination += nearest_object['diffuse'] * light['diffuse'] * np.dot(intersection_to_light,
12                         normal_to_surface)
13
14 # specular
15 intersection_to_camera = normalization(camera -
16                                         intersection)
17 H = normalization(intersection_to_light +
18                     intersection_to_camera)
19 illumination += nearest_object['specular'] * light['specular'] * np.dot(normal_to_surface, H) ** (
20                         nearest_object['shininess']/4)
21
22 image[i, j] = np.clip(illumination, 0, 1)
```

Algoritmo

Primo risultato

- ▶ Eseguendo il codice, incrementando altezza e lunghezza dello schermo (3840×2160), quello che si ottiene è la seguente immagine



Algoritmo

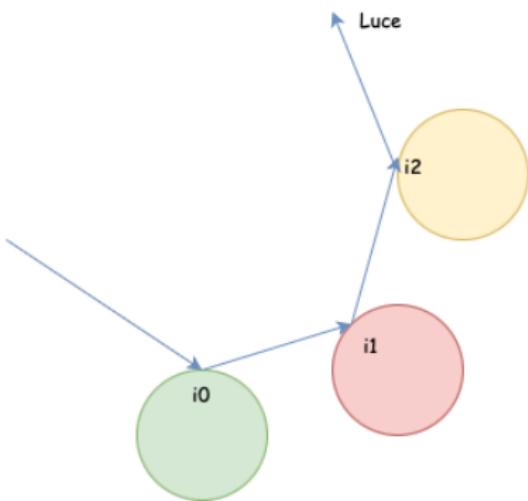
Riflessione

- ▶ Il raggio renderizzato finora, parte dalla sorgente luminosa, colpisce la superficie di un oggetto e rimbalza verso la camera.
- ▶ Potrebbe capitare che prima colpisca altri oggetti: tale fenomeno è detto **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 0 indica che non riflette luce, mentre 1 è una superficie specchio.

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

Algoritmo

Riflessione



- ▶ 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 d'intersezione
- ▶ Tale operazione va ripetuta per un certo numero di volte, che è scelto arbitrariamente.

Algoritmo

Riflessione

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

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

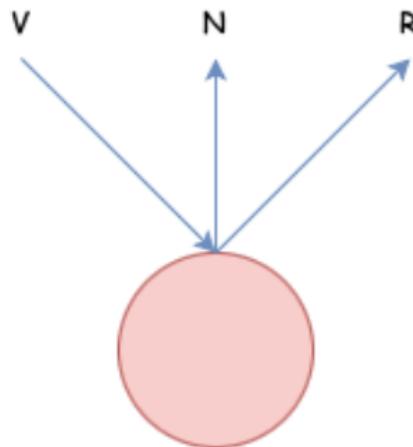
dove

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

- ▶ Si sceglie arbitrariamente quando fermare la somma.

Algoritmo

Riflessione



► La direzione del raggio è calcolata come segue:

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

dove

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

Algoritmo

Riflessione

- ▶ Viene aggiunta quindi tale funzione

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

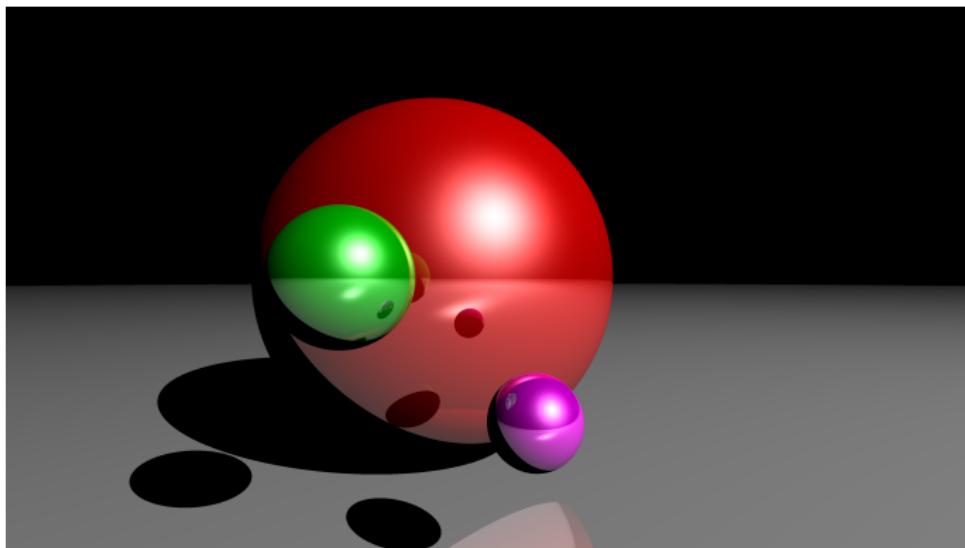
Algoritmo

Riflessione

- ▶ Per il codice completo cliccare qui

Algoritmo

Risultato finale



Algoritmo

Risultato finale

- ▶ Il tempo impiegato per il rendering è stato di 21:21 minuti con una risoluzione 3840×2160 .
- ▶ Volendo generare un video, è possibile renderizzare singoli frame facendo variare la posizione della camera.

Fine