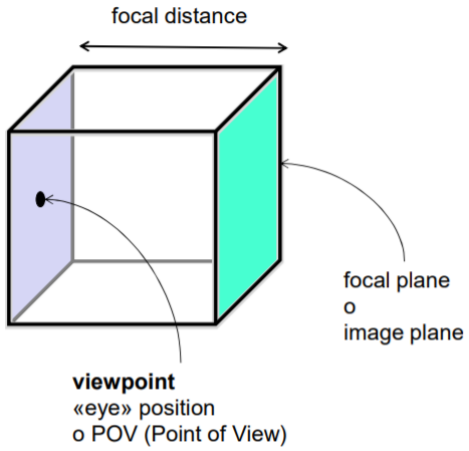
H1 Ray Tracing

H2 Introduzione

Per introdurre l'idea dietro al **Ray Tracing**, possiamo ricondurci a un modello di sintesi di immagini a partire dalla realtà, ma semplificato e astratto: la *pin-hole camera*.

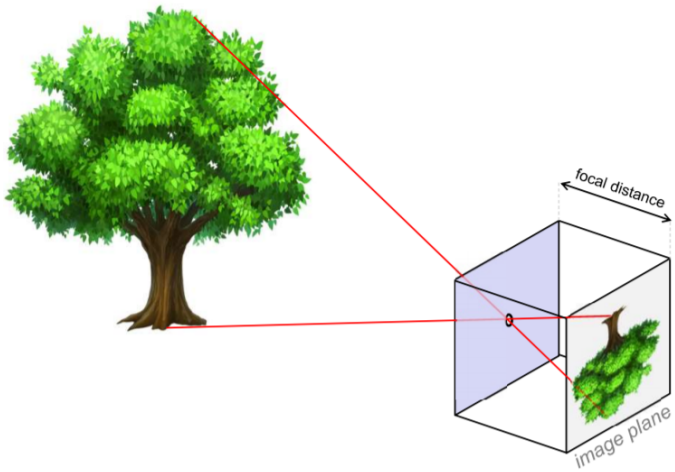
H3 Pin-hole camera



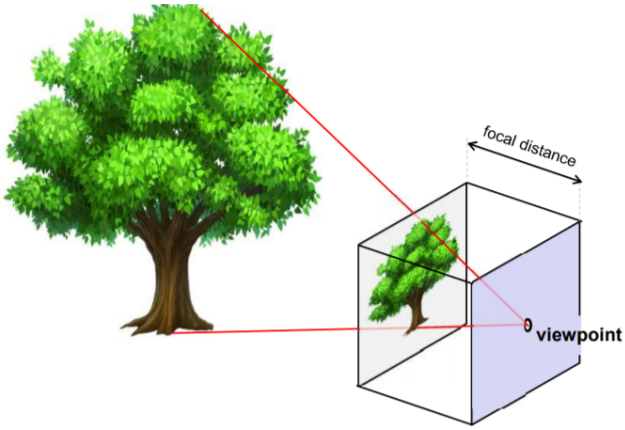
Quello che manteniamo nel modello come si vede in figura è:

- Il POV , cioè il punto da cui la luce, e quindi le immagini, entrano
- Il piano focale o piano immagine , dove l'immagine viene catturata
- la distanza focale , cioè la distanza tra il POV e il piano immagine, che determina l'angolo di campo (*FoV*) e la grandezza dell'immagine

Vediamo un esempio:



per semplificare ancora di più il modello, e renderlo simile al funzionamento del ray tracing, possiamo usare la seguente variante:



La pin-hole camera cattura la luce che passa per il POV, misurando quanta luce arriva da ogni **direzione**. In altre parole, a ogni pixel `p[i,j]` del piano immagine viene assegnata la quantità di luce che è passata per la direzione `POV - p[i,j]` (raggi rossi).

L'algoritmo ***ray casting***, ha proprio quest'ultimo modello come ispirazione per il suo funzionamento.

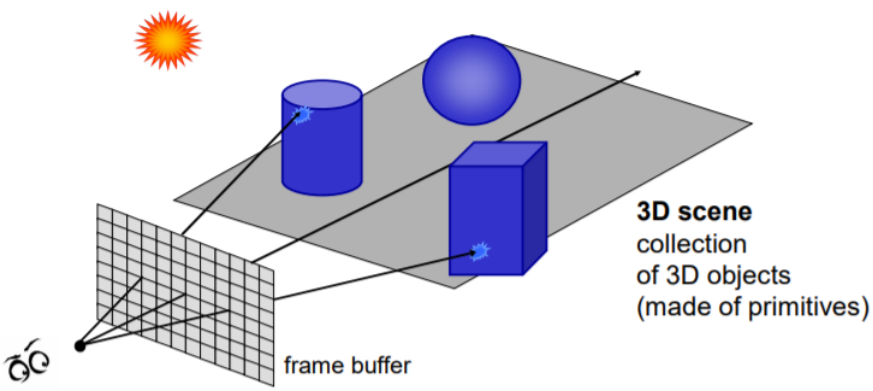
H2 Ray Casting

Per ogni pixel sullo schermo:

1. costruisco un raggio (detto *raggio primario* di quel pixel)
2. trovo le sue intersezioni con gli oggetti (le *primitive*) della scena
3. scelgo l'intersezione più vicina al POV
4. assegno al pixel il colore in base a tale intersezione

In pseudocodice:

```
FOREACH pixel p{
    make a ray r = POV - p
    FOREACH primitive o in scene
        find intersect(r, o)
    oj = closest intersection
    find color of oj at p
}
```



Il cuore dell'algoritmo è quindi il calcolare l'intersezione **raggio-primitiva**. Il costo dell'algoritmo sarà dato dal numero di pixel $\times$ numero di primitive in scena.

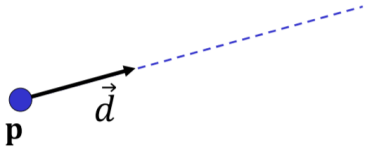
H3 Intersezione raggio-primitiva

Per fare ray-tracing, dobbiamo essere essere in grado di calcolare l'intersezione fra un raggio e una primitiva. Vi sono quindi algoritmi di ray-tracing per le mesh, per le superfici implicite, per le superfici parametriche etc., in quanto il calcolo dell'intersezione cambia in base al tipo di primitive. Facciamo un esempio di intersezione di un raggio con una sfera rappresentata come superficie implicita.

Possiamo modellare un raggio che parte dal POV come un vettore unitario $\vec{d}$ che parte dal punto p. Ogni punto sul raggio sarà definibile come:

$$p + k\vec{d} \tag{1}$$

dove $k \geq 0$ è uno scalare.

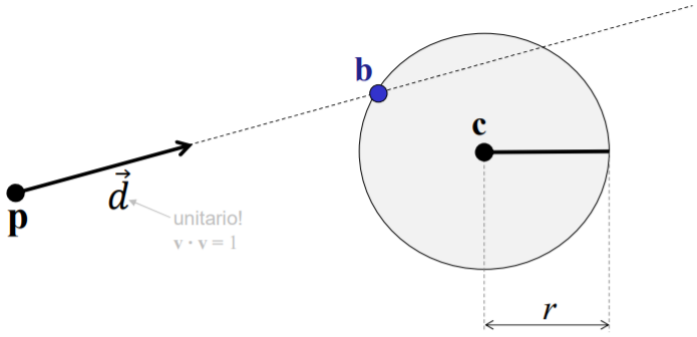


la sfera come superficie implicita sarà definita con la funzione

$$f(q) = ||q - c||^2 - r^2 \tag{2}$$

dove i punti q che appartengono alla superficie della sfera sono quelli per cui

$$f(q) = 0$$



quindi verificare che un punto del raggio interseca la sfera, significa verificare se

$$\exists k \geq 0 \mid \quad f(p + k\vec{d}) = 0$$

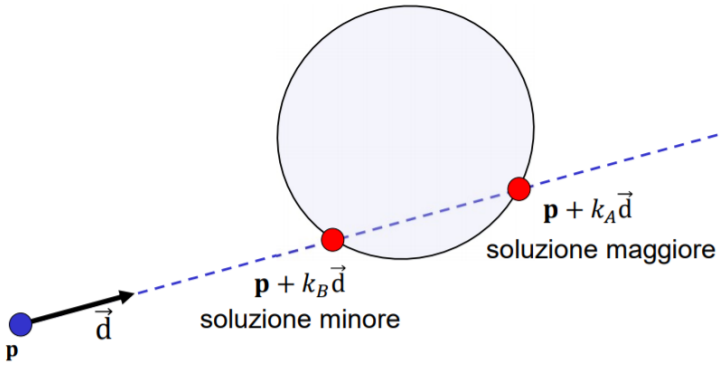
sostituendo il generico punto q nella (2) con (1):

$$
\begin{gathered}
||p + k\vec{d} - c||^2 - r^2 = 0 \\
\leftrightarrow \\
(p + k\vec{d} - c) \cdot (p + k\vec{d} - c) - r^2 = 0 \\
\leftrightarrow \\
((p - c) + k\vec{d}) \cdot ((p - c) + k\vec{d}) - r^2 = 0 \\
\leftrightarrow \\
k^2 \underbrace{||\vec{d}||^2}_{a} + k \underbrace{2(p - c) \cdot \vec{d}}_{b} + \underbrace{||p - c||^2 - r^2}_{c} = 0
\end{gathered}
$$

quindi abbiamo un'equazione di secondo grado in k:

$$
k_{A,B} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
$$

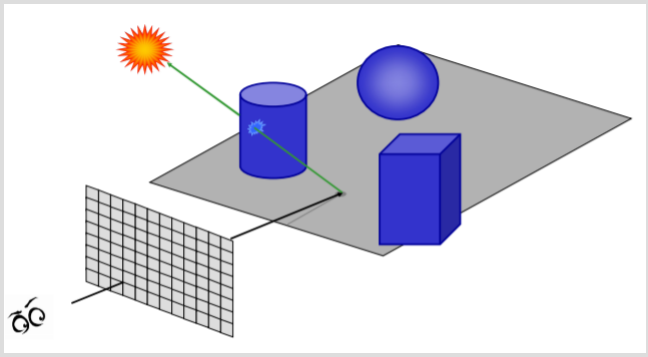
ma di cui prenderemo la soluzione più vicina, cioè quella negativa, in modo da ottenere l'intersezione più vicina al POV:



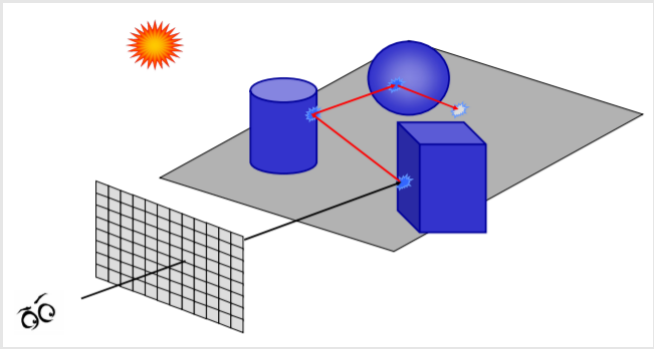
Ovviamente questo è il caso in cui esistono 2 soluzioni, ovvero quando il $\delta \geq 0$, quindi avremo 2 intersezioni dell'oggetto. Nel caso in cui $\delta = 0$, avremo un'unica intersezione con la primitiva (praticamente il raggio "sfiora" la superficie), mentre se $\delta < 0$, non ci saranno intersezioni fra il raggio e la primitiva presi in considerazione.

H2 Ray Tracing

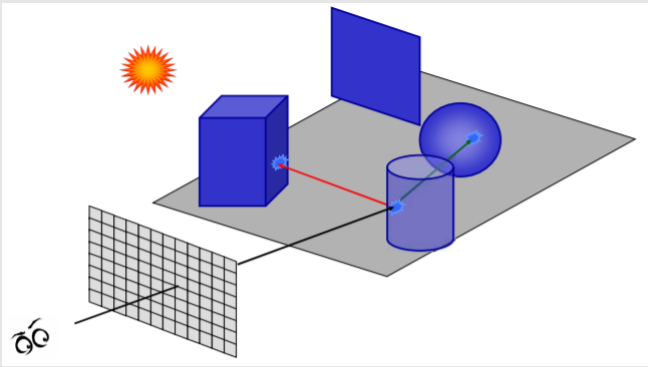
Il ray casting è il caso specifico in cui si considerano solo le intersezioni con le primitive per trovare il colore da assegnare al pixel. Nel caso generico in cui si percorrono i raggi per calcolare tali colori, si parla di **ray tracing**. Oltre al semplice ray casting, alcuni algoritmi di ray tracing più sofisticati possono calcolare ulteriori intersezioni dei raggi per ottenere un risultato più realistico rispetto all'illuminazione della scena. Ad esempio è possibile calcolare se un oggetto (ad esempio il pavimento) è in ombra, oppure se una primitiva riflette il colore di un'altra parte della scena, oppure ancora se un oggetto semitrasparente rifrange la luce.



Calcolo dell'ombra: il raggio primario interseca un punto nel pavimento, e per capire se quel punto è in ombra o meno si cerca un eventuale oggetto che si frappone fra lui e la fonte luminosa con un raggio secondario



Calcolo della riflessione: il raggio primario interseca un punto nella primitiva più vicina, da lì se l'oggetto è riflettente calcolo con dei raggio secondari da dove era riflessa luce



Calcolo della rifrazione in un oggetto semitrasparente: una parte del raggio primario viene riflessa, una parte rifratta

In generale, per produrre raggi secondari è necessario trovare prima il punto di intersezione come nel ray casting, e da lì far partire uno o più raggi secondari e vedere cosa e se intersecano altri elementi della scena.

```
FOREACH pixel p{
    make a ray r = POV - p
    WHILE{
        FOREACH primitive o in scene
            find intersect(r, o)
        oj = closest intersection
        r = new_bounce(oj);
    }
    find color of oj at p
}
```

Come si nota, per il calcolo di questi effetti è necessario calcolare più intersezioni per lo stesso pixel, aumentando il costo dell'algoritmo, ma con un risultato più fotorealistico.

H4 Prestazioni

In generale un algoritmo di ray tracing che fa uso di raggi secondari, avrà costo pari a

$$O(N \times M \times K)$$

dove:

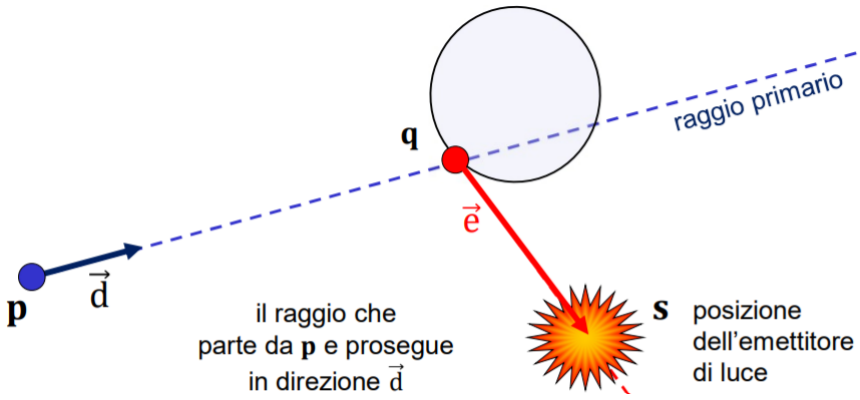
- N : è il numero di pixel dello schermo
- M : è il numero di primitive della scena
- K : è il numero di raggi per pixel (1 primario e $K-1$ secondari)

Il costo è abbastanza elevato, e per questo motivo è stato soprattutto in passato impiegato soprattutto nel rendering offline. Le prestazioni però sono altamente migliorabili poiché l'algoritmo è altamente parallelizzabile a livello di GPU, anche se questo sono finora state specializzate per il rendering *rasterization-based*.

H3 Approfondimento sul calcolo di ombre, riflessi e rifrazioni

Vediamo più dettagliatamente i piccoli problemi matematici legati al calcolo di questi effetti visivi

H4 Ombre



Definendo q come

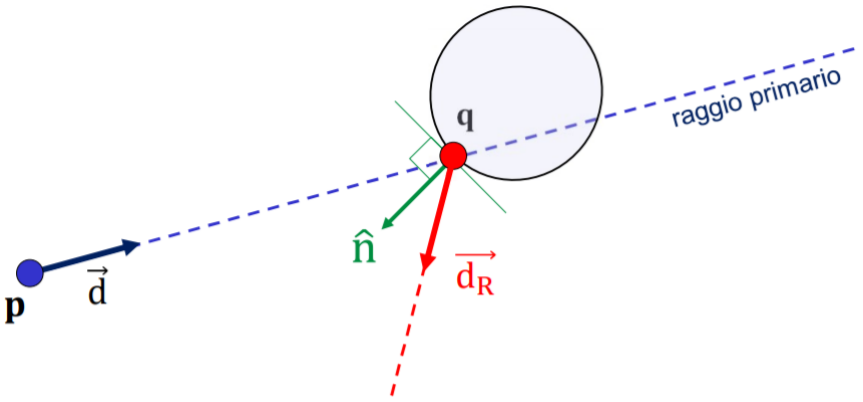
$$q = p + k\vec{d}$$

per un certo k, il raggio secondario verrà proiettato da q alla fonte luminosa s, quindi sarà definito come il vettore

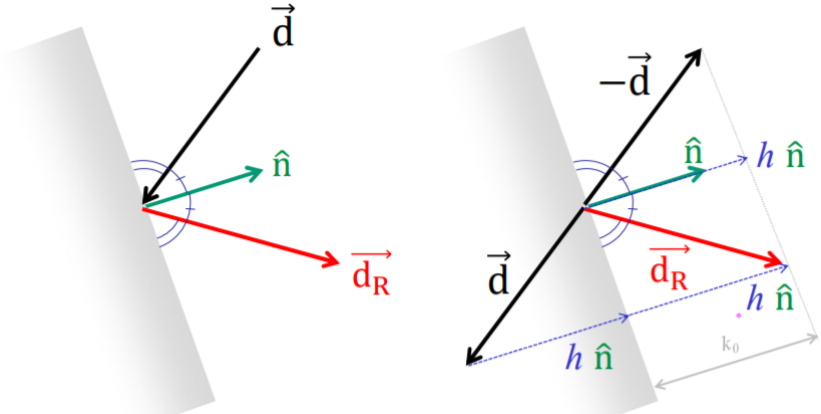
$$\vec{e} = s - q$$

Per capire se il punto q è in ombra, bisogna capire se esiste qualcosa che si frappone tra s e q, in pratica bisogna cioè stabilire se il vettore $\vec{e}$ interseca qualche altra primitiva o meno.

H4 Riflessi



In questo caso, il raggio secondario $\overrightarrow{d_R}$ viene riflesso dalla superficie con normale $\hat{n}$, quindi dobbiamo calcolare il vettore $\overrightarrow{d_R}$.



Dalla figura, si può dedurre che

$$\overrightarrow{d_R} = \vec{d} + 2h\hat{n}$$

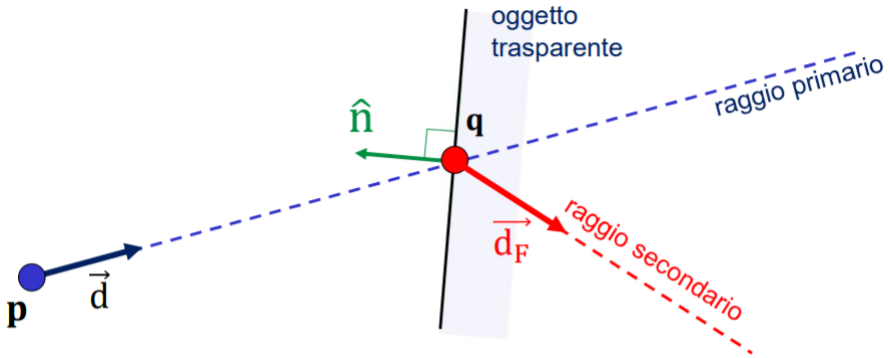
dove

$$h = -\vec{d} \cdot \hat{n}$$

in quanto prodotto dot che calcola la proiezione di $-\vec{d}$ su $\hat{n}$, che come si vede in figura è proprio h.

Una volta calcolato $\overrightarrow{d_R}$, bisogna verificare se tale raggio secondario intersechi o meno un'altra primitiva. In tal caso, il colore nel punto q sarà influenzato dal riflesso di qualcos'altro nella scena.

H4 Rifrazioni



In questo caso, dal punto q dobbiamo proiettare il raggio *rifranto* dalla primitiva, in quanto questa è trasparente. Per trovare $\overrightarrow{d_F}$ sarà necessario l'*indice di rifrazione* del materiale della primitiva.