

Lezione 7 27/03/2025

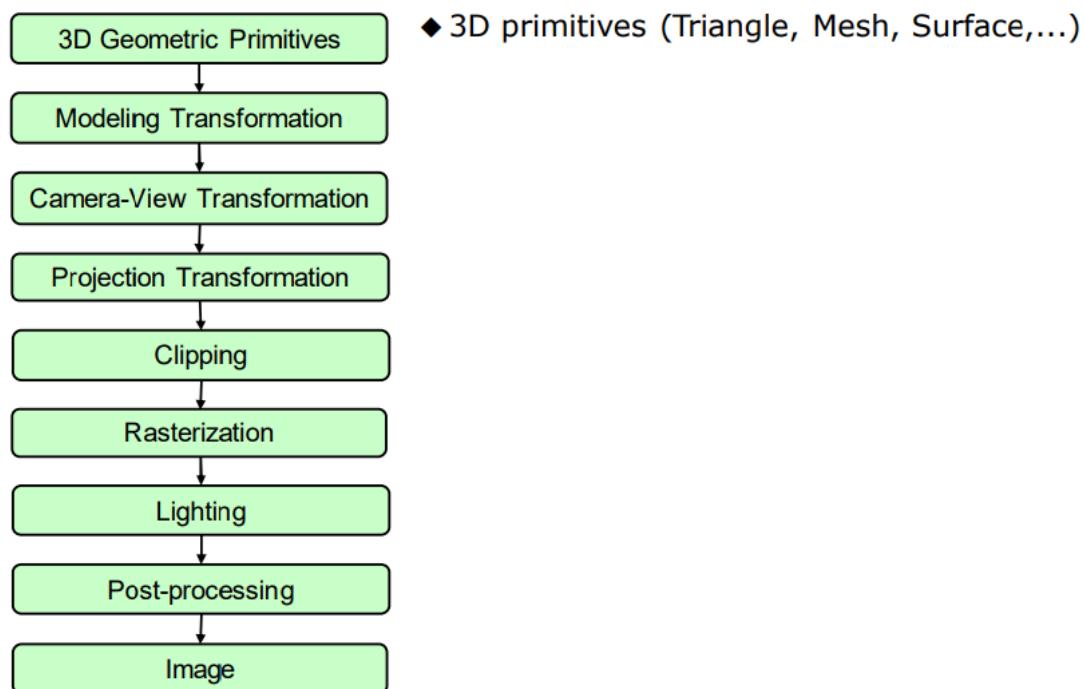
Pipeline di rendering

La **pipeline di rendering** definisce una sequenza di operazioni che devono essere seguite per passare da un'informazione di input, che è la descrizione di una scena 3D, arrivando ad avere a schermo un'immagine digitale che rappresenta la scena. Si chiama pipeline perché sono **operazioni sequenziali**, indipendenti, che hanno un input e output che passano ai blocchi successivi.

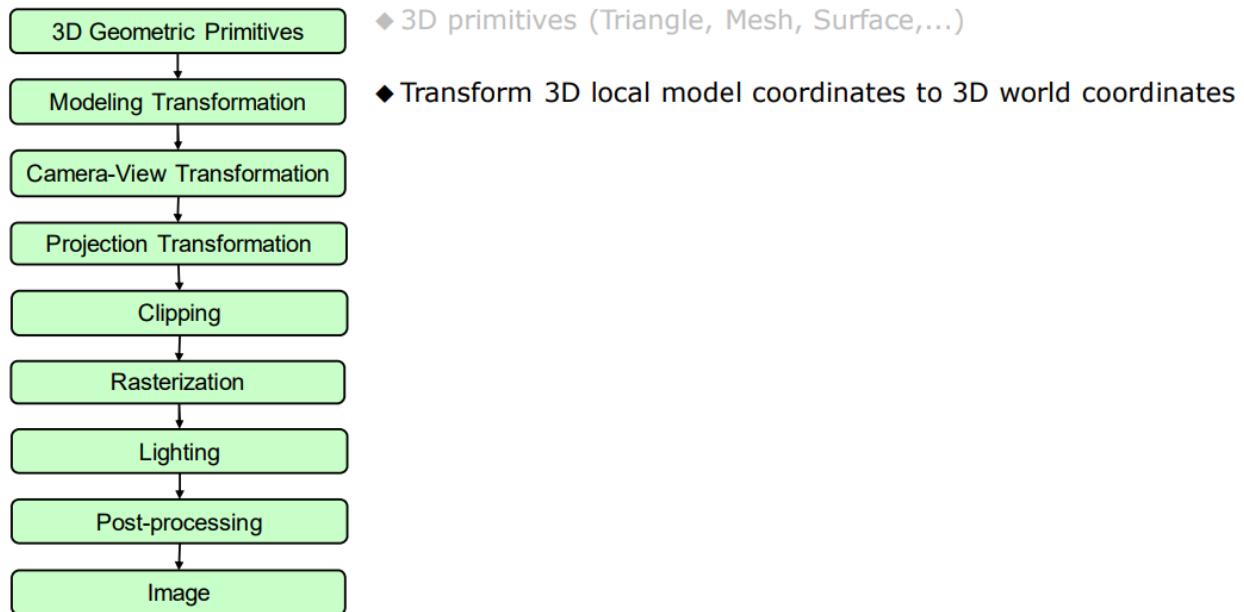
Bisogna definire qual è il formalismo più opportuno per descrivere una scena 3D. Nel nostro caso, la descrizione è una **sequenza di vertici** definiti nello spazio 3D. Questi vertici tipicamente presi 3 a 3 definiscono un **triangolo**, e triangoli connessi presi in sequenza definiscono parti della mesh. La **mesh** è il reticolo che avvolge un oggetto e ne definisce la **geometria**.

Dati questi punti nello spazio che descrivono discretamente e approssimativamente la forma dell'oggetto, noi vogliamo definirne anche l'**apparenza**. Dobbiamo descrivere i materiali o le proprietà visuali.

Quindi si passa da un mondo 3D con vertici, ad un'immagine 2D di pixel.

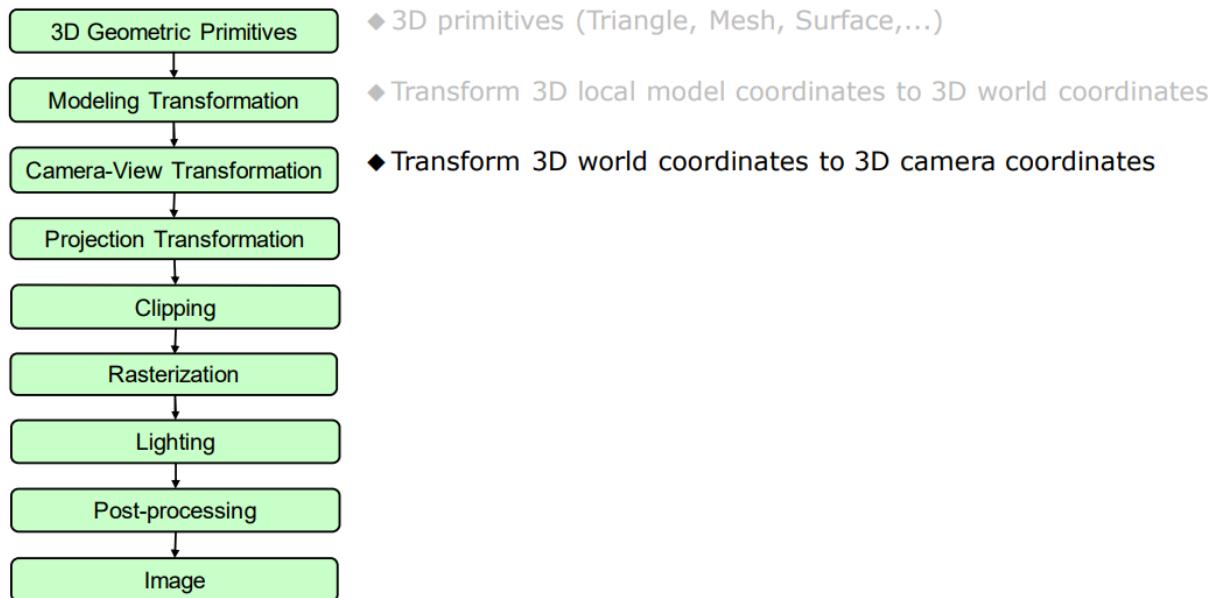


Le primitive per noi sono i vertici, che definiscono triangoli, che definiscono un modello di qualche entità. Quindi la forma 3D di un oggetto è modellata tramite le mesh, definendo le coordinate dei punti, però quando dobbiamo costruire la scena, andiamo a prendere questo oggetto e dobbiamo ambientarlo nella scena, magari ci servono anche più istanze.

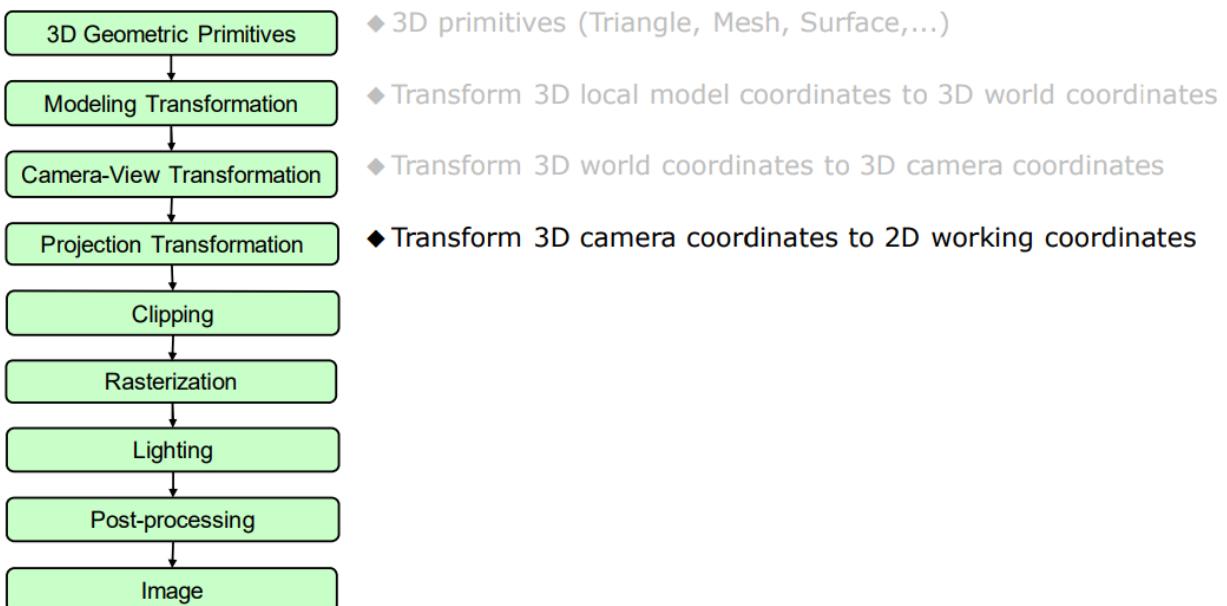


per fare questo è necessario prendere le coordinate spaziali dei vertici nel sistema di riferimento del modello, e modificarle per portarle nel sistema di riferimento della scena 3D. Questo significa che banalmente trasformo i valori con una trasformazione geometrica.

Quindi **modeling transformation** significa che prendo i vertici che descrivono l'oggetto 3D, e li sottopongo a delle trasformazioni geometriche, che sono operazioni matriciali, in nuove coordinate del mondo 3D.

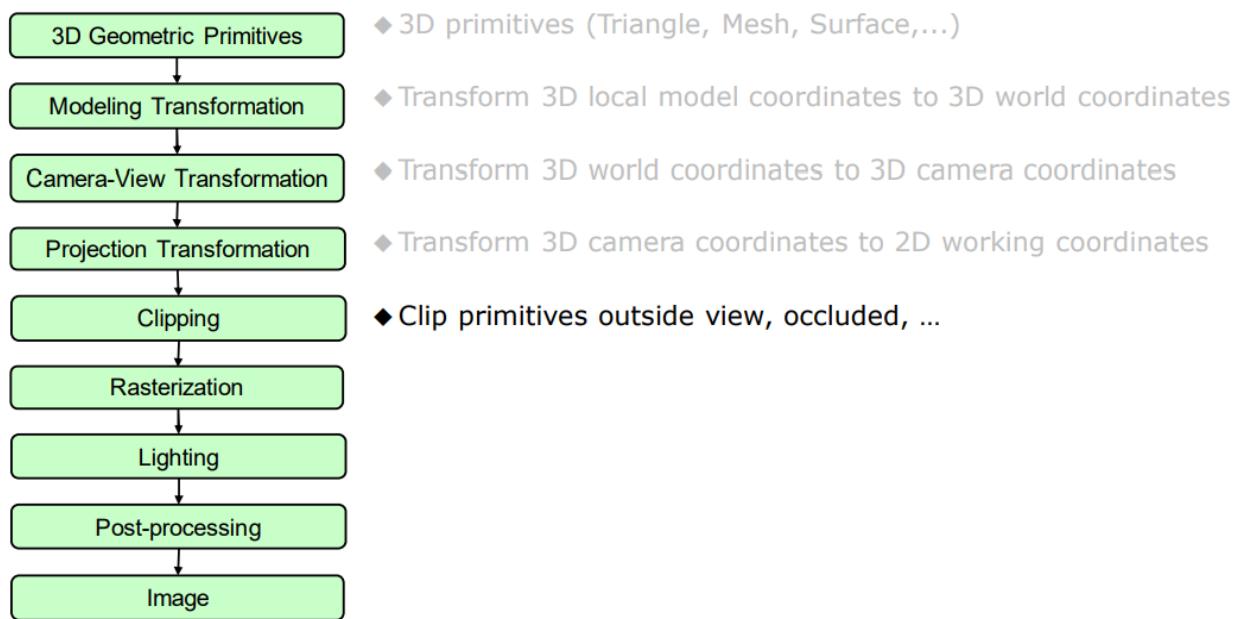


Una volta che ho deciso come mettere gli oggetti nel mondo, dobbiamo definire una **camera virtuale**, che inquadra un pezzo della scena, riduce l'area di interesse. Quindi piazziamo nel mondo un oggetto virtuale che inquadra il mondo da una certa direzione, perchè noi vogliamo vedere la parte del mondo inquadrato dalla camera. Questa è un'altra trasformazione geometrica perchè dobbiamo trasformare le coordinate degli oggetti dal sistema mondo al sistema camera.

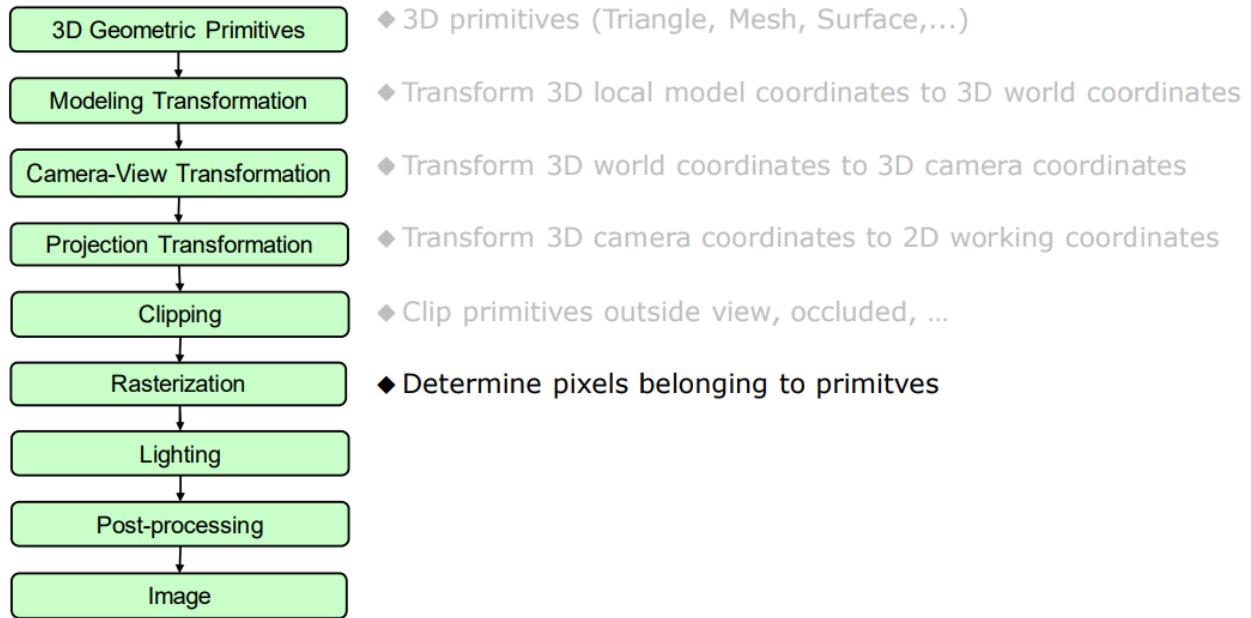


La camera, oltre a restringere la regione di interesse, fa anche la **trasformazione di proiezione** perchè la camera prende le coordinate nel mondo 3D e deve proiettarle su un piano immagine 2D.

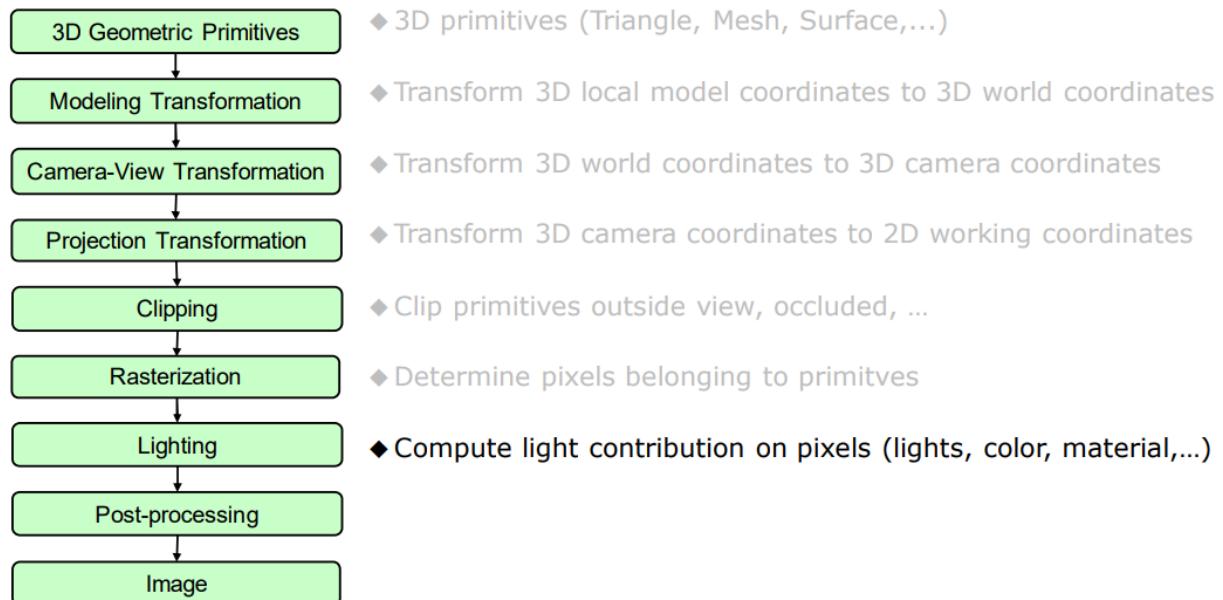
Inoltre la trasformazione di proiezione serve anche per realizzare la **prospettiva**, dal punto di vista della camera oggetti lontani sono più piccoli.



Tutte queste trasformazioni sono fatte su tutti i vertici della scena 3D (o quasi, non è vero), però solo una parte della scena è inquadrata dalla camera, quindi tutte quegli oggetti che cascano fuori dal punto di vista della camera non saranno mai disegnati a schermo. È necessario, per evitare calcoli inutili, eliminarli dalla pipeline di rendering. Il **clipping** è la riduzione delle primitive della scena a solo quelle utili, che poi verranno disegnate a schermo.



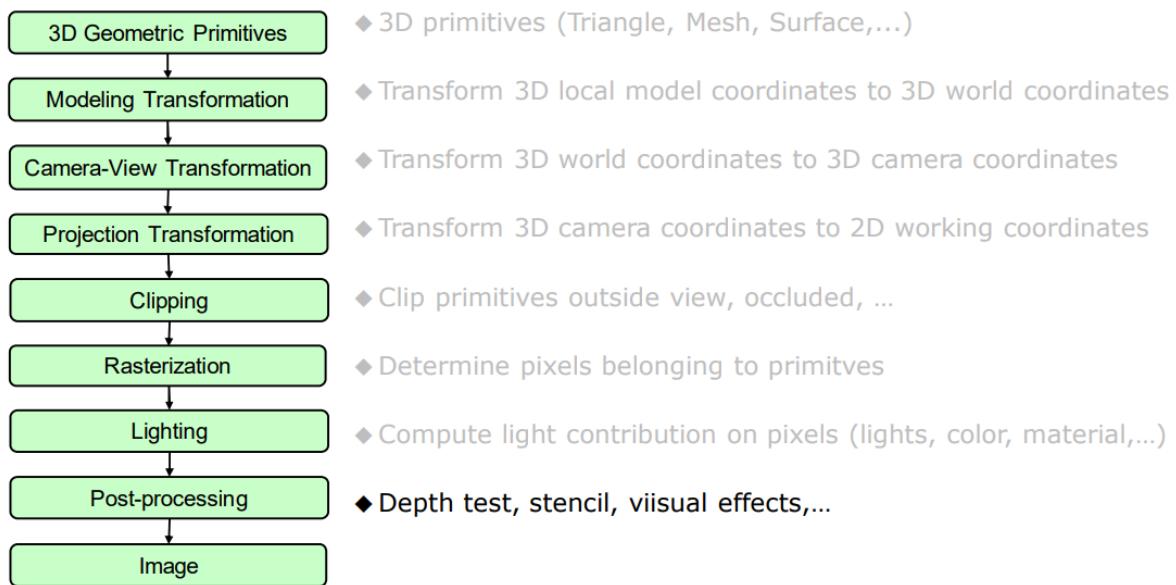
La **rasterization** si occupa di trovare effettivamente primitiva per primitiva, triangolo per triangolo, dove deve essere disegnato a schermo.



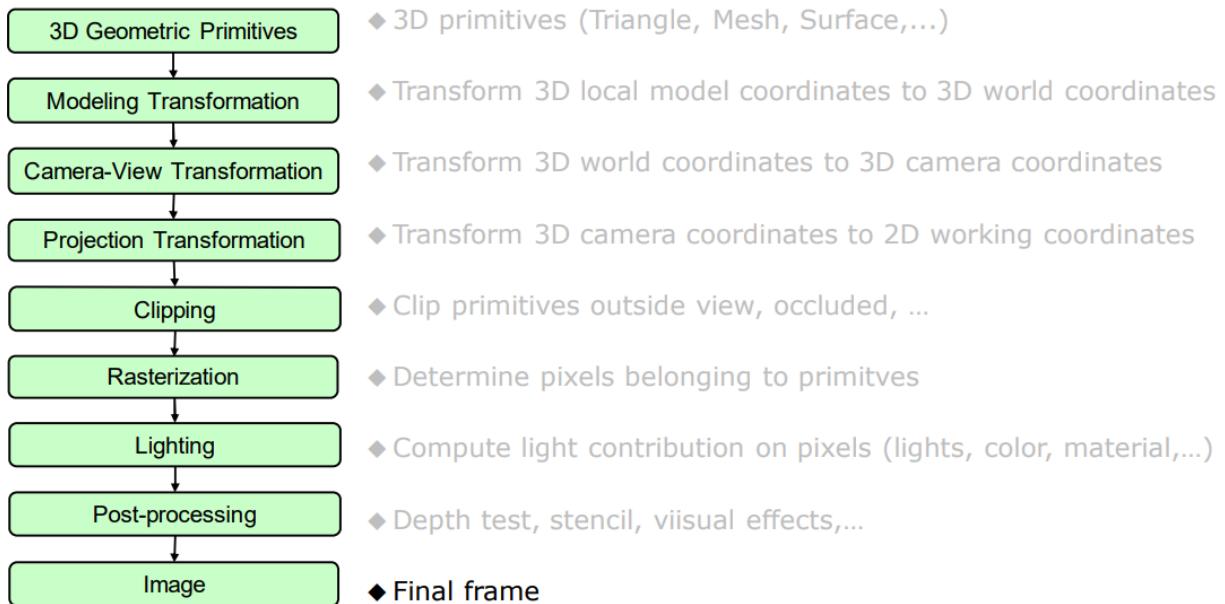
Per fare questo ciascuna posizione deve essere associata ad un **colore**.

Quindi questo blocco prende le posizioni dei pixel che devono essere colorate, recupera tutti gli attributi associati a quel triangolo che è stato associato ad una qualche superficie e magari anche a qualche tipo di materiale, e decide quel

singolo pixel, sotto quelle condizioni e con quegli attributi, che valori RGB finali avrà.



Se necessario poi verrà applicato del post processing, per esempio blending rispetto ai pixel nell'intorno, oppure se quella regione di schermo va sovrascritta dalla HUD del gioco (stencil).

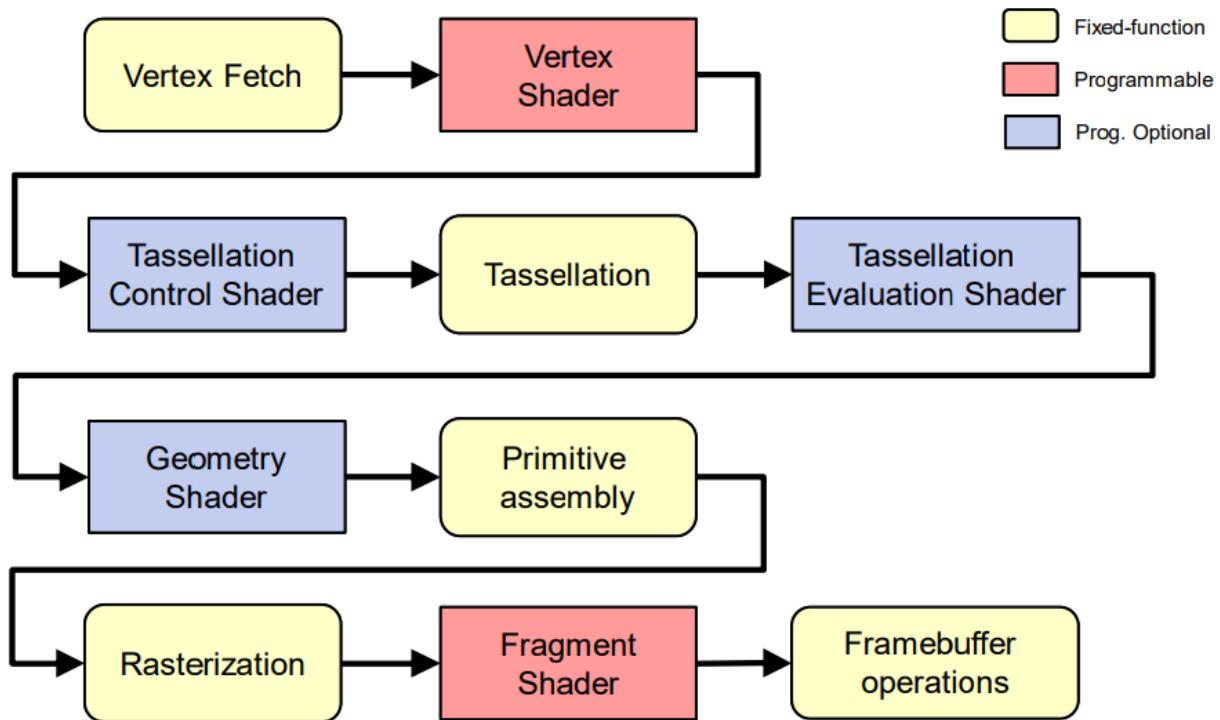


Alla fine di tutto questo si ha l'immagine a schermo fatta di pixel, che rappresenta la scena 3D.

Pipeline di rendering di OpenGL

è più basso livello, ma fa le stesse cose.

Si dice che è una pipeline di rendering programmabile (come anche per DirectX), dato che alcuni passi della pipeline sono standard, immutabili, mentre altri sono liberamente definibili.

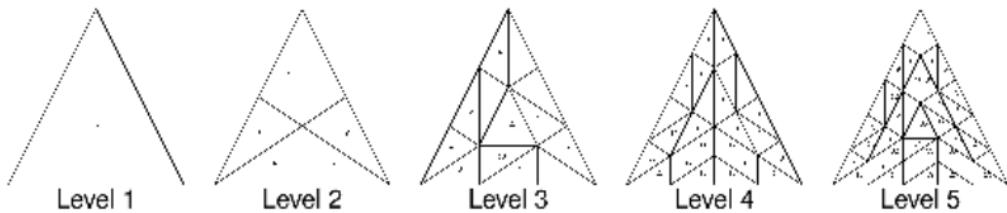


Questa pipeline ha il primo blocco, fisso, **vertex fetch**, che è l'input di tutta la pipeline di rendering. Prende i dati che noi abbiamo spedito sulla scheda grafica, e vertice per vertice spedisce delle informazioni al resto della pipeline di rendering.

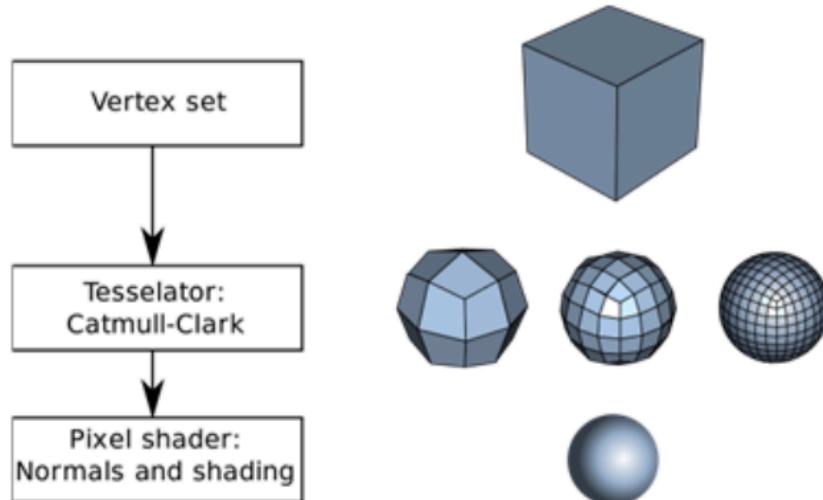
Poi abbiamo **vertex shader**, programmabile. Questo è l'insieme del modeling trasformation, camera-view transformation e projection transformation. è responsabile di applicare tutte le trasformazioni geometriche alle coordinate spaziali dei vertici di input. L'output è una lista di pixel. Quindi vede un singolo vertice (ricevuto dal vertex fetch) e ci applica tutte le trasformazioni geometriche che servono per portarlo dalle coordinate del sistema modello alle coordinate schermo, passando per il mondo e la camera. Questo blocco è un micro programma che deve manipolare questi attributi che riceve in input, però non tutto

deve essere sequenziale, infatti il vertex fetch spedisce gli attributi di input ad una serie di vertex shader che girano in parallelo.

Poi abbiamo i 3 blocchi di **tassellation**, opzionali. L'obiettivo è quello di andare a prendere una primitiva (triangolo) e andare a decidere se va diviso in triangoli più piccoli, andando a generare nuovi vertici.



Questa cosa può essere fatta a livello di modellazione, che però aumenta di molto le informazioni da gestire, per questo motivo a volte è più conveniente avere in input un oggetto più semplice e delegare la pipeline ad aggiungere i triangoli più piccoli.



Per esempio in base alla distanza dell'utente dall'oggetto, avremo più o meno triangoli perchè non è utile mostrarli tutti.

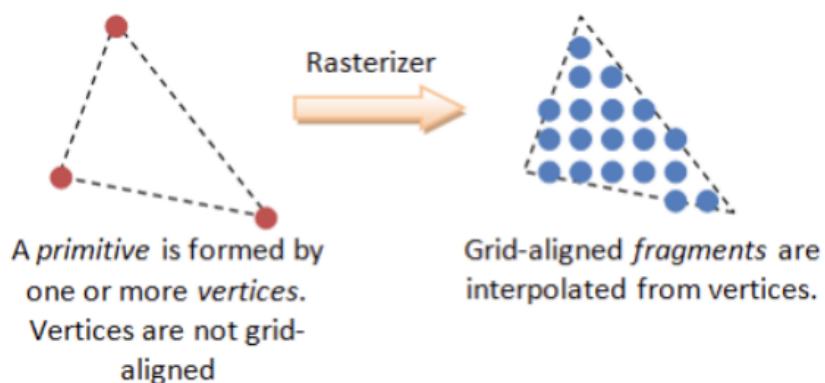
L'ultimo livello (tesselation evaluation shader) va a rimuovere quelli superflui.

Il **geometry shader** lavora a livello di primitive, si comporta in modo diverso dal tesselation anche se anche lui potrebbe aggiungere o rimuovere triangoli, permette

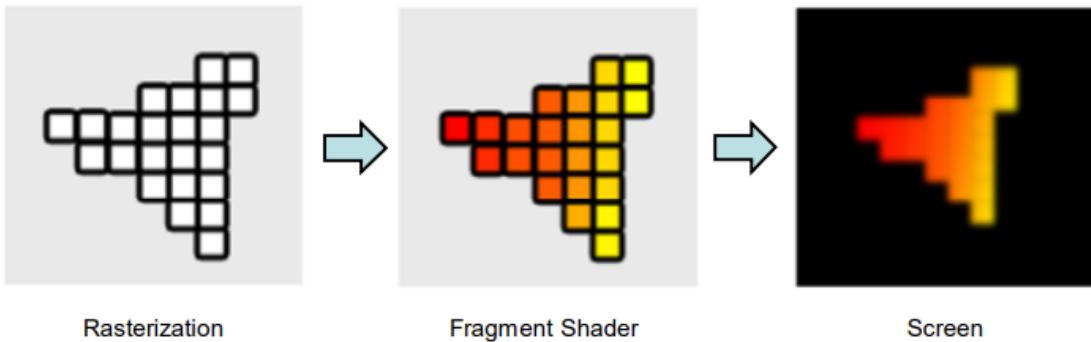
di definire delle logiche per modificare la geometria dell'oggetto.

Il **primitive assembly** è un blocco standard. Si occupa di prendere i vertici processati dai blocchi precedenti della pipeline e decidere quali vanno rimossi e per quale motivo. Applica le tecniche di clipping, per esempio rimuovendo tutto quello che c'è al di fuori della regione visibile, e quelle di culling, ovvero quello che è nascosto dalla camera. È possibile che una primitiva sia parzialmente dentro e parzialmente fuori la camera, in questo caso la trasformo in una nuova primitiva che vive dentro il campo di vista della camera.

Il **rasterizer** prende le coordinate spaziali dei vertici del triangolo che dobbiamo disegnare, e cerca di capire quali sono i pixel che dobbiamo colorare per dare l'impressione di avere un triangolo intero (e non solo i suoi vertici). Il rasterizer definisce cos'è la parte interna. Riceve 3 vertici e ne restituisce N.



Il **fragment shader** è programmabile ma non opzionale, è responsabile di prendere tutti gli attributi che sono stati passati all'origine della pipeline di rendering, tutte le informazioni del singolo pixel, mette tutto insieme e decide il colore che deve avere a schermo quel singolo pixel.

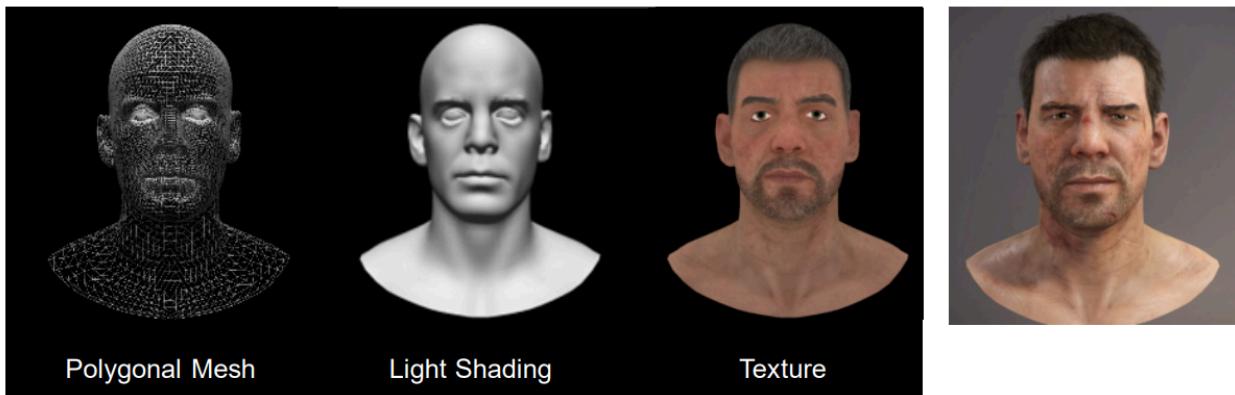


Fino a qua le informazioni usate sono le coordinate dei vertici. Tutte le informazioni aggiuntive devono arrivare al fragment shader.

Infine c'è il blocco di **framebuffer operations**, che potrebbe fare una serie di operazioni aggiuntive sull'immagine finale per cambiare la sua apparenza, per esempio modificare i colori perché bisogna rappresentare un evento di gioco (tipo il vignetting).

◆ A 3D model is composed of

- Geometry information (the vertexes, normals, triangles, ...)
- Material appearance (light interaction, texture, surface properties, ...)



Per fare tutto questo abbiamo bisogno del **modello dei materiali**, ovvero come descriviamo l'apparenza di una superficie, come descriviamo le caratteristiche del materiale che dovrebbe rivestire quella superficie 3D?

In computer grafica il modello del materiale viene definito in termini di **equazioni di illuminamento**, un modello materiale che prende certi input, mischiati insieme in

un certo modo, mi dicono punto per punto la superficie 3D che colore dovrebbe avere. Questi modelli devono essere velocemente computabili.

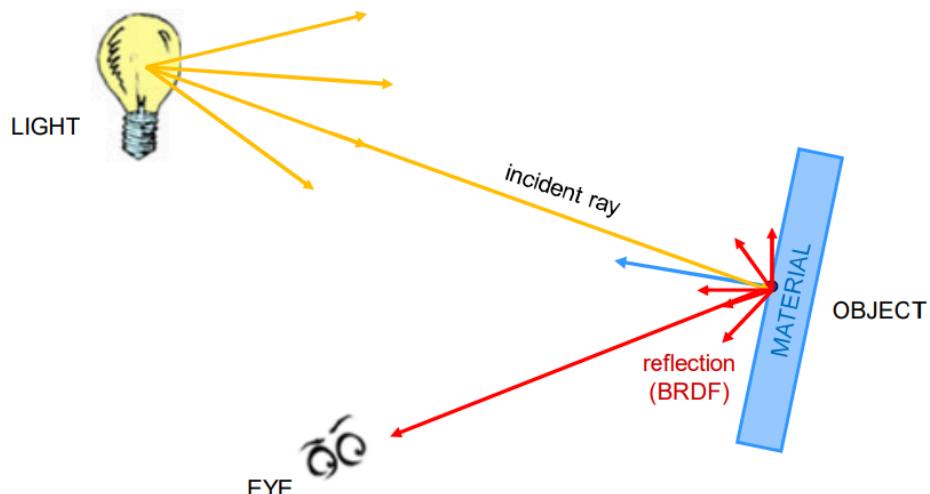
Quindi sono sostanzialmente delle funzioni. Dovrebbero essere applicati ad ogni singolo punto dell'immagine, ma tipicamente sono stati pensati per delle superfici monocromatiche, perché sono modelli approssimativi del comportamento della luce, non descrivono l'apparenza punto per punto ma assumono per esempio che tutto il materiale è nero, tutto il materiale è riflettente... però sappiamo che i materiali hanno variabilità che cambiano punto per punto.

Questi modelli di illuminamento vengono aiutati dalle **textures** per poter definire punto per punto le informazioni spaziali di un materiale.

Questi modelli sono delle equazioni i cui parametri possono essere diverse cose: scalari, immagini texture, altre equazioni...

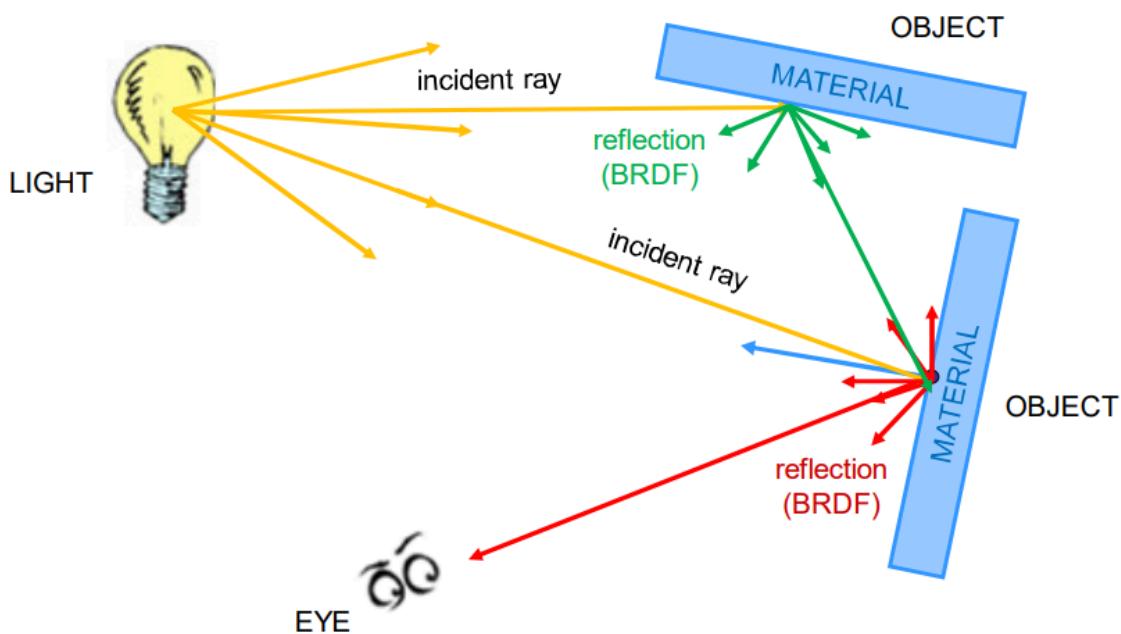
Un **modello di illuminamento** è una funzione matematica che prende dei valori di input e decide punto per punto che valori di colore dovrebbe avere. Esistono modelli di illuminamento locale o globale.

I **modelli locali** sono semplificati, prevedono di stimare l'apparenza di un punto della superficie andando a considerare come delle luci illuminano quel punto. Come parametri hanno quelle sorgenti di luce e la conformazione della geometria in quel punto preciso. Quindi usano informazioni locali al punto.

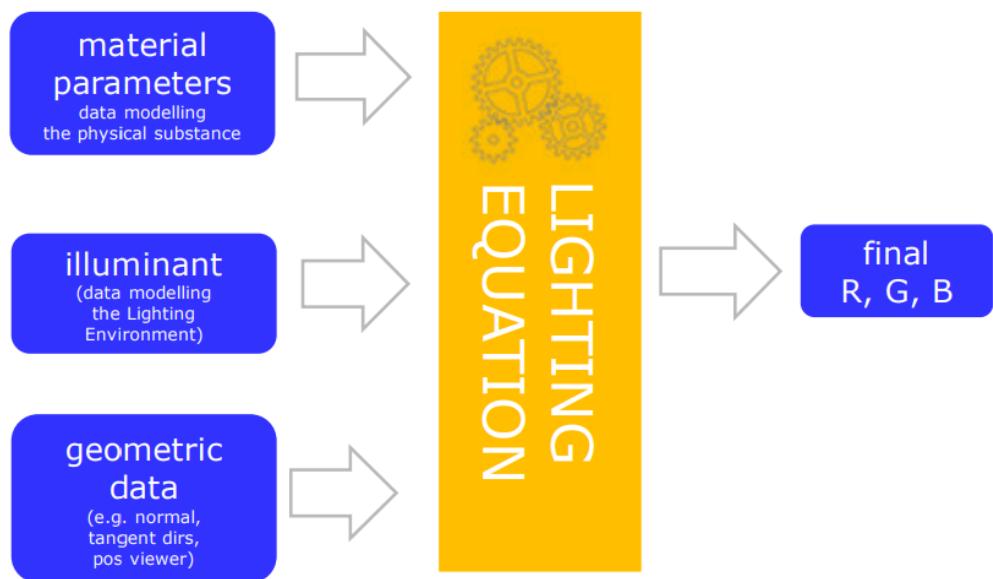


I **modelli globali** invece cercano di approssimare meglio il comportamento della luce reale. Il colore di un punto sulla superficie non è definibile solo rispetto alla

luce che riceve dalla sorgente, ma anche dalla luce di altri punti della scena, riflettendo i colori dell'oggetto colpito dalla luce, per esempio una tenda rossa riflette luce rossa, questa è luce indiretta, e questo succede in qualunque punto del mondo. Esistono tecniche come il ray tracing, è molto complicato, non è fattibile farlo in real time al 100%, i giochi real time si basano su modelli di illuminamento locale più o meno furbi, che non sono fisicamente realistici ma plausibilmente visibilmente realistici.



◆ Mathematical / algorithmical representation of the lighting model



Esistono diversi modelli di illuminamento locale che si possono usare, la scelta dipende dalla plausibilità dei risultati che vogliamo ottenere e dalla velocità di computazione.

Modello di illuminamento di Phong

è un modello di illuminamento a luce diretta, quindi locale. L'equazione è una semplice somma di 4 termini. La luce che vediamo in un certo punto è data dalla somma di:

- **Luce emissiva:** se la superficie ha una sorgente di luce
- **Luce ambientale**
- **Luce diffusiva**
- **Luce speculare**

$$I_{total} = I_{emiss} + I_{amb} + I_{diff} + I_{spec}$$

Queste equazioni si riferiscono solo al contributo colore (RGB), la componente A (alpha blending) viene gestita a parte.

◆ Emissive component

- Light emitted by the surface
- Is a purely additive component and depends on the object's material
- Few materials are emissive

$$I_{emiss} = k_{emiss} = \begin{pmatrix} e_r \\ e_g \\ e_b \end{pmatrix}$$



La componente emissiva è una terna di valori RGB, la luce che emette, che si somma al resto.

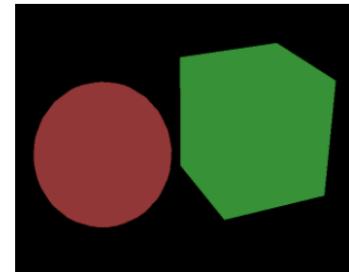
Di solito se si ha questa componente non si hanno le altre 3.

◆ Ambient Component

- Is an omni-directional light.
- Empirically models the contribution of indirect illumination of bouncing lights in the scene.
- It does not depend on the surface geometry.
- It depends on the material only

$$I_{amb} = k_{amb} \cdot I_A = \begin{pmatrix} a_r \\ a_g \\ a_b \end{pmatrix} \cdot \begin{pmatrix} A_r \\ A_g \\ A_b \end{pmatrix}$$

Ambient Light
Ambient component
(material dependent)



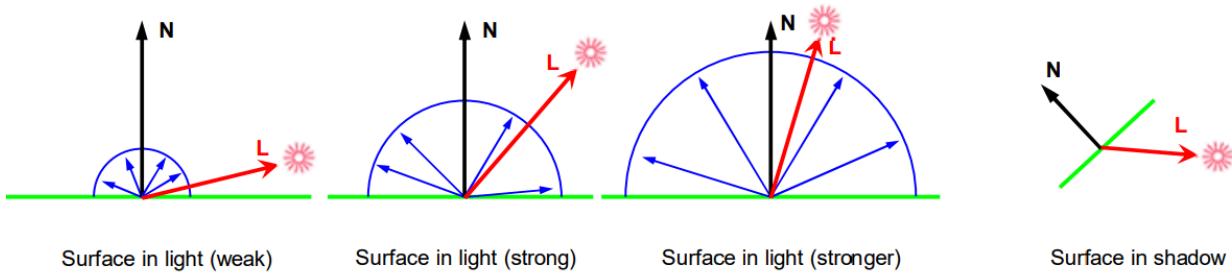
Ambient Light Only

Questo è un modo per simulare i contributi della luce degli altri oggetti, questa quantità è una sorta di luce di base che tutte le superfici hanno. Quindi non potendo calcolare la luce riflessa che le vari superfici generano nella scena, metto una quantità base data dai vari riflessi potenziali (luce bianca).

Non c'è nessuna percezione della geometria, stiamo vedendo solamente la forma dei bordi dell'oggetto. Quella rossa è una sfera ma potrebbe essere un cerchio. Nell'equazione non c'è nessuna informazione della geometria dell'oggetto.

◆ Diffusive Component

- The amount of reflected light with respect to the incident light and surface's normal angle
- The incident light has a direction
- The light is reflected equally in all directions (Lambertian surface)



L'informazione geometrica viene data dalla componente diffusiva.

La quantità di luce che arriva sulla superficie e viene riflessa dipende dalla direzione di luce e da come questa direzione di luce è allineata alla normale della

superficie.

La linea verde è il piano, è un punto della superficie. N è la normale, perpendicolare alla superficie. Il vettore rosso è la direzione di luce, ovvero dove è orientata la sorgente rispetto alla superficie. Più la luce è allineata alla normale della superficie e più la quantità di luce che viene riflessa aumenta.

La legge lambertiana (per le superfici lambertiane) dice che una superficie diffusiva, se viene illuminata, riflette la luce in egual modo in tutte le direzioni

◆ Diffusive Component

$$I_{diff} = k_{diff} \cdot I_{LD} \cdot \cos\theta = k_{diff} \cdot I_{LD} \cdot (\hat{L} * \hat{N})$$

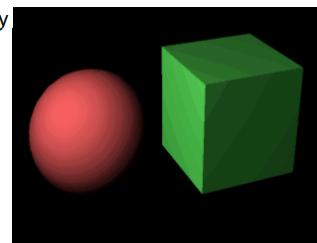
Diffusive component
(material dependent)

Angle between L and N

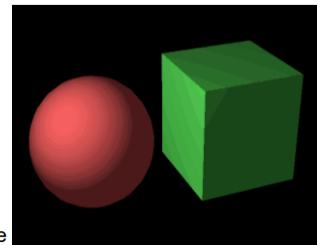
Incident Light

* Dot product

Diffusive Light Only



Ambient + Diffusive

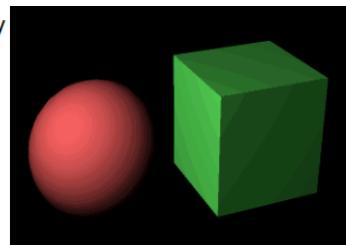


◆ Diffusive Component

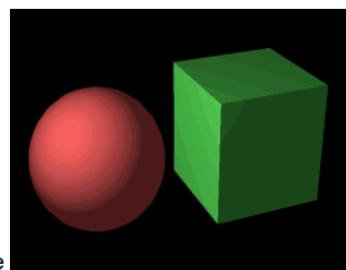
$$I_{diff} = \begin{pmatrix} d_r \\ d_g \\ d_b \end{pmatrix} \cdot \begin{pmatrix} L_r \\ L_g \\ L_b \end{pmatrix} \cdot (\hat{L} * \hat{N})$$

* Dot product

Diffusive Light Only

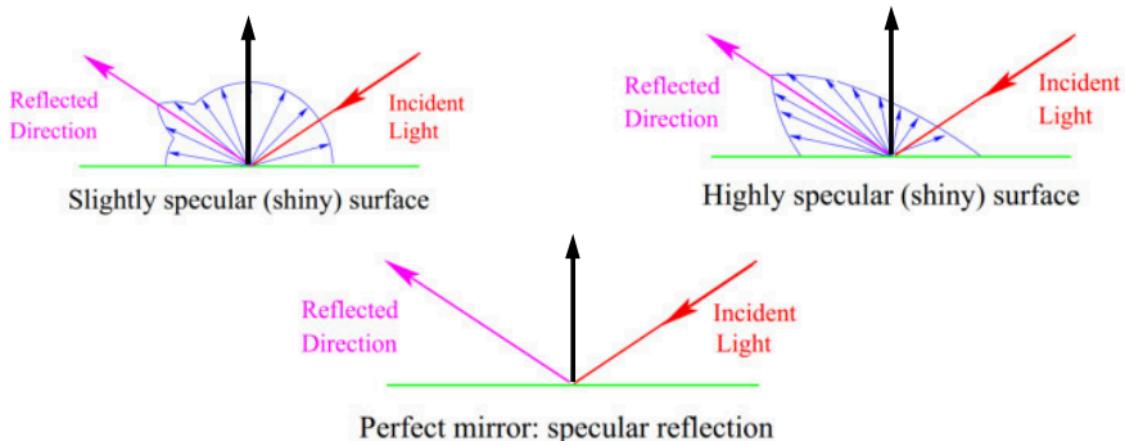


Ambient + Diffusive



Infine, la **componente speculare** cerca di modellare materiali lucidi, che se vengono illuminati da una certa luce e li guardi da una certa direzione, noti dei

riflessi. A seconda del tipo di materiale, la luce viene riflessa in modi diversi.



Questo comportamento riflettente è modellato dall'equazione:

◆ Specular Component

$$I_{spec} = k_{spec} \cdot I_{LS} \cdot \cos\alpha = k_{spec} \cdot I_{LS} \cdot (\hat{V} * \hat{R})^n$$

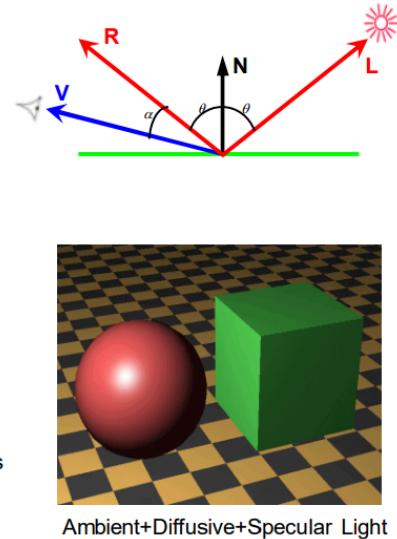
Specular component (material dependent)

Angle between V and R

View direction

Reflection direction

* Dot product



L'intensità della componente di luce speculare per i materiali riflettenti è data dal prodotto di un coefficiente k di luce speculare per l'intensità della luce incidente per il coseno di alpha (angolo tra la direzione di riflessione della luce e la direzione di vista, quindi dipende da dove guardiamo la superficie).

Questa è l'equazione di phong totale:

◆ The Phong lighting equation

- Given a SINGLE light source (L)

$$I_{total} = \underbrace{k_{emiss}}_{\text{Emissive}} + \underbrace{k_{amb} \cdot I_A}_{\text{Ambient}} + \underbrace{k_{diff} \cdot I_{LD} \cdot (\hat{L} * \hat{N})}_{\text{Diffusive}} + \underbrace{k_{spec} \cdot I_{LS} \cdot (\hat{H} * \hat{N})^n}_{\text{Specular (Phong)}}$$

Property of the Material

Property of the Light

Property of the Geometry

* Dot product

$$I_{total} = \underbrace{\left(\begin{matrix} e_r \\ e_g \\ e_b \end{matrix} \right)}_{\text{Emissive}} + \underbrace{\left(\begin{matrix} a_r \\ a_g \\ a_b \end{matrix} \right) \cdot \left(\begin{matrix} A_r \\ A_g \\ A_b \end{matrix} \right)}_{\text{Add only once}} + \underbrace{\left(\begin{matrix} d_r \\ d_g \\ d_b \end{matrix} \right) \cdot \left(\begin{matrix} L_r \\ L_g \\ L_b \end{matrix} \right)}_{\text{Add for every light source}} \cdot (\hat{L} * \hat{N}) + \underbrace{\left(\begin{matrix} S_r \\ S_g \\ S_b \end{matrix} \right) \cdot \left(\begin{matrix} L_r \\ L_g \\ L_b \end{matrix} \right) \cdot (\hat{N} * \hat{H})^n}_{\text{Specular (Phong)}}$$

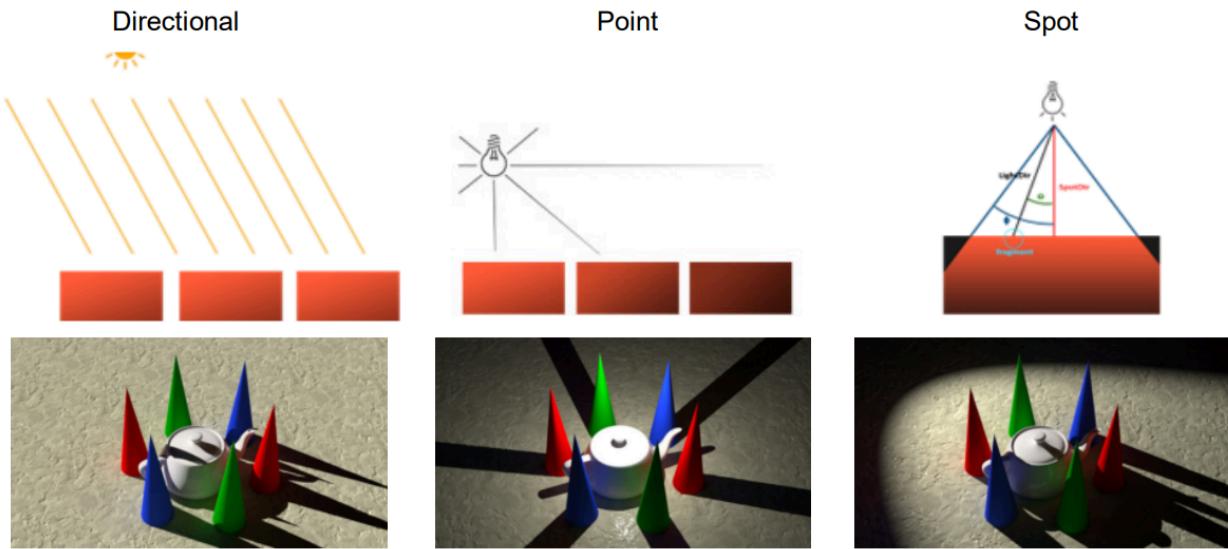
Property of the Material

Property of the Light

Property of the Geometry

* Dot product

Le luci non sono tutte uguali, a seconda del tipo si comportano in modi diversi.



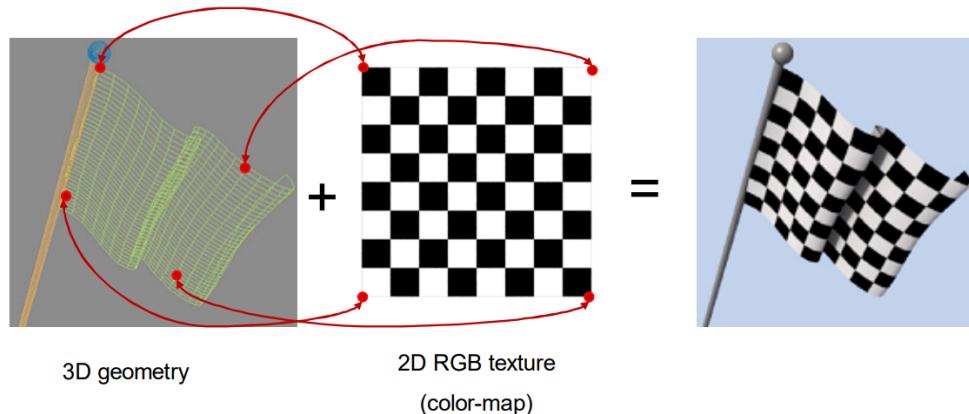
Esistono altri modelli di luce, nelle slides.

Nei game engine si va a definire un modello che rappresenta le proprietà di un materiale in modo semplice. Non dobbiamo ragionare in termini di normali, geometrie, luci incidenti... si modella invece degli attributi percettivi, tipo quanto è riflettente... gli diamo delle misure, degli scalari. Tipo quanto è metallico tra 0 e 1, quanto è opaco/ruvido da 0 a 1.

Tutto il lavoro sporco viene fatto dal game engine, dal modello che non vediamo. Noi settiamo delle proprietà percettive.

◆ **Texture:** N-Dimesional data structure containing some kind of information

- Texture coordinates are associated to geometry coordinates
- E.g. Colors of a surface

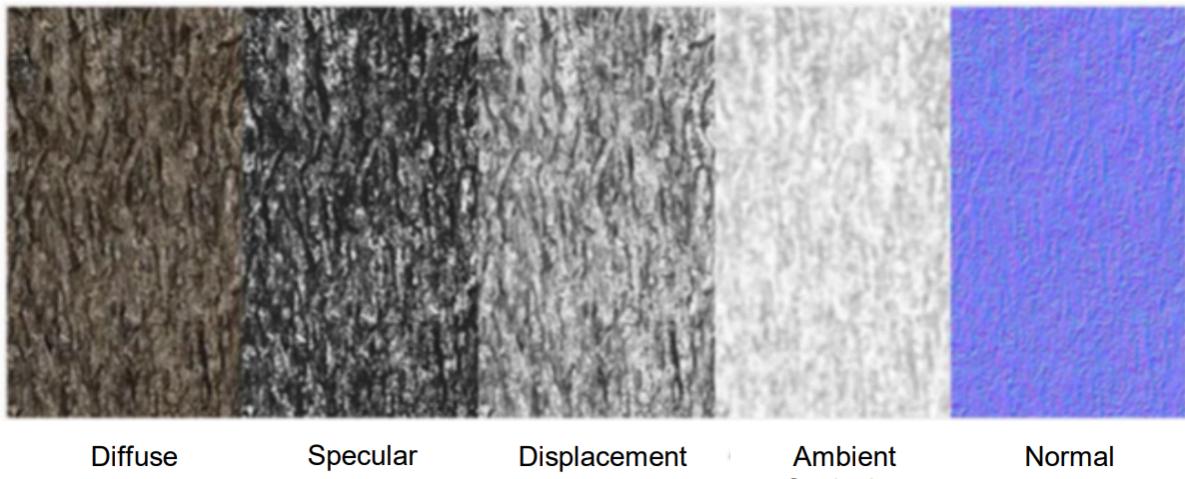


Il colore diffusivo è il colore del materiale.

Come modelliamo variazioni come le venature del legno? con le **texture**. Sono delle immagini digitali che ci dicono punto per punto che colore dovrebbe avere quella regione.

Spesso, si definiscono tramite texture anche altre parti dell'equazione.

- ◆ Parameters for the lighting equation taken from the different textures



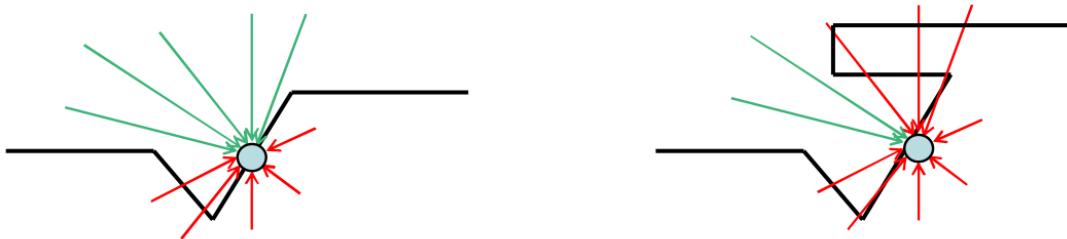
Manca quindi una parte di effetti di luce, come **l'ambient occlusion**, ovvero quell'effetto di luce quando parte della luce viene occlusa da parti della scena.

- ◆ Adds realism by taking into account attenuation of light due to surface occlusions



Fin ora abbiamo visto la pipeline di rendering forward, o diretta. Poi c'è il **deferred rendering**.

L'ambient occlusion si può andare ad applicare un modello di luce globale (ma non si può fare in real time) oppure usando un trucco. Concettualmente, dovrei capire per ogni punto della scena quanta luce arriva.



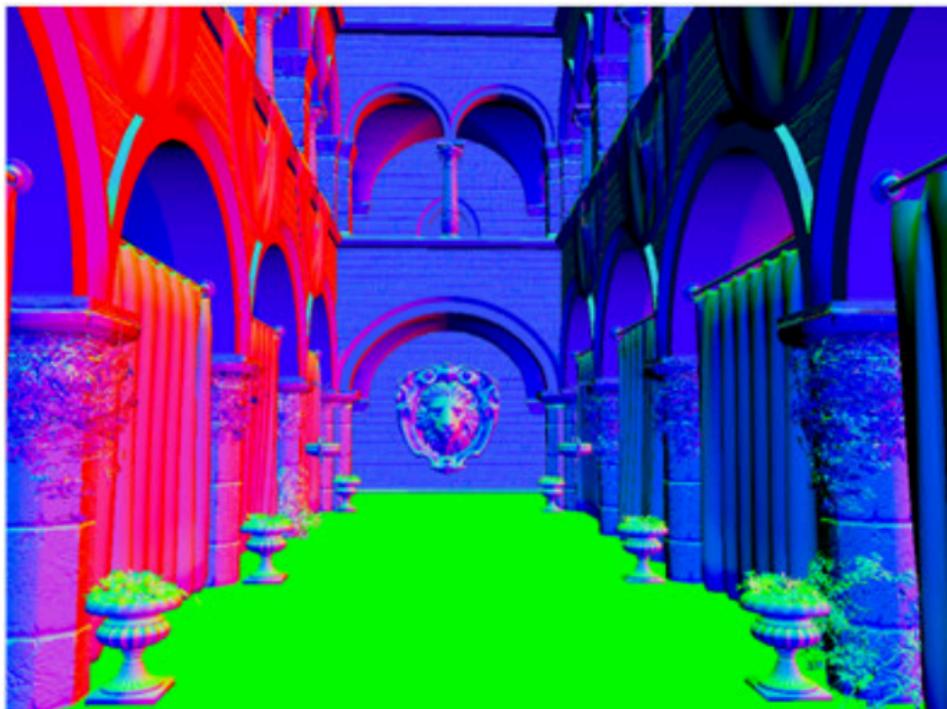
Però posso fare questo solo se conosco l'intera geometria della scena, ma la pipeline di rendering normale lavora sui triangoli.

C'è un trucco, ovvero il **deferred rendering**. Significa fare tutta la pipeline di rendering senza il rendering finale dei pixel. Il vantaggio è che strada facendo, siamo in grado di calcolare diverse informazioni sull'intera immagine, per esempio alla fine riusciamo ad avere l'informazione pixel per pixel della distanza dalla camera, quindi punto per punto sappiamo la depth dell'immagine. Questo è un esempio: i punti più chiari sono quelli più lontani.



Depth, surface's distance from the viewer (scaled for visualization)

Un'altra informazione che viene raccolta sono le normali.



Normals (coded in RGB for visualization)

Queste informazioni sono generate naturalmente, e possono essere viste come texture.

Quindi facciamo di nuovo la pipeline di rendering, una volta per generarle queste informazioni punto per punto e l'altro giro per utilizzarli come informazione aggiuntiva.

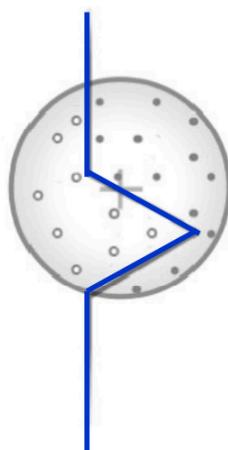
Queste informazioni possono essere usate per calcolare l'ambient occlusion. Non ci serve sapere tutta la geometria, ci basta la depth delle superfici.

- ◆ Screen Space Ambient Occlusion – SSAO

- Used by Crytek in *Crysis* (2007)

- ◆ Idea

- No ray casting
 - Depth gives a rough approximation about scene geometry
 - Depth map can be sampled in a sphere around the point
 - If z inside object -> occlusion
 - Normal helps retaining useful samples



Si ottiene questa informazione, che è quella che ci serviva:

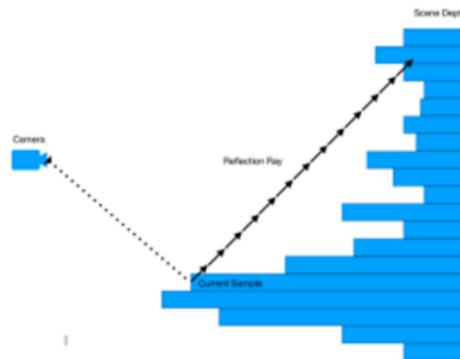


Computed Screen Space Ambient Occlusion (SSAO)

Poi ci sono le **riflessioni**, che si possono gestire in un modo simile.

- ◆ General Idea

- ◆ For every pixel of reflective surface
 - Cast ray from camera to reflective surface
 - Compute reflected ray at the surface (normal map)
 - Detect intersection with geometry (depth map)
 - Obtain intersection point appearance (color map)
 - Color the original pixel with the obtained color



- ◆ View coordinates need to be transformed in screen coordinates

Ho la camera che inquadra la depth, ho un raggio di vista che passa dalla camera alla superficie e suppongo che venga riflesso in una certa direzione, quindi il colore si decide in base al colore della superficie attuale e quella riflessa.

Questi sono effetti applicati sui pixel della scena prendendo informazioni geometriche calcolate durante il deferred rendering, quindi sono "**screen space**". Il costo sono i due giri di rendering.