Grafica al calcolatore

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2025/2026

Indice

1 Introduzione

1.1 Interazione con l'utente

Un modo per far interagire l'utente con il programma è l'interfaccia grafica. L'interazione può essere ottenuta in vari modi, ad esempio:

- Finestre di dialogo
- Realtà virtuale
- Realtà aumentata
- Giochi

1.2 Sintesi e analisi

La sintesi è il processo di creazione di un'immagine a partire da una descrizione matematica, mentre l'analisi è il processo di estrazione di informazioni da un concetto già esistente.

2 Modeling

La modellazione 3D è un processo di **descrizione** di un oggetto o una scena al fine di poterla disegnare. La descrizione avviene attraverso:

- **Struttura**: Viene descritta dalla geometria degli oggetti e dalla loro posizione reciproca nello spazio tridimensionale
  - Definizione geometrica
  - Trasformazioni 3D
- **Apparenza**: Descrive come la superficie del modello interagisce con la luce (colori, riflessi, ecc...)
  - Definizione telecamere virtuali
  - Definizione sorgenti di luce
  - Definizione proprietà dei materiali

2.1 Definizione geometrica

Ci sono varie tecniche di modellazione:

- **Low poly diretta**, ad esempio con Wings3D. È la costruzione manuale di una mesh poligonale a bassa risoluzione, partendo anche da primitive già fatte.
- **Subdivision surfaces**, ad esempio con Blender. Si parte da una mesh poligonale a bassa risoluzione e si applicano algoritmi di suddivisione per ottenere superfici più lisce e dettagliate.
- **Digital sculpting**, ad esempio con ZBrush
- **Modellazione procedurale**, ad esempio con Houdini. Si utilizzano algoritmi e regole per generare automaticamente modelli 3D complessi, ad esempio generazione di terreni, vegetazione, edifici, ecc...

2.1.1 Mesh

Gli oggetti tridimensionali vengono codificati come una maglia (mesh) di triangoli. I triangoli vengono utilizzati perchè sono il poligono più semplice che può essere utilizzato per approssimare una qualsiasi superficie. Una mesh è composta da:

- **Vertici**: Punti nello spazio 3D
- **Facce**: Insiemi di vertici che formano triangoli

Definizione 2.1 (Definizione matematica di mesh)**.** Una mesh di triangoli è una discretizzazione lineare a tratti di una superficie continua (un "2-manifold") immersa in $\mathbb{R}^3$, cioè un oggetto bidimensionale che si trova in uno spazio tridimensionale. Le componenti sono:

- **Geometria**: i vertici, ciascuno con coordinate $(x, y, z) \in \mathbb{R}^3$
- **Topologia**: come sono connessi tra loro i vertici, nel caso di una tri-mesh ogni faccia è definita da un insieme di tre vertici

2.2 Trasformazioni 3D

Per posizionare e orientare gli oggetti nello spazio 3D, si utilizzano trasformazioni geometriche, che possono essere rappresentate tramite matrici. Le principali trasformazioni sono:

- **Traslazione**: Spostamento di un oggetto da una posizione a un'altra
- **Rotazione**: Rotazione di un oggetto attorno a un asse specifico
- **Scalatura**: Modifica delle dimensioni di un oggetto lungo gli assi X, Y e Z

2.3 Telecamere virtuali

Per visualizzare una scena 3D su uno schermo 2D, è necessario utilizzare una telecamera virtuale che definisce il punto di vista da cui viene osservata la scena. Il problema è che nel passaggio dal 2D al 3D c'è perdita di informazione.

Per definire una telecamera virtuale, sono necessari:

- **View point**: da dove si osserva
- **Look at point**: dove si guarda
- **View direction**: orientamento della telecamera
- **Regole di proiezione**:
  - Ortografica: mantiene le proporzioni reali degli oggetti
  - Prospettica: simula la visione umana, con oggetti più lontani che appaiono più piccoli

2.3.1 Proiezione

Il mondo non è infinito, quindi bisogna definire il **cono di vista** (frustum), che delimita la porzione di scena visibile dalla telecamera. Il frustum è definito dal parallelepipedo delimitato da due piani:

- **Near plane**: piano più vicino alla telecamera
- **Far plane**: piano più lontano dalla telecamera

Gli oggetti al di fuori del frustum non vengono proiettati (fase di clipping). La proiezione avviene su un piano di vista (view plane), che rappresenta lo schermo 2D.

2.4 Illuminazione

Tramite l'illuminazione si riesce a distinguere la forma degli oggetti tridimensionali. La modellazione delle luci della scena si occupa del loro posizionamento e del tipo di luce utilizzata. I tipi di luce più comuni sono:

- **Directional light**: luce proveniente da una direzione specifica, simula la luce solare
- **Point light**: luce che si propaga in tutte le direzioni da un punto specifico, simula una lampadina
- **Spotlight**: luce che si propaga in un cono da un punto specifico, simula un faro
- **Ambient light**: luce diffusa che illumina uniformemente tutta la scena, senza una direzione specifica

2.5 Proprietà dei materiali

Il materiale di cui è fatta la superficie di un oggetto condiziona il suo aspetto nel momento in cui viene colpito dalla luce. Le proprietà principali dei materiali sono:

- **Colore**
- **Riflettività**
- **Rugosità**

3 Rendering

Il rendering ha l'obiettivo di creare un'**immagine bidimensionale** a partire dalla descrizione di una scena tridimensionale. Ogni pixel dell'immagine deve avere un colore che dipende dalla geometria, dall'illuminazione e dalle proprietà dei materiali della scena. Per renderizzare una scena c'è bisogno di tradurre il processo fisico della luce in un algoritmo e questo ha bisogno di molte semplificazioni e approssimazioni.

3.1 Modello fisico dell'illuminazione

La luce è una radiazione elettromagnetica con lunghezza d'onda tra i 400 e i 700 nm che parte da una **sorgente** verso un **ricevente**. La sorgente può essere un **emittente** (sorgente primaria) oppure un **riflettore** (sorgente secondaria). La lunghezza d'onda determina il colore della luce percepita dall'occhio umano:

- Luce monocromatica: Quando è presente soltanto una lunghezza d'onda (es. laser)
- Luce policromatica: Quando sono presenti più lunghezze d'onda (es. luce solare)

L'energia trasportata dalla luce determina l'**intensità** luminosa.

3.1.1 Interazione della luce con i materiali

La luce può interagire con la materia in vari modi:

- Una sorgente di luce (luce **incidente**) illumina la superficie di un oggetto
- Una parte della luce **riflessa** da un punto si distribuisce uniformemente in tutte le direzioni (luce **diffusa**)
- Una parte della luce viene **riflessa** da un punto verso una direzione preferita (luce **speculare**)
- Una parte della luce viene assorbita all'interno del materiale (luce **trasmessa**)

Per generare l'immagine bisogna tenere conto della **quantità di luce** che viene trasportata fino all'osservatore (camera virtuale) interagendo con i vari materiali. Il pixel dell'immagine misura dunque la **radianza** o **luminanza** delle superfici visibili dalla camera virtuale.

- Fissata la camera virtuale e fissato un pixel si osserva una porzione di superficie di un oggetto della scena
- Il pixel prende un valore sulla base dello stato fotometrico di questa porzione di superficie.

3.2 Equazione del rendering

Per poter modellare al meglio i fenomeni fotometrici dei materiali viene definita la **Bidirectional Reflectance Distribution Function** (BRDF), che descrive come la luce viene riflessa da una superficie in funzione della direzione di arrivo e della direzione di uscita della luce:

$$f_r(x, \omega_i, \omega_o)$$

Dove:

- x è il punto della superficie
- ω_i è l'angolo della luce incidente
- ω_o è l'angolo della luce riflessa

3.2.1 Path tracing

Il path tracing è un algoritmo di rendering che simula il comportamento della luce nella scena tracciando il percorso dei raggi di luce. Al posto di calcolare quali raggi partendo dalla sorgente di luce colpiscono la camera, si parte dalla camera e si tracciano i raggi che da essa colpiscono la fonte di luce. Questo metodo è più efficiente perchè si concentra solo sui raggi che effettivamente contribuiscono all'immagine finale.

3.2.2 Algoritmi di lighting

Ci sono diversi metodi per calcolare l'illuminazione in una scena 3D:

- I metodi **locali** tengono conto solo dell'effetto delle sorgenti di luce dirette sulla superficie, senza considerare le interazioni tra le superfici. Esempi: **Flat shading**, **Gouraud shading**, **Phong shading**
- I metodi **globali** considerano tutte le interazioni della luce con le superfici della scena, inclusi riflessi, rifrazioni e ombre. Esempi: **Ray tracing**, **Radiosity**, **Path tracing**

3.2.3 Shading

Lo shading ha lo scopo di determinare il colore effettivo dei pixel a partire da un modello di illuminazione dato. Determina come e quando applicare il modello di illuminazione prescelto. Ci sono tre tipi principali di shading:

- **Flat shading**: A ogni primitiva geometrica (triangolo) è associato uno stesso colore uniforme. È il metodo più semplice e veloce, ma meno realistico.
- **Gouraud shading**: Il valore di illuminazione viene calcolato ai vertici e viene **interpolato** per i pixel (fragment) interni
- **Phong shading**: Vengono interpolate le normali ai vertici per ottenere le normali di ciascun pixel e il modello viene calcolato per ogni pixel. Produce risultati migliori e più realistici, ma è più costoso computazionalmente.

4 Animazione

L'animazione in 3D è il processo di creazione di modelli (o camera virtuale) in movimento all'interno di una scena tridimensionale.

4.1 Produzione del modello

Il modello viene costruito con una descrizione geometrica del suo aspetto e in base al tipo di **cinematica** adottata per l'animazione è possibile dotare il modello di una struttura scheletrica (**rigging**) che ne consente il movimento.

4.1.1 Rigging

È la fase di preparazione del modello per fare in modo che possa essere animato. Vengono inserite delle componenti aggiuntive al modello che permettono all'animazione di diventare **parametrica** (cioè dipendente da parametri variabili nel tempo). L'obiettivo è quello di definire i parametri che governano l'animazione in modo da inserire una sorta di **semantica** nel **movimento**.

L'animazione è la fase in cui i parametri definiti dal rigging vengono istanziati per generare i movimenti e le deformazioni desiderate.

4.1.2 Animazione del volto

Per animare un volto umano in modo realistico si può ricomporre il movimento del volto a partire dal controllo dei movimenti elementari della muscolatura. Questo viene fatto muovendo gruppi di punti di controllo o interpolando tra una forma di riferimento. Le tecniche più diffuse sono:

- **Blend based**: si costruiscono numerose versioni del modello (dette shape di espressioni di base). L'animazione deriva dalla combinazione di queste versioni.
- **Rig based**: si inseriscono delle "ossa" all'interno della faccia che controllano i movimenti e li propagano sulla superficie del volto.
- **Phisycally based**: vengono simulate le proprietà fisice ed elastiche dei tessuti molli della pelle e deu muscoli.

4.1.3 Animazione del corpo umano

L'animazione del corpo umano consiste nella creazione di un modello geometrico articolato, cioè un manichino. Per animare il manichino si devono definire:

- Parametri fisici della struttura articolata
- Tipologie standard di comportamento
- Interazione con l'ambiente circostante
- Aspetto del manichino

Un esempio di modello di struttura articolata è lo **stick figure**, cioè aste rigide collegate con giunti in grado di ruotare l'uno rispetto all'altro. Questo simula uno scheletro umano semplificato.

Le parti di una struttura articolata sono organizzate in una gerarchia ad albero, dove ogni nodo rappresenta una parte del corpo e le trasformazioni applicate a un nodo vengono **propagate** ai nodi figli.

4.1.4 Animazione automatica

Animare in 3D con metodi informatici significa attribuire variazioni ai valori assunti dai parametri che descrivono l'aspetto e lo stato di un oggetto nella scena 3D. Si crea qundi una funzione parametrica che varia nel tempo. Ci sono due approcci principali:

- **Traiettoria analitica**: Si definisce una funzione matematica che descrive il movimento dell'oggetto nel tempo. Ad esempio, una traiettoria circolare può essere descritta con funzioni trigonometriche.
- **Keyframing**: Si definiscono dei fotogrammi chiave (keyframe) che rappresentano stati specifici dell'oggetto in determinati istanti di tempo. Il software interpola automaticamente i valori tra i keyframe per creare un'animazione fluida. Il tipo di interpolazione può essere lineare, cubica, spline, ecc...

Un modo alternativo per definire delle traiettorie complesse è l'utilizzo di **motion capture**, che consiste nel registrare i movimenti di un attore reale e trasferirli a un modello 3D.

5 Graphics Pipeline

La graphics pipeline è il processo che trasforma una scena 3D in un'immagine 2D visualizzabile su uno schermo. Tra il "mondo" 3D e il "mondo" 2D bisogna introdurre delle trasformazioni intermedie per poter rappresentare correttamente la scena.

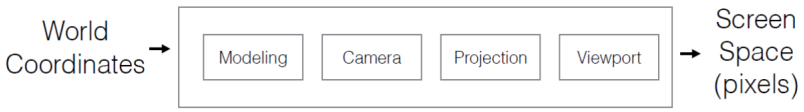

Figura 1: Trasformazione della scena 3D in immagine 2D

Per applicare queste trasformazioni bisogna avere un sistema di coordinate ben definito in ogni fase della pipeline:

- **Object coordinates**: coordinate locali dell'oggetto
- **World coordinates**: coordinate globali della scena
- **Camera coordinates**: coordinate relative alla telecamera virtuale
- **Screen coordinates**: coordinate 2D dell'immagine finale

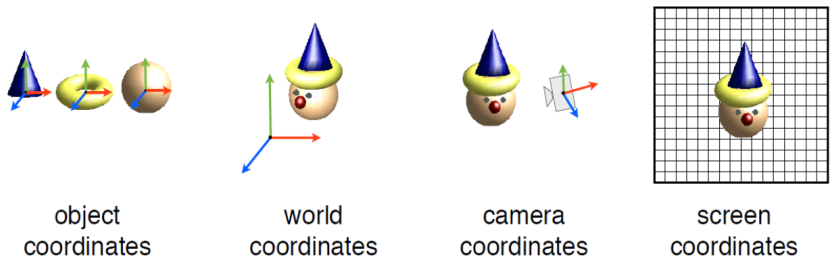

Figura 2: Sistemi di coordinate nella graphics pipeline

5.1 Trasformazioni

Per realizzare la pipeline grafica bisogna capire il processo che mappa le componenti della scena da uno spazio di coordinate ad un altro. Le trasformazioni principali sono:

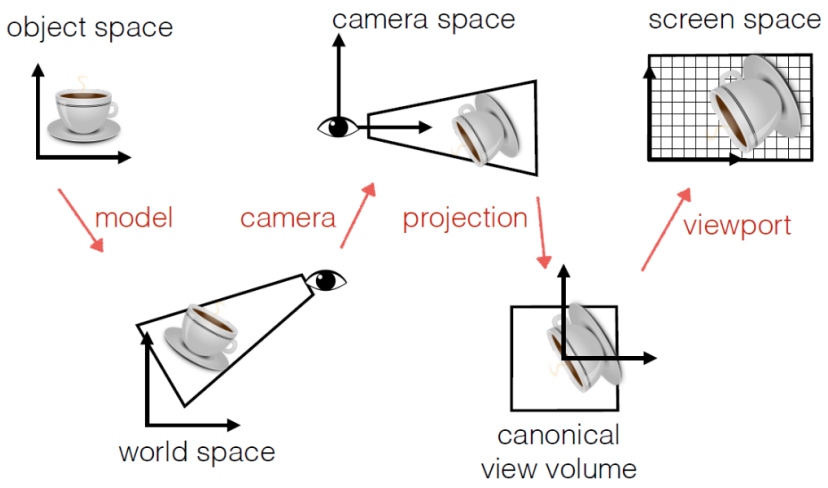

Figura 3: Trasformazioni nella graphics pipeline

5.1.1 Model transformation

La model transformation mappa le coordinate locali dell'oggetto (object coordinates) nelle coordinate globali della scena (world coordinates) in base al grafo della scena. Questa trasformazione tiene conto della posizione, dell'orientamento e della scala dell'oggetto nella scena.

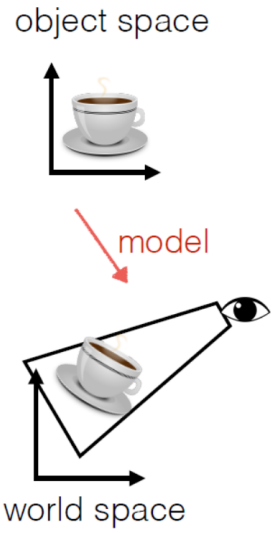

Figura 4: Model transformation

5.1.2 Camera transformation

La camera transformation mappa le coordinate globali della scena (world coordinates) nelle coordinate relative alla telecamera virtuale (camera coordinates). Questa trasformazione tiene conto della posizione e dell'orientamento della telecamera nella scena. Questa trasformazione dipende soltanto dalla posizione della telecamera.

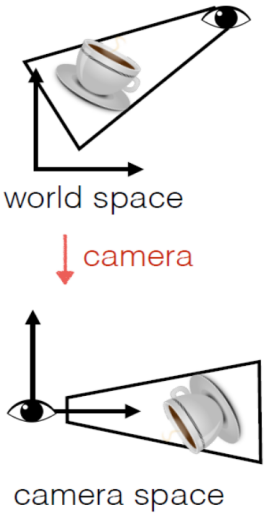

Figura 5: Camera transformation

5.1.3 Projection transformation

La projection transformation mappa le coordinate della telecamera (camera coordinates) nelle coordinate normalizzate del dispositivo (clip space), cioè uno spazio di coordinate 3D normalizzato in cui le coordinate X e Y sono comprese tra -1 e 1.

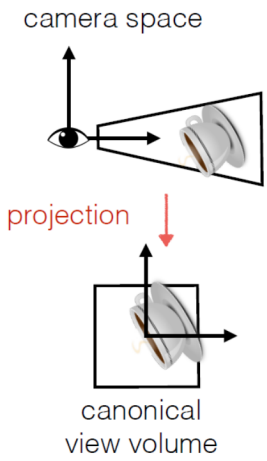

Figura 6: Projection transformation

La coordinata Z dipende dal tipo di proiezione (ortografica o prospettica).

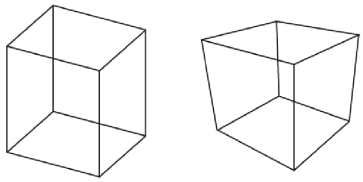

Figura 7: Proiezione ortografica (sinistra) e prospettica (destra)

5.1.4 Viewport transformation

La viewport transformation mappa le coordinate normalizzate del dispositivo (clip space) nelle coordinate dello schermo (screen coordinates), cioè le coordinate 2D dell'immagine finale. Dipende soltanto dalla risoluzione dello schermo.

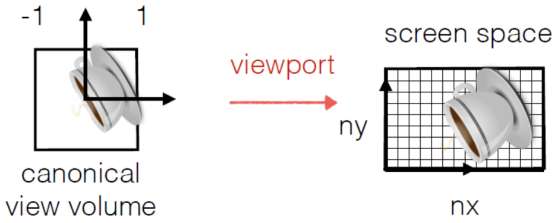

Figura 8: Viewport transformation

5.2 Pipeline

La sequenza di operazioni richiesta per trasformare gli oggetti di una scena in pixel su uno schermo è chiamata **graphics pipeline**. Ci sono due tipi di pipeline:

- **Hardware pipeline**: utilizza la GPU tramite API grafiche come OpenGL o Direct3D
- **Software pipeline**: implementa tutte le fasi della pipeline tramite codice eseguito dalla CPU

Le fasi principali della graphics pipeline sono:

1. **Input**: Gli oggetti della scena, sempre descritti da triangoli, vengono passati alla pipeline
2. **Vertex processing**: Ogni vertice dei triangoli viene trasformato dalle coordinate locali (object coordinates) alle coordinate della telecamera (camera coordinates) tramite la model transformation e la camera transformation.
3. **Rasterization**: I triangoli vengono convertiti in insiemi di pixel (fragments)
4. **Fragment processing**: Per ogni pixel viene calcolato il colore in base al modello di illuminazione scelto e alle proprietà dei materiali.

5. **Fragment blending**: I pixel vengono scritti nel framebuffer per creare l'immagine finale

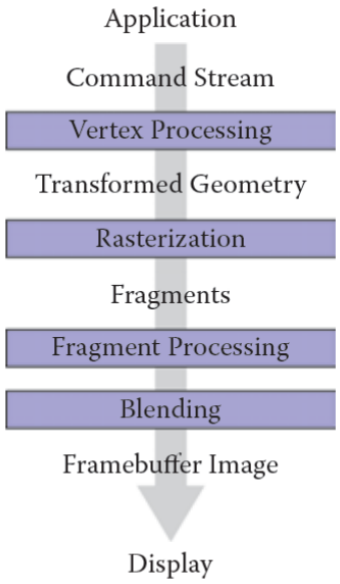

Figura 9: Fasi della graphics pipeline

Il programmatore può intervenire in due fasi della pipeline (viola nell'immagine):

- **Vertex shader**: Permette di personalizzare il processo di trasformazione dei vertici
- **Fragment shader**: Permette di personalizzare il processo di calcolo del colore dei pixel

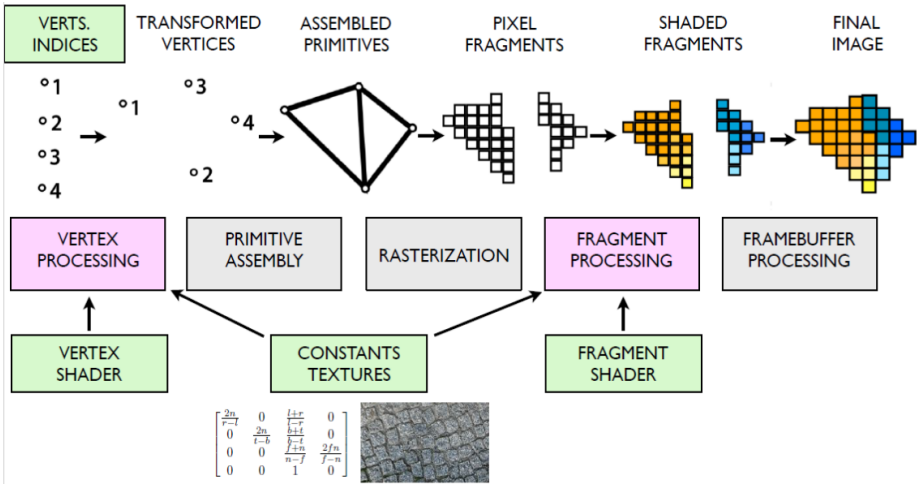

Figura 10: Esempio di graphics pipeline

5.2.1 Rasterization

La rasterization è il processo di conversione delle primitive geometriche (di solito triangoli) in pixel sull'immagine 2D.

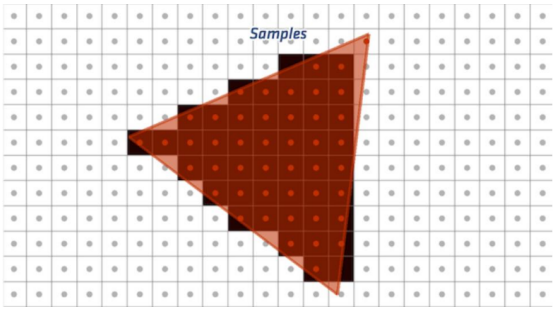

Figura 11: Rasterization

Durante la rasterization, ogni triangolo viene suddiviso in pixel e per ogni pixel viene calcolato se disegnarlo oppure no. Quando il rasterizer riceve una primitiva esegue questi passi:

1. **Enumera** i pixel coperti dalla primitiva
2. **Interpola** i valori dei vertici (coordinate, colori, normali, ecc...), chiamati attributi, per ogni pixel della primitiva

L'output del rasterizer è un insieme di **fragments**, uno per ogni pixel, che copre tutta l'area della primitiva. Ogni fragment si trova in un pixel specifico dello schermo e contiene il suo insieme di attributi.

- Prima che una primitiva venga rastereizzata, i vertici che la definiscono devono essere trasformati in coordinate screen space e i colori e altri eventuali attributi devono essere noti. Questo viene effettuato dalla fase di **vertex processing**. Anche gli attributi devono essere trasformati in clip space.
- Dopo aver rasterizzato si svolgono altri processi sui fragment, come il calcolo del colore o della profondita e viene svolto dalla fase di **fragment processing**.

5.2.2 Rasterization di linee

Per rasterizzare una linea tra due punti (x_0, y_0) bisogna rappresentarla come un rettangolo per fornire uno spessore alla linea. Si disegnano tutti i pixel che hanno il centro all'interno del rettangolo.

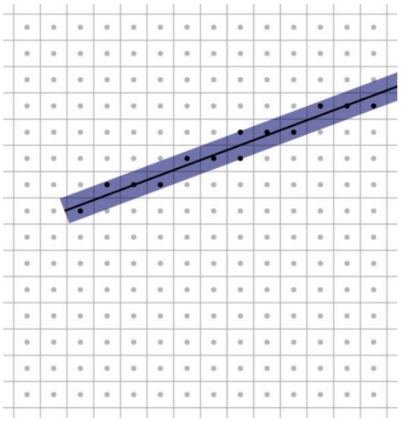

Figura 12: Rasterization di una linea

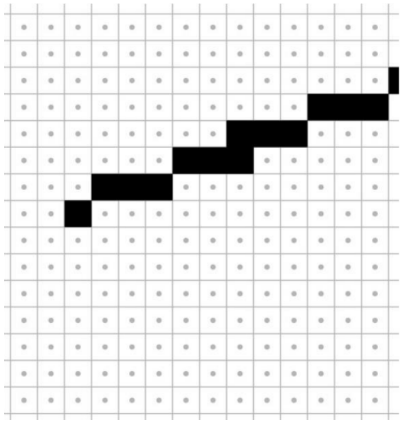

Figura 13: Output della rasterization di una linea

Si può migliorare l'approssimazione utilizzando l'algoritmo **Mid-Point** che seleziona i pixel in modo da minimizzare l'errore tra la linea ideale e la linea rasterizzata. Consideriamo l'equazione implicita di una linea tra due punti:

$$f(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + (x_0y_1 - x_1y_0) = 0$$

L'algoritmo (solo per le linee della griglia orizzontali) è il seguente:

```
y = y_0
for x = x_0 to x_1:
    draw(x, y)

    if f(x + 1, y + 0.5) < 0:
        y = y + 1
```

L'effetto di questo algoritmo è il seguente:

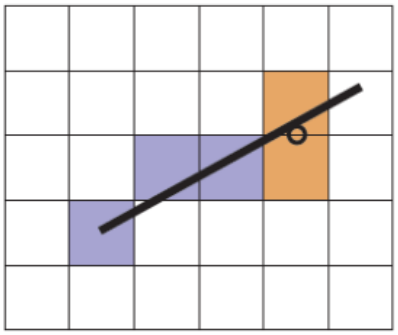

Figura 14: La linea si trova sopra il mid-point, quindi viene disegnato il pixel in alto

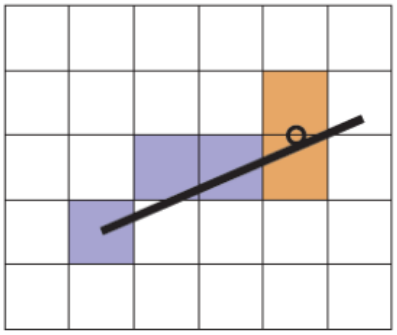

Figura 15: La linea si trova sotto il mid-point, quindi viene disegnato il pixel in basso

5.2.3 Rasterization di triangoli

Un triangolo è rappresentato come una tripla di coordinate bidimensionali:

$$(x_0, y_0), (x_1, y_1), (x_2, y_2)$$

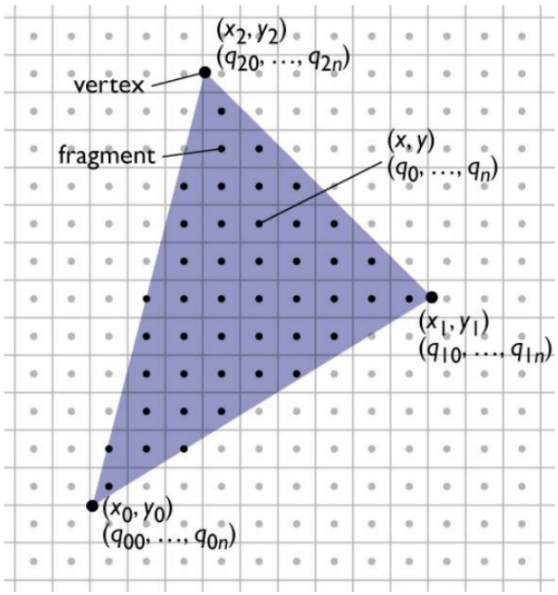

Figura 16: Rappresentazione di un triangolo

Il metodo più utile per rasterizzare un triangolo è usando le **coordinate baricentriche**, che permettono di esprimere la posizione di un punto all'interno del triangolo come combinazione lineare dei vertici del triangolo stesso:

$$x = \alpha x_0 + \beta x_1 + \gamma x_2$$
$$y = \alpha y_0 + \beta y_1 + \gamma y_2$$
$$\text{con } \alpha + \beta + \gamma = 1$$

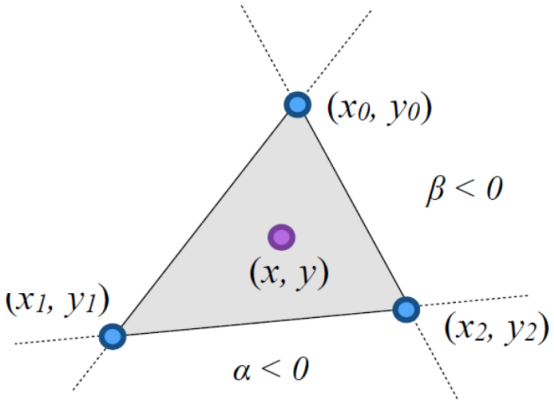

Figura 17: Coordinate baricentriche

Bisogna quindi calcolare i valori di α, β, γ che dipendono dal triangolo. Consideriamo il seguente triangolo:

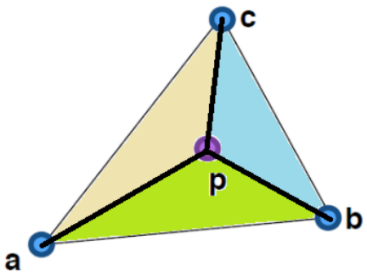

Figura 18: Triangolo per il calcolo delle coordinate baricentriche

$$\alpha(p) = \frac{area(p,b,c)}{area(a,b,c)} \quad \text{vale 1 quando } p = a$$

$$\beta(p) = \frac{area(a,p,c)}{area(a,b,c)} \quad \text{vale 1 quando } p = b$$

$$\gamma(p) = \frac{area(a,b,p)}{area(a,b,c)} \quad \text{vale 1 quando } p = c$$

La combinazione lineare delle coordinate baricentriche permette di ottenere la posizione di un qualsiasi fragment all'interno del triangolo, ma anche di interpolare qualsiasi attributo associato ai vertici del triangolo (colore, normale, coordinate texture, ecc...). Il codice per rasterizzare un triangolo è il seguente:

```
for all y:
    for all x:
        compute (alpha, beta, gamma) for pixel (x, y)
        if (alpha >= 0 and alpha <= 1 and
          beta >= 0 and beta <= 1 and
          gamma >= 0 and gamma <= 1):
            draw(x, y)
```

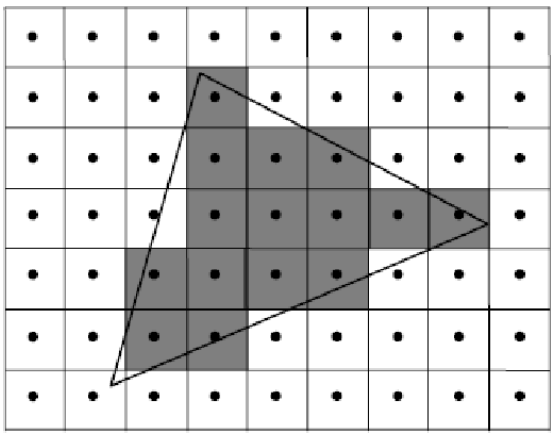

Figura 19: Rasterization di un triangolo

Un esempio di interpolazione del colore ai vertici di un triangolo è il seguente:

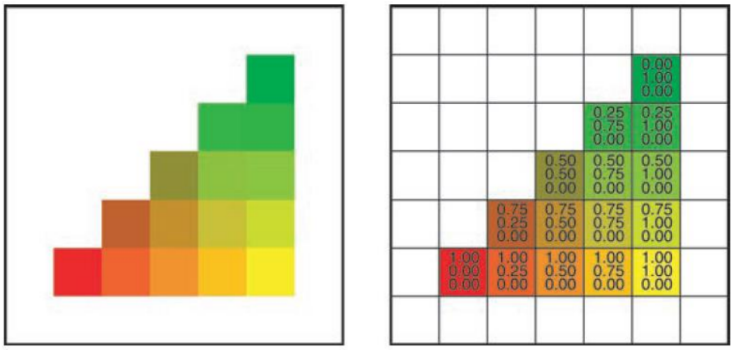

Figura 20: Interpolazione del colore in un triangolo

Quando si rasterizzano più triangoli bisogna stare attenti a non disegnare più volte lo stesso pixel:

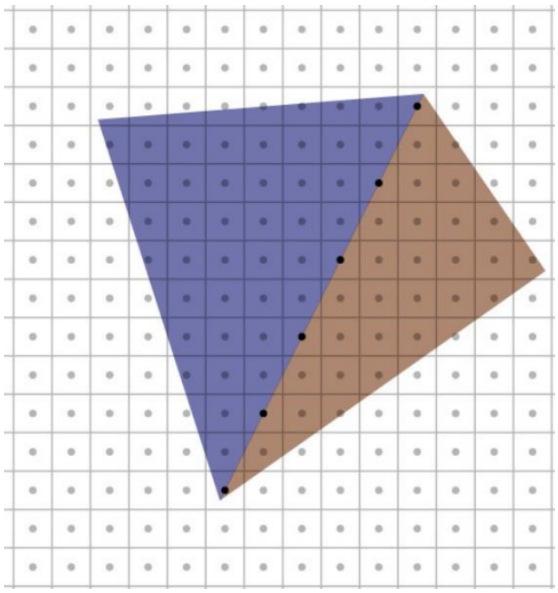

Figura 21: Esempio di overlap tra triangoli

5.2.4 Clipping

Il clipping è il processo di rimozione delle parti delle primitive geometriche che si trovano al di fuori dello schermo e al di fuori della camera virtuale. Bisogna quindi tagliare la parte non visible della primitiva e mantenere solo la parte visibile modificando eventualmente la sua topologia:

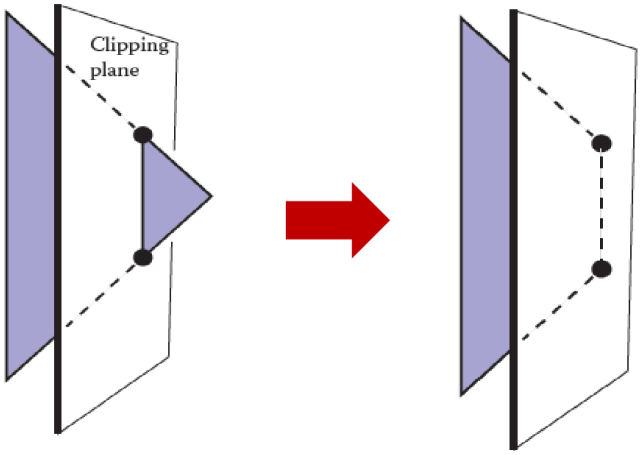

Figura 22: Clipping di una primitiva

La camera virtuale è rappresentata da un **frustum**, cioè un volume delimitato da due piani (near plane e far plane) e da quattro piani laterali (left, right, top, bottom). La scena è visualizzata soltanto se si trova all'interno del frustum, quindi tra il near plane e il far plane.

Ci sono quattro casi durante un clip:

- Tutta la primitiva è dentro il frustum: non serve fare nulla
- Tutta la primitiva è fuori dal frustum: si scarta la primitiva
- La primitiva è parzialmente dentro il frustum: si calcolano i punti di intersezione con i piani del frustum e si crea una nuova primitiva con la parte visibile
- La primitiva attraversa più piani del frustum: si calcolano i punti di intersezione con tutti i piani del frustum e si crea una nuova primitiva con la parte visibile

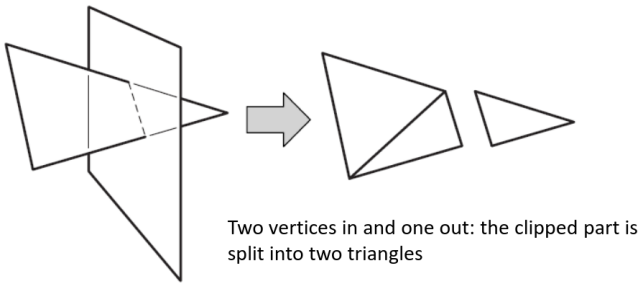

Figura 23: Cambio di topologia dopo il clipping

5.2.5 Intersezione tra una linea e un piano

Consideriamo l'equazione di un piano:

$$f(p) = n \cdot p + D = 0$$

dove:

- n è il vettore normale al piano
- p è un punto generico del piano
- D è la distanza del piano dall'origine

E l'equazione di una linea:

$$p = a + t(b - a)$$

dove:

- a e b sono i due punti estremi della linea che definisce la direzione
- t è un parametro scalare

Sostituendo l'equazione della linea in quella del piano si ottiene l'equazione per calcolare il parametro t del punto di intersezione:

$$n \cdot (a + t\,(b - a)) + D = 0$$

risolvendo per t si ottiene:

$$t = \frac{n \cdot a + D}{n \cdot (a - b)}$$

5.2.6 Blending

La fase di blending combina i fragment generati dalle primitive che si sovrappongono a ogni pixel per calcolare il colore finale. Questa fase è molto importante per gestire le occlusioni e per la rimozione delle superfici nascoste (hidden surface removal, HSR).

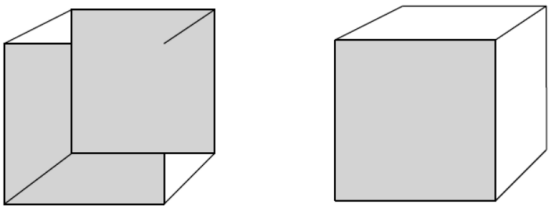

Figura 24: Blending di fragment sovrapposti

L'algoritmo di blending più è il **Painter's algorithm**, disegna per primi i fragment più lontani e poi quelli più vicini, sovrapponendoli. In questo modo, i fragment più vicini coprono quelli più lontani.

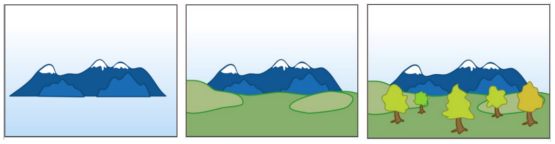

Figura 25: Painter's algorithm

Un altro metodo per il blending è il **Z-buffering**, che consiste nel mantenere una **depth map** (Z-buffer) che memorizza la profondità di ogni pixel. Se la profondità del nuovo fragment è minore di quella memorizzata nel Z-buffer, il fragment viene disegnato e la profondità viene aggiornata, altrimenti viene scartato.

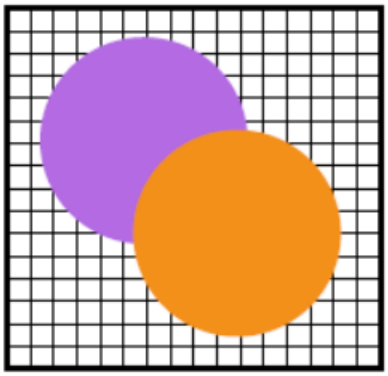

Figura 26: Buffer dell'immagine

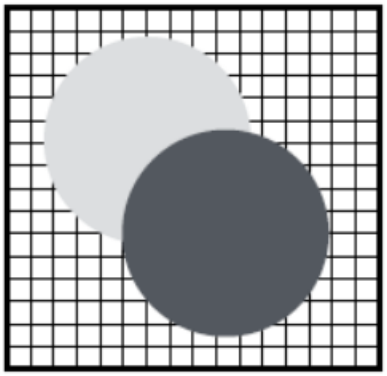

Figura 27: Buffer della profondità

Per rappresentare la profondità di solito più un pixel è scuro più è vicino alla camera. Un esempio di Z-buffering è il seguente:

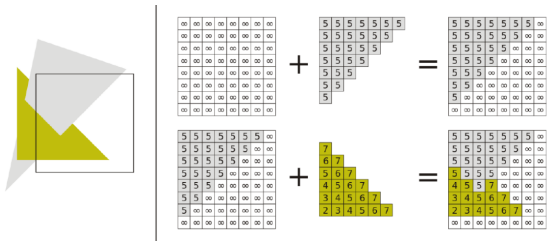

Figura 28: Esempio di Z-buffering

Siccome il buffer della profondità ha una risoluzione limitata, possono verificarsi dei problemi di **Z-fighting**, cioè due fragment molto vicini tra loro che creano artefatti visivi nell'immagine finale:

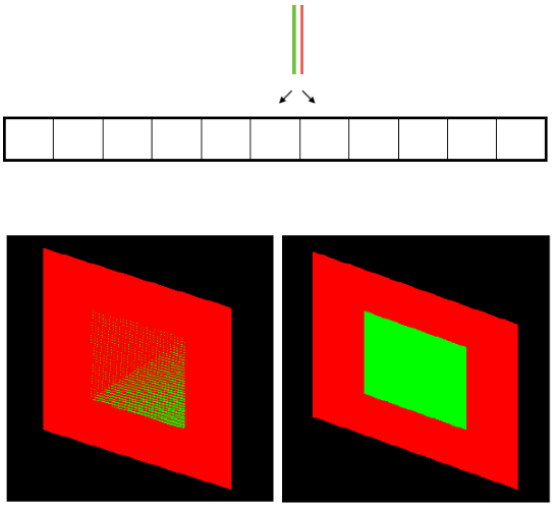

Figura 29: Esempio di Z-fighting

6 OpenGL

La pipeline di OpenGL è la seguente:

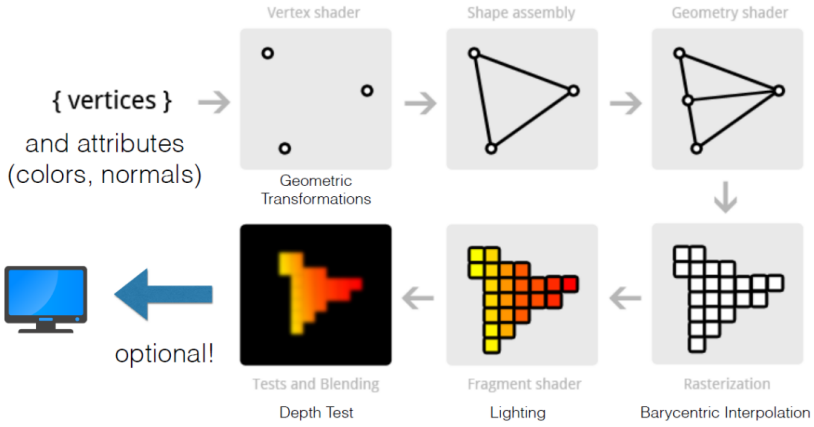

Figura 30: Pipeline di OpenGL

Questa pipeline permette all'utente di intervenire in due fasi:

- **Vertex shader**: Permette di personalizzare il processo di trasformazione dei vertici
- **Fragment shader**: Permette di personalizzare il processo di calcolo del colore dei pixel

Le componenti principali di OpenGL sono:

- **Buffers**: un allocatore lineare di memoria sul dispositivo che può memorizzare diversi tipi di dati sui quali la GPU può operare
- **Stato**: la scheda grafica mantiene uno stato computazionale che determina come avvengono le operazioni associate con i dati della scena e agli shader sulla GPU
- **Shaders**: rappresentano il meccanismo secondo il quale la computazione avviene sulla GPU. Spesso sono associati al processo per-vertex e per-fragment, ma anche sulla geometria

OpenGL è una grande macchina a stati, cioè una collezione di variabili che definiscono come OpenGL dovrebbe operare. Lo stato di OpenGL è solitamente detto **context**.

6.1 Buffers

Rappresentano la memoria interna della scheda grafica che permette alla GPU di accedere rapidamente ai dati necessari per il rendering. La CPU può scrivere direttamente nei buffers. Alla fine dell'esecuzione i contenuti del buffer possono essere copiati dalla memoria GPU alla memoria CPU.

6.1.1 Display Buffers

Alla fine della pipeline grafica si trova il **Display Buffer**, che contiene il risultato finale del rendering, cioè il valore di ogni pixel dell'immagine, solitamente un array 2D di valori RGBA.

6.1.2 Double buffering

La maggior parte delle applicazioni grafiche utilizza il **double buffering**, che consiste nell'avere due display buffer: uno visibile (front buffer) e uno nascosto (back buffer). La GPU disegna l'immagine nel back buffer mentre il front buffer viene visualizzato sullo schermo. Quando il rendering è completo, i due buffer vengono scambiati (swap) tramite uno scambio di puntatori per mostrare l'immagine appena disegnata.

6.2 Shaders

Gli shader sono **general purpose functions** che vengono eseguite in parallelo sulla GPU e permettono di personalizzare il processo di rendering. Gli shader sono scritti in un linguaggio simile al C chiamato GLSL (OpenGL Shading Language). Le fasi della pipeline che si possono personalizzare con gli shader sono:

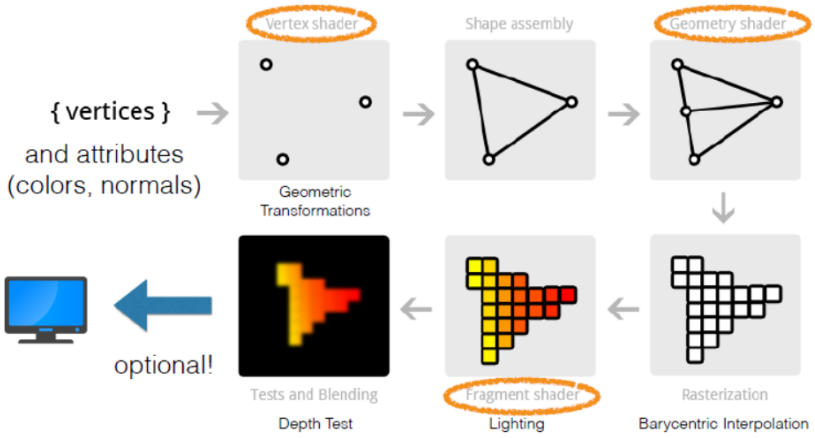

Figura 31: Fasi della pipeline personalizzabili con gli shader

6.2.1 Vertex shader

Il vertex shader è un programma sulla GPU che esegue operazioni su ciascun vertice e i relativi attributi prima che vengano inseriti nel vertex array. Questa funzione deve mettere in output la posizione finale del vertice nello spazio del dispositivo (device coordinates) e anche qualsiasi altro attributo utile per il fragment shader. Tutte le trasformazioni dalle coordinate del mondo alle coordinate del dispositivo avvengono nel vertex shader.

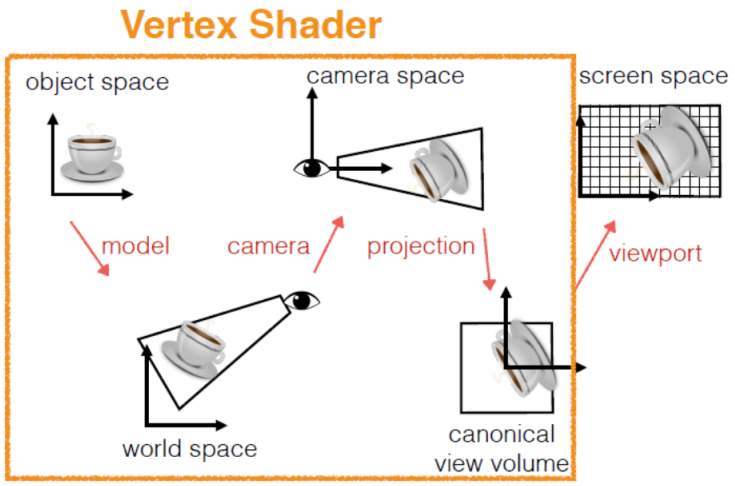

Figura 32: Vertex shader

6.2.2 Fragment shader

L'output del vertex shader viene interpolato per ogni pixel coperto da una primitiva sullo schermo. Questi pixel, chiamati **fragments**, sono i dati su cui opera il fragment shader. L'output del fragment shader è il colore finale del pixel che verrà scritto nel display buffer.

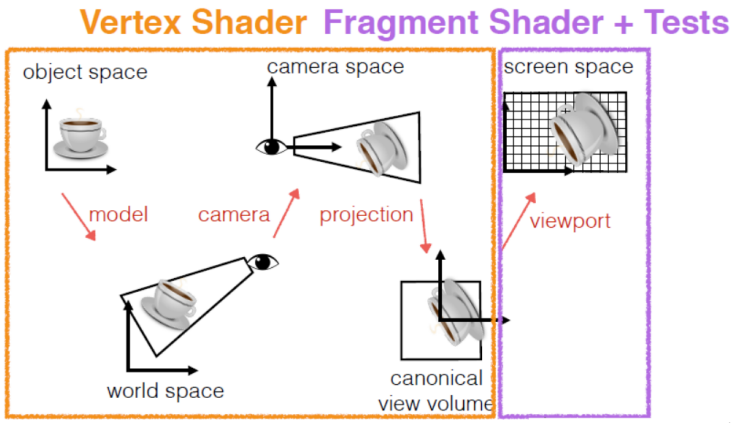

Figura 33: Fragment shader

6.3 Basi di OpenGL

6.3.1 Creazione del contesto

Prima di poter disegnare qualcosa con OpenGL bisogna creare un **contesto**, cioè:

- Aprire una finestra (ad esempio chiedere al sistema operativo di dare accesso ad una porzione di schermo)
- Inizializzare l'API di OpenGL e assegnargli la porzione di schermo disponibile

Questa fase dipende molto dal sistema operativo e dall'hardware.

6.3.2 Window manager

Ci sono molte librerie che permettono di creare una finestra e un contesto OpenGL, nascondendo tutte le complessità legate al sistema operativo e fornisce un'interfaccia cross-platform. In questo corso si utilizza **GLFW** `https://www.glfw.org/`. Questa libreria permette non solo di creare una finestra, ma anche di gestire gli eventi come input da tastiera e mouse.