

Appunti Grafica

Matteo Berti

A.A. 18/19

Contenuti

1	Sistema grafico	3
1.1	Frame Buffer	3
1.2	Rendering Pipeline	4
2	Geometria	5
2.1	Prodotto scalare (dot product)	5
2.2	Norma Euclidea	5
2.3	Proiezione vettoriale	5
2.4	Prodotto vettoriale (cross product)	5
2.5	Coordinate baricentriche (2D)	5
2.6	Equazione parametrica di una retta	5
2.7	Coordinate omogenee	5
2.8	Cambio del sistema di riferimento	6
2.9	Trasformazioni	6
3	Introduzione alla modellazione	7
3.1	Punti	7
3.2	Curve	7
3.3	Superfici	8
3.4	Continuità	8
4	Curve di Bézier	8
4.1	Proprietà	8
4.2	Valutazione	9
4.3	Suddivisione	9
4.4	Rendering	10
4.5	Connessione di curve	10
5	Curve spline	11
5.1	Proprietà	11
5.2	Controllo della forma	12
5.3	Valutazione	12
5.4	NURBS	12
6	Rendering	12
6.1	Geometry Stage	12
6.2	Clipping	15
6.3	Rasterization Stage	16
6.4	Fragment Stage	18
7	Illuminazione	19
7.1	Sorgenti luminose	19
7.2	Modello di illuminazione di Phong	19
7.3	Shading	20
7.4	Ombre	22
7.5	Trasparenza	23

8 Ray Tracing	24
8.1 Ray Casting	24
8.2 Riflessione	24
8.3 Rifrazione	25
8.4 Ombre	25
8.5 Ricorsione	25
9 Texturing	26
9.1 Texture mapping	26
9.2 Bump mapping	27
9.3 Procedural texturing	28
9.4 Environment mapping	28
10 Shaders	29
10.1 GLSL	29
11 Animazione digitale	30
11.1 Animazioni basate sulla fisica	30
11.2 Motion Capture (MOCAP)	31
11.3 Keyframes	31
12 Camera path	32
12.1 Fernet frame	32
12.2 Center of Interest	32
12.3 Look ahead	32

1 Sistema grafico

Scanning 3D

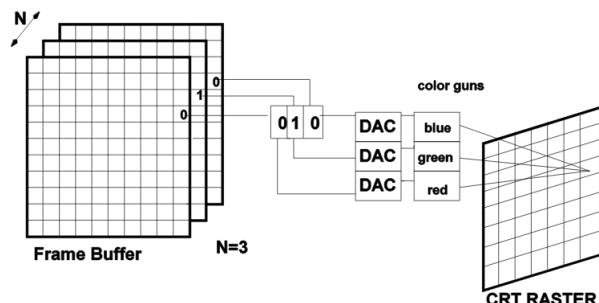
- Con contatto
 - Misurazione diretta (coordinate measuring machines, AFM, digitizer)
- Senza contatto
 - Passivo
 - * Shape-from-X: multi-view, siluhettes, motion
 - Attivo
 - * Trasmissivo
 - Computed Tomography (CT)
 - Trasmissive Ultrasound
 - * Riflessivo
 - Metodi non-ottici (radar, sonar, ultrasuoni riflessivi)
 - Time-of-Flight
 - Triangolazione (laser e camera che osserva la forma)

1.1 Frame Buffer

Il frame buffer è una memoria RAM che memorizza immagini prima di visualizzarle sullo schermo. Contiene un insieme di matrici ognuna delle quali rappresenta tutti i pixel dello schermo. Ha tante matrici quanti sono i bit con cui si vuole rappresentare il colore di ogni singolo pixel.

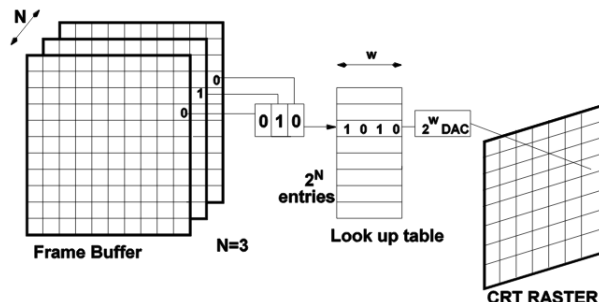
Modello true-color

Il frame buffer contiene i componenti del colore per ogni pixel, quindi un frame buffer con N piani memorizza 2^N valori diversi di colore.



Modello pseudocolor

Il frame buffer contiene gli *indirizzi* ad una tabella di colori chiamata **colormap**, la quale memorizza 2^N elementi ognuno di w bit. Quindi la colormap può memorizzare fino a 2^w colori in ogni cella, ma solamente 2^N colori diversi sono disponibili sul frame buffer.



In generale il framebuffer utilizza 32 bit per ogni pixel: $8 + 8 + 8 + 8$ RGBA.

1.2 Rendering Pipeline

Definizione: sequenza di passaggi usata per creare una rappresentazione *raster* 2D di una scena 3D.

La fase di **Application** nella CPU passa alla GPU un'insieme di dati che descrivono posizioni dei vertici, ma anche le textures, i colori, le luci, la camera e tutte le risorse utili per renderizzare la scena (es. un file .obj).

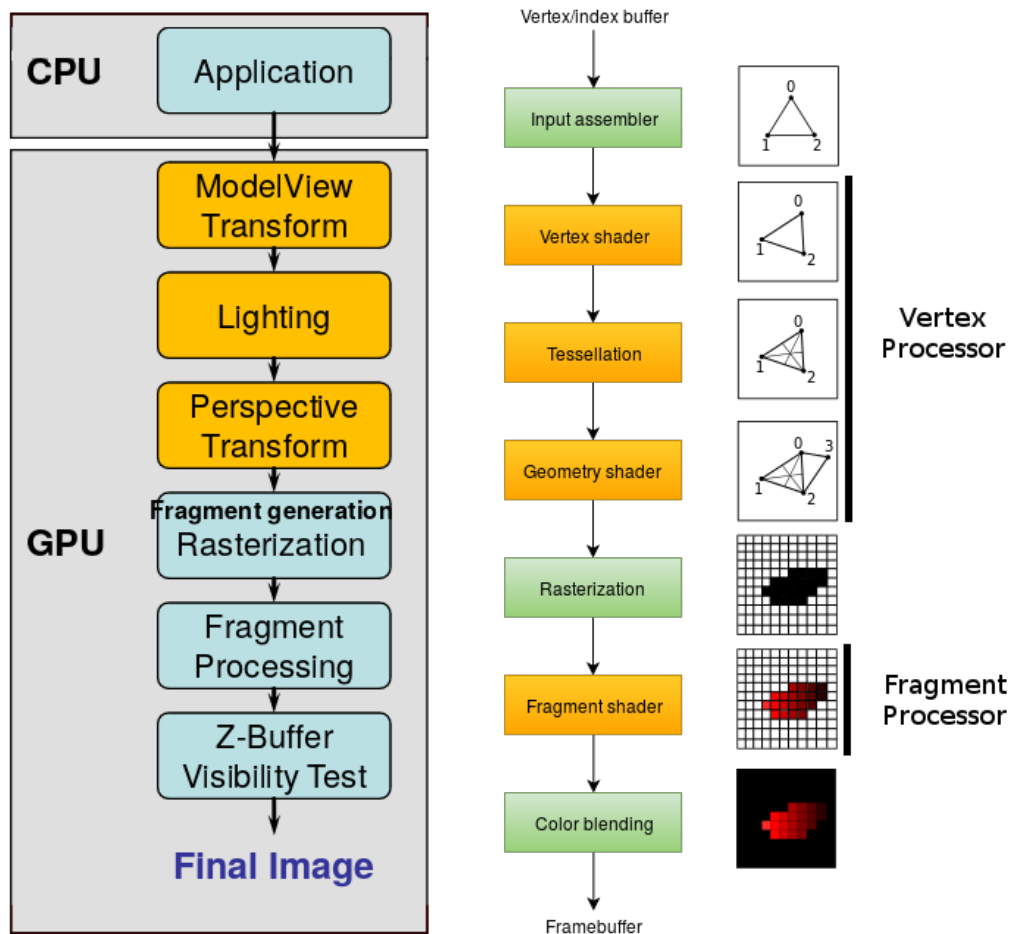
Successivamente nella fase di **Vertex processing** per prima cosa si identificano i vertici e si posizionano nel WCS (**vertex shader**), poi si suddividono in forme elementari (i.e. triangoli) **tassellando**, e infine gli elementi derivati dall'illuminazione, trasportando la scena in VCS.

La fase di **Rasterization** non è programmabile e non fa altro che convertire i vertici in *frammenti* (che sono diversi dai pixel) tramite interpolazioni ed elaborazioni dei vertici 3D.

Infine la fase di **Fragment processing** ha come scopo quello di calcolare il colore finale di ogni frammento, tenendo conto di vari aspetti quali illuminazione, posizione dell'osservatore, interpolazione dei colori ai vertici e molto altro.

In conclusione dopo aver gestito trasparenze e sovrapposizioni degli elementi tramite **Z-Buffer** si produce in output un insieme di dati che andranno a riempire il *frame buffer*.

L'intera pipeline di rendering è **multithreading**, ovvero possono essere eseguiti ad esempio vertex processing e fragment processing in parallelo (chiaramente su elementi diversi).



2 Geometria

2.1 Prodotto scalare (dot product)

Dati due vettori a e b in \mathbb{R}^n si ottiene un valore scalare che corrisponde a (definizione algebrica):

$$\langle a, b \rangle = \sum_{i=1}^n a_i b_i.$$

Il valore ottenuto indica la relazione tra due vettori, precisamente, se i vettori sono sovrapposti il risultato sarà 1, se vi è un angolo di 90° il risultato sarà 0, se l'angolo è di 180° sarà -1 (supponendo $\|a\| = \|b\| = 1$).

2.2 Norma Euclidea

La norma è una nozione di lunghezza che viene preservata da rotazioni, traslazioni e riflessioni dello spazio, dato un vettore a in \mathbb{R}^n : $\|a\|_2 = \sqrt{\sum_{i=1}^n a_i^2}$.

2.3 Proiezione vettoriale

Dato un vettore b , proiettato su un vettore u , è possibile calcolarne il vettore proiezione a tramite:

$$\|a\| = \frac{\langle b, u \rangle}{\|u\|}.$$

2.4 Prodotto vettoriale (cross product)

Dati due vettori si ottiene un terzo vettore perpendicolare ad entrambi chiamato **normale**, in \mathbb{R}^3 :

$$a \times b = [a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x].$$

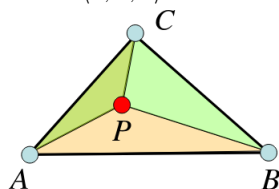
$$\mathbf{x} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix} \quad \mathbf{z} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

2.5 Coordinate baricentriche (2D)

Dato un triangolo composto da 3 punti A , B e C , la posizione del punto P all'interno di esso è data da:

$$P = \frac{\langle P, B, C \rangle}{\langle A, B, C \rangle} A + \frac{\langle P, C, A \rangle}{\langle A, B, C \rangle} B + \frac{\langle P, A, B \rangle}{\langle A, B, C \rangle} C.$$

In cui $\langle *, *, * \rangle$ indica l'area del triangolo.



2.6 Equazione parametrica di una retta

L'insieme di tutti i punti che passano per P_0 nella direzione del vettore v sono rappresentabili tramite: $l(t) = P_0 + tv$ in cui $t \in (-\infty, \infty)$ per la retta e $t \in (0, \infty)$ per il raggio.

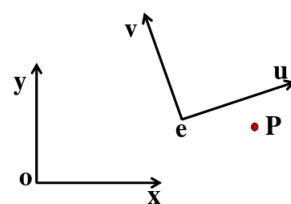
2.7 Coordinate omogenee

Rappresentare vettori e punti con tre coordinate può risultare ambiguo, per questo viene aggiunto un quarto elemento (0 o 1) che indica se si è in presenza di un vettore o un punto:

vettore: $w = a_1 v_1 + a_2 v_2 + a_3 v_3 + 0 \cdot P_0$

punto: $P = a_1 v_1 + a_2 v_2 + a_3 v_3 + 1 \cdot P_0$

2.8 Cambio del sistema di riferimento



$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

$$P_{xy} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} P_{uv}$$

I vettori u e v sono unitari, usati per fornire l'orientamento del sistema di coordinate, mentre il vettore e indica l'origine del sistema.

2.9 Trasformazioni

Traslazione

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Non cambia l'orientamento del sistema di coordinate, ma solo il posizionamento dell'"origine" (4° vettore).

Rotazione (Asse x)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

L'asse x rimane immobile, cambia solo l'orientamento degli altri due assi, lasciando comunque invariato il loro orientamento rispetto ad x .

Rotazione (Asse y)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotazione (Asse z)

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Scala

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Generalizzazione

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Linear Transformations
 Translations
 Perspective Projection

3 Introduzione alla modellazione

Per descrivere una geometria ci sono due modi: implicito ed esplicito (parametrico).

3.1 Punti

Implicita: i punti x sono definiti da una funzione implicita $\phi(x)$ tale che: $\phi(x) = 0$.

Esempio: $\phi = x^2 - 1$, l'interfaccia (ovvero i punti) saranno $\delta\Omega = \{-1, 1\}$. Se $\phi > 0$ si è fuori dalla parabola, se $\phi < 0$ si è dentro alla parabola e se $\phi = 0$ si è proprio sulla parabola.

Esplicita: rappresentata dalle coordinate in 2D o 3D.

3.2 Curve

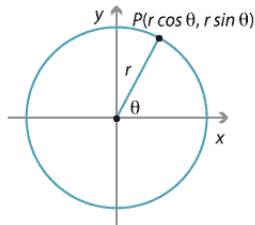
Implicita: la curva è descritta dall'insieme di punti (x, y) che soddisfa la funzione: $\phi(x, y) = 0$.

Esempio $\phi(x, y) = x^2 + y^2 - 1$ (cerchio) ha un'interfaccia (ovvero la curva) $\delta\Omega = \{\frac{x}{||x||} = 1\}$.

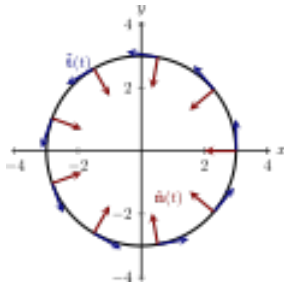
Esplicita: si definiscono esplicitamente punti sulla curva.

Curva - forma parametrica

Una curva parametrica si esprime come una funzione vettoriale composta da tanti vettori quante sono le dimensioni dello spazio, questo aggiunge una proprietà, la direzione dei vettori. La funzione vettoriale ha forma $C(t) = (x(t), y(t), z(t))$ di parametro $t \in [a, b]$. Variando t , le coordinate $(x(t), y(t), z(t))$ rappresentano un punto che si muove lungo la curva, e permette di conoscerne anche la velocità di percorrimento.



Il vettore tangente, ovvero il vettore che è tangente ad una curva/superficie in un certo punto, è una funzione $C(t) = (x(t), y(t))$ con $t \in [0, 1]$ dominio parametrico. La velocità di movimento di un punto sulla curva è calcolabile tramite la derivata in t : $C'(t) = (x'(t), y'(t))$.



Forme parametriche

- Linea: $C(t) = P_0 + tv$; con $t \in (-\infty, \infty)$
- Raggio: $C(t) = P_0 + tv$; con $t \in (0, \infty)$
- Segmento: $C(t) = P_0 + t(P_1 - P_0)$; con $t \in [0, 1]$
- Cerchio: $C(t) = (\cos(t), \sin(t))$; con $t \in [0, 2\pi]$

3.3 Superfici

Implicita: le superfici implicite rappresentano una superficie come un particolare insieme di livello di una funzione di embedding di una dimensione più alta su \mathbb{R}^3 $\phi(x, y, z) = 0$.

Esempio la sfera unitaria sono tutti i punti x tali che $x^2 + y^2 + z^2 = 1$. La regione interna è $\Omega - (\phi(x, y, z) < 0)$, la regione esterna $\Omega + (\phi(x, y, z) > 0)$, mentre la superficie è $\delta\Omega$ ($\phi(x, y, z) = 0$).

Esplicita: $f: \mathbb{R}^2 \rightarrow \mathbb{R}^3; (u, v) \rightarrow (x(u, v), y(u, v), z(u, v))$.

3.4 Continuità

- C^0 : è sufficiente non abbia “buchi” ovvero spazi interrotti, in gergo “watertight”.
- C^1 : oltre ad essere C^0 ha anche derivate continue.
- C^2 : oltre ad essere C^1 ha anche derivate seconde continue (è importante per lo shading).

4 Curve di Bézier

Una curva espressa in forma parametrica $C(t) = at^2 + bt + c$ può essere riscritta come:

$$C(t) = P_0(1-t)^2 + P_1 2t(1-t) + P_2 t^2$$

In cui i punti P_0 , P_1 e P_2 rappresentano i **control points**. Questa curva è chiamata di Bézier e ha grado 2.

Generalizzando l'idea, è possibile rappresentare una curva di n control points tramite la sommatoria della moltiplicazione di ogni punto per la corrispondente **base di Bernstein**: $C(t) = \sum_{i=0}^n P_i B_i^n(t)$. In cui le funzioni delle basi di Bernstein sono:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \quad i = 0, \dots, n$$

La **derivata prima** di una curva di Bézier risulta essere un'altra curva di Bézier di grado diminuito di 1, e si calcola banalmente derivando ogni componente della funzione. La derivata di una curva rappresenta il **vettore tangente** alla curva in un certo punto.

4.1 Proprietà

- La curva interpola solo il primo e l'ultimo punto di controllo: $C(0) = P_0$, $C(1) = P_n$.
- La sua variazione di diminuzione ha al più le stesse intersezioni con una retta che l'attraversa, che il suo poligono di controllo.
- È **invariante** in base ad una trasformazione affine (traslazione, rotazione, scala o shear): applicare una

trasformazione affine ai punti di controllo e poi valutarne la curva in t_i , equivale ad applicare la trasformazione affine al punto $C(t_i)$.

- La curva è **tangente** al primo ed ultimo punto di controllo.
- La curva è contenuta nell'involuppo convesso (**convex hull**) formato dai punti di controllo, il quale è all'interno del più piccolo poligono formato dai suoi punti di controllo.
- **Precisione lineare**: segue dalla proprietà precedente che quando tutti i punti di controllo risiedono su una linea, allora la curva di Bézier è il segmento linea che interpola i punti.

4.2 Valutazione

1. Basi polinomiali di Bernstein

I polinomi di Bernstein di grado n sono invarianti sotto una riparametrizzazione affine, ovvero dato un generico intervallo $[a, b]$, il seguente polinomio risulta del tutto identico a quello visto in precedenza:

$$B_i^n(x) = \binom{n}{i} \frac{(b-x)^{n-i} (x-a)^i}{(b-a)^n} \quad i = 0, \dots, n$$

Per valutare $f(x)$ con $x \in [a, b]$ prima di tutto va determinato $t = \frac{x-a}{b-a}$ poi si valuta $f(t)$, si porta quindi la valutazione nell'intervallo $[0, 1]$ che così risulta identica.

2. Forma matriciale

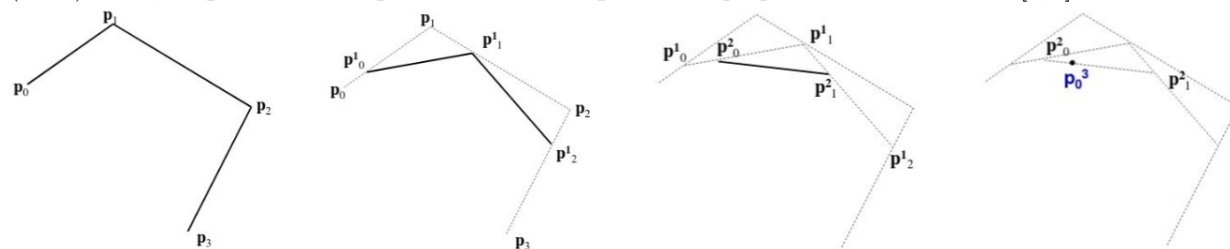
Un polinomio è una combinazione lineare delle funzioni delle basi di Bernstein: $f(t) = c_0 B_0^n(t) + c_1 B_1^n(t) + \dots + c_n B_n^n(t)$ e può essere rappresentato in forma matriciale nel seguente modo:

$$f(t) = \begin{bmatrix} 1 & t & t^2 & \dots & t^n \end{bmatrix} \begin{bmatrix} b_{00} & 0 & \dots & 0 \\ b_{10} & b_{11} & \dots & 0 \\ \dots & \dots & \dots & 0 \\ b_{n0} & b_{n1} & \dots & b_{nn} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \dots \\ c_n \end{bmatrix}$$

3. Algoritmo di de Casteljau

Si valuta il valore del polinomio $f(t)$ applicando l'algoritmo ad ogni componente della curva $(x(t), y(t), z(t))$ poi si traccia la curva per mezzo di una sequenza di interpolazioni lineari ricorsive.

L'interpolazione lineare calcola un valore nel mezzo di altri due valori, il quale può essere uno scalare, un vettore, un colore, ecc. L'interpolante lineare tra due punti a e b con parametro t è data da: $Lerp(t, a, b) = (1-t)a + tb$, in questo modo è possibile trovare il punto t in *proporzione* all'intervallo $[a, b]$.



Ripetendo questo processo per un numero elevato di $t \in [0, 1]$ si può ricostruire la curva.

4.3 Suddivisione

Per suddividere una curva si spezza in un certo punto t_0 in modo da avere due curve C_1 in $[0, t_0]$ e C_2 in $[t_0, 1]$ tali che:

$$C_1(t) = C(t \cdot t_0)$$

$$C_2(t) = C(t_0 + t \cdot (1 - t_0))$$

I punti di controllo delle due sotto-curve saranno, per la prima, tutti i primi punti di ogni iterazione, e per

la seconda, tutti gli ultimi punti di ogni iterazione.

La suddivisione in forma matriciale è rappresentata in forma di basi di Bernstein nell'intervallo $[0, \frac{1}{2}]$:

$$C_1(t) = \begin{bmatrix} 1 & t & \left(\frac{t}{2}\right)^2 & \left(\frac{t}{2}\right)^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1/4 & 0 \\ 0 & 0 & 0 & 1/8 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} =$$

$$= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} SM \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix} = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} MM^{-1}SM \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$S_{[0,1/2]} = M^{-1}S M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \\ 1/4 & 1/2 & 1/4 & 0 \\ 1/8 & 3/8 & 3/8 & 1/8 \end{bmatrix}$$

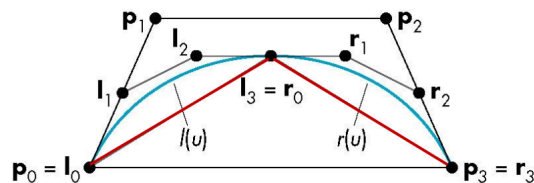
$\mathbf{T} \cdot \mathbf{M} = \mathbf{B}$

$$C_1(t) = TMS_{[0,1/2]}P = BS_{[0,1/2]}P$$

4.4 Rendering

Si disegna una curva di Bézier come una serie di segmenti, per far ciò si hanno due metodi:

- **Metodo uniforme:** si discretizza l'intervallo parametrico in N punti equidistanti, poi si disegna l'unione poligonale corrispondente ai punti valutati sulla curva. Tuttavia se avessimo 100 punti tutti allineati si eseguirebbe 100 volte un calcolo con lo stesso risultato.
- **Metodo della suddivisione adattiva:** è un metodo di rendering ottimizzato basato su suddivisioni adattive della curva: si spezza la curva in sotto-curve sempre più piccole finché ogni sotto-curva non è sufficientemente vicina ad essere una linea retta (il "sufficientemente" è dato da una soglia impostabile), in modo che renderizzare le sotto-curve come linee rette dia risultati adeguati. Per ogni punto di controllo interno si calcola la distanza d dalla corda che unisce il primo e l'ultimo punto di controllo della curva. Se tutte le distanze d sono inferiori ad una certa tolleranza ϵ , le sotto-curve sono considerate piatte.



4.5 Connessione di curve

Le curve di Bézier non possono avere grado infinito, anzi è anche consigliabile mantenere un grado basso suddividendo una curva molto complessa in varie sotto-curve di grado inferiore. Queste sotto-curve vanno unite tra loro, e per far ciò si hanno due modi:

- **Curve di Bézier a tratti:** polinomi a tratti, una relazione geometrica dei punti di controllo adiacenti alle congiunzioni determina le condizioni di continuità.
- **Spline polinomiale:** si ha un polinomio a tratti con certe condizioni di regolarità nelle congiunzioni.

Continuità parametrica e geometrica

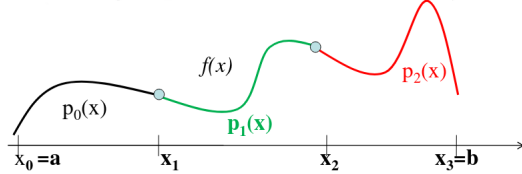
- C_0 : continuo, la congiunzione può essere appuntita.
- C_1 : continuità parametrica, le tangenti sono le stesse (uguale direzione e modulo) nella congiunzione.
- G_1 : continuità geometrica, le tangenti hanno le stesse direzioni.
- C_2 : continuità della curvatura, le tangenti e le loro derivate sono le stesse.

5 Curve spline

Le spline sono usate per creare curve più strette, ovvero più vicine al poligono di controllo. Dato un insieme di punti (nodi) $\Delta = \{x_i\}_{i=1 \dots k}$ che partizionano l'intervallo $[a, b]$:



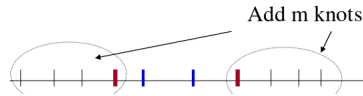
È possibile costruire un **polinomio a tratti** composto da $k + 1$ polinomi di grado n , uno per ogni intervallo I_i , uniti con continuità C^0 . L'ordine di una curva spline è $m = n + 1$.



Ogni **funzione spline** $s(t)$ di grado n (ed ordine $m = n + 1$) può essere rappresentata come una combinazione lineare delle Funzioni Base dello spazio spline $S(P_m, M, \Delta)$. In cui Δ è l'insieme dei nodi che partizionano l'intervallo ed $M = (m_1, \dots, m_k)$ è un vettore di molteplicità di nodi:

$$s(t) = \sum_{i=1}^{m+K} c_i N_{i,n}(t)$$

Dove $K = \sum_{i=1}^k m_i$ ed $N_{i,n}(t)$ è una funzione base per lo spazio spline. L'insieme M contiene i nodi da inserire prima e dopo i k nodi della spline, si chiama partizione estesa:



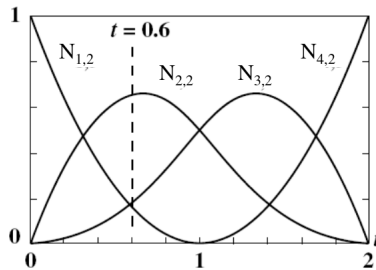
Le **Funzioni Base** sono calcolate tramite:

$$N_{i,n}(t) = \frac{t - t_i}{t_{i+n} - t_i} N_{i,n-1}(t) + \frac{t_{i+n+1} - t}{t_{i+n+1} - t_{i+1}} N_{i+1,n-1}(t)$$

$$N_{i,0}(t) = 1 \text{ sse } t_i \leq t < t_{i+1} \text{ altrimenti } 0$$

Esempio

Dato il grado $n = 2$ e il vettore di nodi $\Delta^* = (0, 0, 0, 1, 2, 2, 2)$, sia $t = 0.6$, il risultato è il seguente:



5.1 Proprietà

- **Supporto locale:** $N_{i,n}(t) = 0$ per ogni t che non appartiene all'intervallo $[t_i, t_{i+n+1})$
- **Non negatività:** $N_{i,n}(t) > 0$ per ogni t che appartiene all'intervallo (t_i, t_{i+n+1})
- **Partizione dell'unità:** $\sum_{i=0}^{m+K} N_{i,n}(t) = 1$ per ogni t che appartiene all'intervallo $[a, b]$

5.2 Controllo della forma

- Aumentare il grado del polinomio: questo lascerà la forma invariata ma permetterà di avere più punti di controllo con cui modificare la curva.
- Spostare i control point esistenti.
- Usare più control point coincidenti: più control point coincidono nello stesso punto, più la curva sarà appuntita in quel punto.
- Usare più nodi interni uguali: se un nodo ha molteplicità n , allora la curva passerà proprio in corrispondenza di quel control point.
- Modificare il vettore di nodi: in base al numero di elementi ripetuti e distinti nel vettore di nodi, cambia notevolmente la forma della spline.

5.3 Valutazione

1. Funzioni Base

Si calcolano le funzioni base $N_{i,n}$ in ogni intervallo e si effettuano le combinazioni lineari.

2. Algoritmo di de Boor

Si calcolano i coefficienti c_l e si combinano.

5.4 NURBS

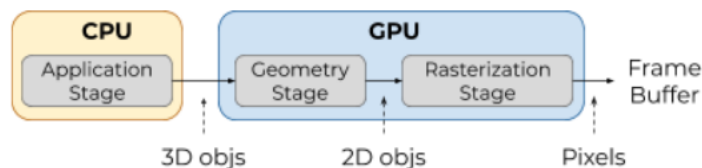
Le NURBS (Non Uniform Rational BSpline) sono una generalizzazione delle spline (le quali sono una generalizzazione delle Bézier). Per queste curve, oltre al vettore di molteplicità dei nodi M viene fornito anche un vettore di pesi W . Una curva NURBS è espressa come:

$$s(t) = \frac{\sum_{i=1}^{ncp} P_i w_i N_{i,n}(t)}{\sum_{i=1}^{ncp} w_i N_{i,n}(t)}$$

Le NURBS forniscono più gradi di libertà delle spline.

6 Rendering

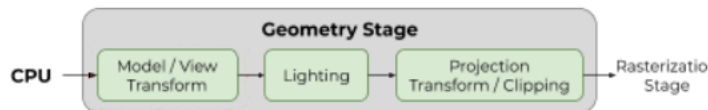
In linea concettuale si hanno 3 fasi nella pipeline: **application**, gestita dalla CPU la quale modella le forme geometriche e passa i dati sui vertici alla GPU; **geometry**, che esegue operazioni geometriche sui vertici 3D come la trasformazione di vertici e della camera, il calcolo della luce, il clipping, ecc; **rasterization**, qui avviene la trasformazione da poligoni 2D a pixel effettivi sul frame buffer, più precisamente un primitiva (es un vertice) viene convertito in un frammento, un elemento che ha una posizione, un colore, una profondità, ecc.



6.1 Geometry Stage

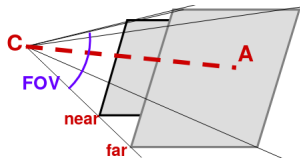
La geometry stage si compone di 3 sotto-fasi principali:

- **Trasformazioni Model/View:** la fase di Model Transform prende in input i vertici degli oggetti espressi in Object Coordinate System (OCS) e restituisce in output i vertici della scena nel sistema di coordinate di un “mondo” 3D condiviso (WCS). La conversione di coordinate avviene tramite la moltiplicazione di ogni vertice per una matrice 4x4 di trasformazione affine. La fase di View Transform prende in input i vertici espressi nelle coordinate mondo (WCS) e restituisce in output i vertici della scena secondo la camera (VCS). Per far ciò la camera è posizionata ed orientata tramite WCS, viene moltiplicata una matrice di trasformazione per cambiare il sistema di riferimento, ed infine si applica questa trasformazione ad ogni vertice in scena.
- **Illuminazione:** l’illuminazione viene affrontata in dettaglio dopo, in breve si calcolano i vettori di provenienza delle sorgenti luminose e in base a quelli si ottengono shading e ombre.
- **Trasformazioni Projection e Clipping:** la fase di Projection Transform prende in input i vertici 3D espressi in coordinate camera (VCS) e ritorna in output le coordinate per uno schermo 2D dei vertici visibili dalla camera (SCS). Il Clipping consiste nel ritagliare (o meglio, non “disegnare”) i vertici fuori dal volume dello spazio tra i piani *front* e *back* sia totalmente esterni che quelli appartenenti ad oggetti posti a metà del volume, definendo di fatto cosa la camera può vedere.



Camera

Per il modello della camera artificiale sono necessarie 4 informazioni: il punto **C**, la posizione della camera in WCS; il punto **A**, il vettore di vista specifica in quale direzione punta la camera, ovvero il centro della scena; il **FOV**, o Field Of View specifica l’ampiezza dell’angolo, la normale, ecc; la **profondità** del campo, data da due piani uno vicino, “*near*”, ed uno lontano “*far*” tra i quale avviene la scena, prima e dopo questi due piani non è visibile nulla.

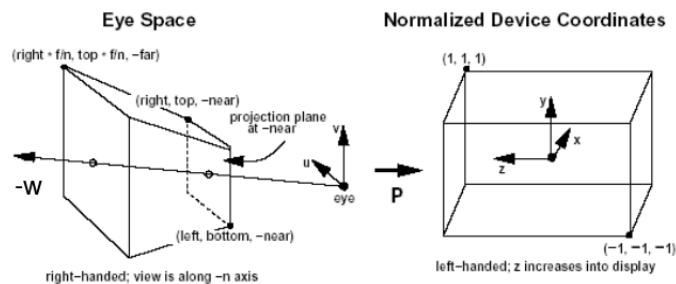


VCS

Supponiamo di voler renderizzare un’immagine di una sedia osservata dalla camera. La sedia è posizionata nello spazio mondo (WCS) con matrice T_m , la camera è posizionata nello spazio mondo (WCS) con matrice T_v , la seguente trasformazione converte ogni vertice v dallo spazio oggetto della sedia nello spazio mondo e poi dallo spazio mondo allo spazio camera: $v' = T_v \cdot T_m \cdot v$.

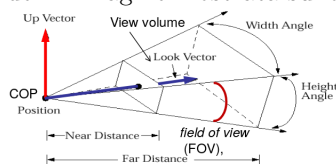
Proiezione

La proiezione può essere **parallela** (ortogonale) in cui tutti i punti dell’oggetto visto sono paralleli alla camera. Oppure **prospettica** in cui vengono mantenute le proporzioni, ovvero man mano che le parti dell’oggetto sono lontane risultano più piccole. Nella fase di projection transformation, prima si determina il volume dello spazio tra i piani di clipping *front* e *back*, che definisce lo spazio limitato che la camera può “vedere” (view volume). Secondo si proietta il volume di vista nel sistema di coordinate normalizzate, questo in quanto alcune operazioni, come il clipping, sono più facili ed efficienti da effettuare su un cuboide con gli assi allineati.

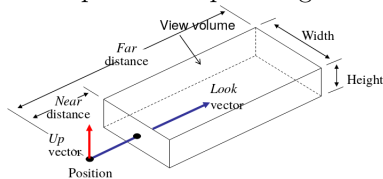


Volume di vista

Il volume di vista è costituito da un tronco di piramide, detto **frustum**. È presente un vettore di vista che attraversa il centro della piramide, mentre l'aspect ratio determina la proporzione di larghezza e altezza dell'immagine mostrata sullo schermo.



La proiezione parallela ortografica è un volume di vista troncato a forma di cuboide, questo elemento non ha un parametro per l'angolo di vista.



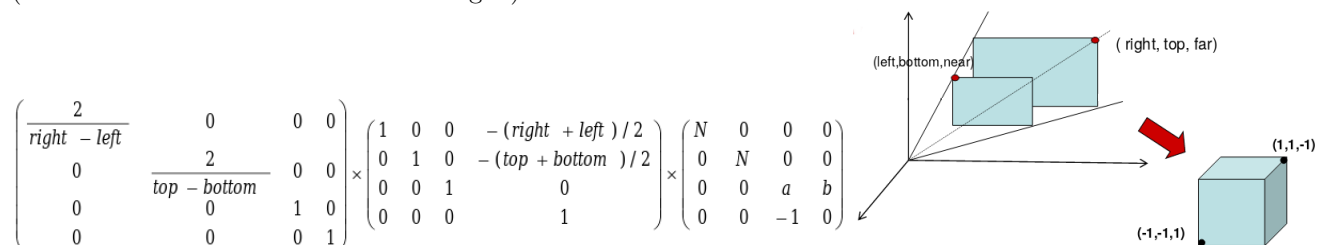
La normalizzazione trasforma il volume di vista in un volume di vista cuboide. Questa procedura permette in una singola pipeline di avere sia una visione prospettica che ortogonale.

Matrice di proiezione prospettica

La matrice di proiezione prospettica trasforma il **frustum** in un volume di vista canonico (un volume da $(-1, -1, 1)$ a $(1, 1, -1)$):

$$P(L, R, T, B, N, F) = \begin{bmatrix} \frac{2 \cdot N}{R-L} & 0 & \frac{L+R}{L-R} & 0 \\ 0 & \frac{2 \cdot N}{T-B} & \frac{B+T}{B-T} & 0 \\ 0 & 0 & \frac{N+F}{N-F} & \frac{2 \cdot N \cdot F}{F-N} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Lo scopo è mappare ogni coordinata dal range del frustum, al range $[-1, 1]$. Questa matrice è ottenuta moltiplicandone 3: una matrice di scala, una di traslazione e la precedente matrice di trasformazione prospettica (la terza colonna è stata cambiata di segno).

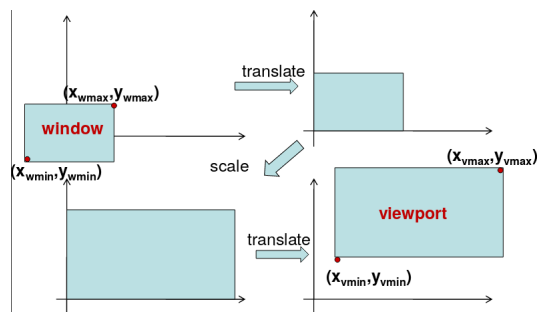


Window-Viewport Transformation

Con la Window-Viewport Transformation si adattano le dimensioni degli oggetti alla schermata dell'utente.

Questo avviene prendendo la window, che si presenta con l'origine esattamente al centro (frutto della trasformazione in cuboide), rimuovendo l'asse z e trasportandola con l'angolo in basso a sinistra sull'origine. A questo punto si ridimensiona la window in base alle proporzioni del viewport, e lo si trasla alle coordinate dello schermo in cui la finestra è stata posizionata.

Se si considera per ogni punto dell'immagine, il centro di ogni pixel, è necessario traslare sopra l'origine e a destra di mezzo pixel entrambi.



6.2 Clipping

Il **clipping** è un taglio rispetto ad un volume di vista: vengono scartati per intero gli elementi completamente fuori dal volume, accettati quelli interamente dentro e ritagliate solo le porzioni nel volume di vista per gli oggetti a cavallo. Per quanto riguarda il clipping di segmenti, si controlla la posizione degli endpoint (i punti di inizio e di fine), se sono entrambi dentro al volume di vista vengono accettati, se sono uno fuori e uno dentro avviene un clip mentre se sono entrambi fuori non si sa, perché il corpo del segmento potrebbe comunque passare all'interno del volume di vista. Per capire quando ci si trova in quest'ultimo caso di usa l'algoritmo di **Cohen-Sutherland**.

Cohen-Sutherland

Si divide il piano in 9 regioni, in cui la regione centrale rappresenta la vista. Ogni quadrante avrà un *Out Code* (OC) di 4 bit in base a dove si trova rispetto al quadrante centrale di vista.

Definiamo x_{min} la coordinata x più a sinistra del quadrante di vista, mentre x_{max} la coordinata x più a destra. Inoltre sia y_{min} la coordinata y più in basso, mentre y_{max} la coordinata y più in alto. L'OC è costruito come segue:

- Primo bit: si imposta ad 1 se $x < x_{min}$, ci si trova fuori dal lato **sinistro**.
- Secondo bit: si imposta ad 1 se $x > x_{max}$, ci si trova fuori dal lato **destro**.
- Terzo bit: si imposta ad 1 se $y > y_{max}$, ci si trova fuori dal lato **sopra**.
- Quarto bit: si imposta ad 1 se $y < y_{min}$, ci si trova fuori dal lato **sotto**.

	1001	0001	0101
y_{min}	1000	0000	0100
y_{max}	1010	0010	0110
	x_{min}	x_{max}	

A questo punto si calcolano i valori a 4 bit per i due punti p e q del segmento (OC_p , OC_q):

- Caso 1: $OC_p = OC_q = 0000$ sono entrambi dentro e viene accettato il segmento.
- Caso 2: OC_p e OC_q contengono almeno un 1 significa che risiedono fuori dal quadrante. In questo caso si esegue l'AND logico tra i due, se il valore risultante contiene almeno un 1 significa che i punti sono entrambi

fuori ma dallo stesso lato, quindi il segmento è scartato.

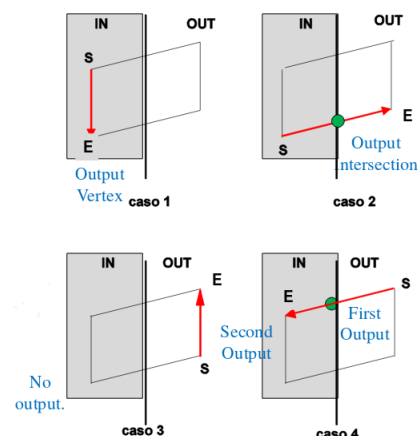
- Caso 3: OC_p e OC_q contengono almeno un 1 e il risultato dell'AND logico ritorna tutti zeri (0000), questo significa che non sono dallo stesso lato ma attraversano il quadrante di vista. A questo punto si calcolano i 2 punti di intersezione con i bordi del quadrante e si useranno come i due nuovi vertici del segmento all'interno del quadrante di vista.

Lo stesso algoritmo può essere effettuato in 3 dimensioni con l'aggiunta di 2 bit ai precedenti 4.

Clipping di poligoni

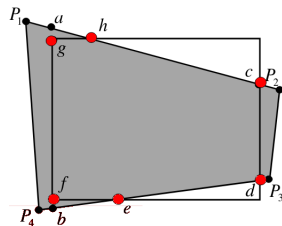
Il clipping di poligoni può generare poligoni con più lati di quelli originali, o addirittura più poligoni distinti se la forma originale era convessa. Per eseguire il clipping, ad ogni modo, si scompone il problema in 4 sotto-problemi: viene effettuato il clipping per ogni bordo della finestra. Durante l'analisi di ogni bordo vengono analizzati tutti i lati del poligono (come coppie di vertici S-E, ovvero *start* e *end*), ognuno può essere:

- Interno alla finestra (sia S che E interni).
- Intersecato verso fuori (S dentro ed E fuori).
- Esterno alla finestra (sia S che E esterni).
- Intersecato verso dentro (S fuori ed E dentro).



Sutherland-Hodgman

L'algoritmo di Sutherland-Hodgman prende in considerazione un bordo della finestra alla volta e lo prolunga all'infinito. A questo punto analizza tutti i lati del poligono per coppie di vertici S-E, se scavalcano il bordo della finestra si *aggiunge* il punto di intersezione come vertice ulteriore del poligono. In questo modo al termine si avranno tutti i vertici del poligono in corrispondenza dei bordi della finestra.



6.3 Rasterization Stage

Il processo di rasterizzazione permette, a partire da vertici, di creare frammenti, che non sono semplici pixels ma contengono oltre ad una posizione anche colore, alpha del vertice ed altre informazioni. Un metodo base prevede di partire dai vertici delle forme geometriche e calcolare i pixel in cui si trovano, dopodichè calcolare i segmenti tra le coppie di vertici, ovvero i lati del poligono, e infine riempire i pixel vuoti all'interno.

Conversione di scansione: dati due punti sullo schermo, si vuole determinare quali pixel dovrebbero essere “disegnati” tra i punti per mostrare una linea di spessore unitario. Gli algoritmi per questo procedimento sono: DDA e Midpoint Algorithm. L’obiettivo dei seguenti algoritmi è trovare il pixel successivo più vicino alla linea reale da disegnare.

- Digital Differential Analyzer (DDA)

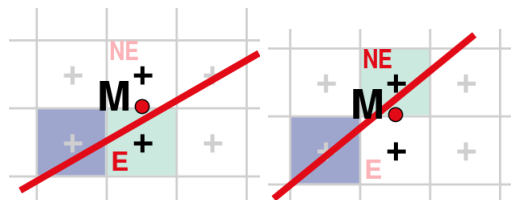
Data l’equazione della retta che connette due punti: $y = mx + B$, si parte dal punto più a sinistra e si incrementa x_i di 1 ottenendo x_{i+1} poi si valuta y_{i+1} come

$$y_{i+1} = mx_{i+1} + B = m(x_i + 1) + B = mx_i + B + m = y_i + m$$

Si colora quindi il pixel $[x, \text{round}(y)]$. Possono tuttavia sorgere problemi in caso di linee molto ripide, in cui la x del pixel successivo incrementa lentamente o non incrementa proprio (linea verticale). La soluzione è che se la pendenza è > 1 si valuta prima la y poi la x .

- Midpoint Algorithm

Si seleziona il pixel verticalmente più vicino alla linea da rappresentare. In generale ad ogni punto (in un tipo di retta crescente, come quella in figura) il pixel successivo può essere sia $E(x_p + 1, y_p)$ [EST] oppure $NE(x_p + 1, y_p + 1)$ [NORD-EST]. Si sceglie E se la linea da rappresentare passa sotto o in mezzo al punto di mezzo M , altrimenti NE se la linea passa sopra ad M .



La funzione F che permette di capire se ci si trova sotto ($F > 0$), sopra ($F < 0$) o su M ($F = 0$) è:

$$F(x, y) = dy \cdot x - dx \cdot y + dx \cdot B$$

Il punto M è in posizione $(x_p + 1, y_p + \frac{1}{2})$, quindi avremo $d = F(x_p + 1, y_p + \frac{1}{2})$ per capire se scegliere NE ($d > 0$) o E ($d < 0$). Per ottimizzare le risorse non viene valutato d calcolando ogni volta F per ogni pixel, bensì $d_{new} = d_{old} + dy$ per E e $d_{new} = d_{old} + dy - dx$ per NE .

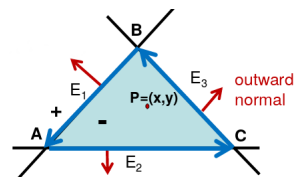
Riempimento di poligoni

La regola base per il riempimento di poligoni è: se un punto è dentro al poligono, va colorato con il colore interno, altrimenti no. Il test per capire se è dentro o fuori può essere o il *triangle test*, applicabile solo ai poligoni convessi o l’*odd-even test* applicabile a tutti i poligoni.

- Triangle Test

Un punto è interno ad un triangolo se è nella metà negativa di tutti e tre i bordi ($E_i(x, y) \leq 0$): i vertici del triangolo sono ordinati in senso antiorario, il punto deve essere sul lato sinistro di ogni bordo:

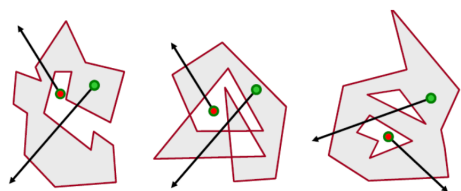
$$E(x, y) = ax + by + c = (x - x_0)(y_1 - y_0) - (y - y_0)(x_1 - x_0)$$



- Odd-even Test

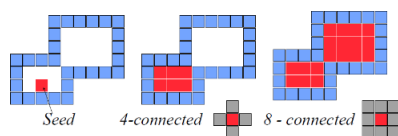
Se un punto P che punta in una qualsiasi direzione all’infinito, incrocia un numero pari di bordi allora il

punto è *interno* al poligono, altrimenti se incrocia un numero dispari è *esterno* al poligono.



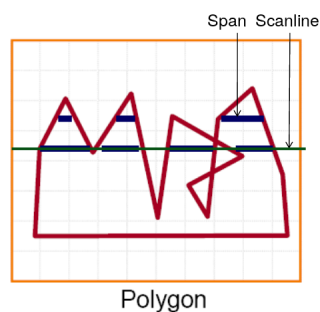
- Metodo Flood

Una volta calcolato il bordo del poligono e dato un punto iniziale (x, y) interno al poligono si possono osservare ricorsivamente i suoi vicini (4 o 8) colorandoli solo se **non** sono punti di bordo.



- Metodo Scan Line

Algoritmo incrementale per trovare gli span per ogni linea di scannerizzazione e determinare se è interno o meno, ogni span può essere processato indipendentemente. Si procede dall'alto verso il basso e da sinistra a destra. Si cercano le intersezioni della linea di scan con i bordi del poligono, si ordinano per coordinate x incrementali e si riempiono i pixel tra le coppie di intersezioni.



6.4 Fragment Stage

Colorazione pixel

Si calcolano le coordinate baricentriche di ogni punto all'interno dei triangoli $(\alpha, \beta$ e $\gamma)$ e si usano per determinare il colore in base ai colori assegnati ai vertici (A_{color} , B_{color} e C_{color}):

$$x_{color} = \alpha A_{color} + \beta B_{color} + \gamma C_{color}$$

Culling

Il culling è il processo in cui si determinano e rimuovono i poligoni non visibili, questo processo rimuove intere primitive.

Più precisamente un particolare triangolo viene rimosso se:

- Non è all'interno del view frustum (view frustum culling)
- Se non ha la faccia rivolta verso la camera (back-face culling)
- Se è degenerato (ovvero ha area = 0).

Il primo caso è frutto dell'algoritmo di clipping (già visto), il terzo caso è identificabile dalla normale $[0, 0, 0]$. Nel back-face culling per convenzione la faccia frontale del triangolo è definita come il lato in cui i vertici sono posizionati in senso antiorario.

Visibilità

- Algoritmo del pittore

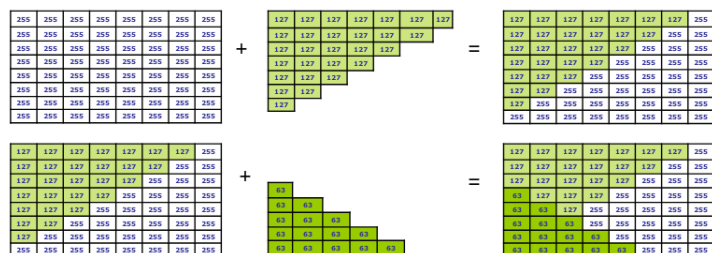
Si “disegnano” prima gli oggetti più lontani e man mano i più vicini per ultimi. Questo richiede un ordinamento in base all’asse z , anche se questo è un processo molto costoso. Tuttavia in alcuni casi l’ordinamento basato su una sola coordinata non è sufficiente e bisogna far riferimento anche alle altre due.

- Ray casting

Si fa partire un raggio dalla camera fino alla prima intersezione che trova, questo per ogni pixel, e si “disegna” il primo elemento colpito dal raggio.

- Z-Buffer Algorithm

Si usano 2 buffer della stessa dimensione aggiunti al frame buffer, uno il **color buffer**, l’altro il **depth** (“Z”) **buffer**, inizializzato a 255. I poligoni sono scannerizzati in ordine arbitrario, quando i pixel si sovrappongono si usa lo Z-buffer per decidere quale poligono verrà disegnato in quel pixel.



7 Illuminazione

Le due componenti fondamentali dell’illuminazione sono: le *sorgenti di luce* (colore, geometria, potenza) e le *proprietà del materiale* (colore, geometria, assorbimento). Quando un fascio di luce colpisce un oggetto, la luce viene parzialmente *assorbita*, *rifratta* e *riflessa* (questa definisce il colore dell’oggetto). Vi sono due modelli di illuminazione:

- **Locale**: il colore di una superficie dipende solo dalla sorgente luminosa, ignorando altri oggetti vicini.
- **Globale**: il colore di una superficie dipende sia dalla sorgente luminosa che dal riflesso degli altri oggetti in scena.

7.1 Sorgenti luminose

- **Luce ambiente**: rappresenta una sorgente di luce con intensità fissa che colpisce equamente tutti gli oggetti. Proviene da tutte le direzioni e viene utilizzata per illuminare uniformemente l’intera scena, simulando una luce indiretta.
- **Luce puntiforme**: viene indicata con un punto $P = [x, y, z, 1]$ e di default emana luce in tutte le direzioni con intensità costante.
- **Luce direzionale**: illumina tutti gli oggetti equamente, ma da una direzione specifica. Viene spesso posizionata molto lontano per simulare sorgenti di luce distanti come il sole. È rappresentata da un vettore $P = [x, y, z, 0]$.
- **Spotlight**: la luce è emessa da un singolo punto nello spazio e si diffonde a cono il cui apice è un punto P_s che punta in una direzione I_s e la cui larghezza è determinata da un angolo Θ . L’intensità di luce è concentrata nel centro del cono e si attenua del tutto in Θ e $-\Theta$.

7.2 Modello di illuminazione di Phong

Il modello considera solo la luce proveniente direttamente dalle sorgenti luminose (chiamata luce diretta) e non dalla luce riflessa dagli oggetti adiacenti. Equazione di Phong in un punto della superficie:

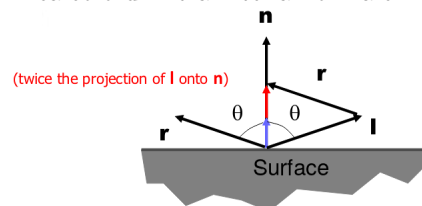
$$I_\lambda = k_e + I_a k_a + I_d k_d + I_s k_s$$

In cui i k_* rappresentano le proprietà del materiale, in ordine: emissività, ambiente, diffusione e specularità. Tutti questi valori sono compresi nell'intervallo $[0, 1]$. Mentre i I_* rappresentano l'intensità della luce in base alle varie proprietà.

- Componente **emissiva**: rappresenta la luminosità che il materiale emana (es. sole), per calcolarla: $I = k_e$.
- Componente **ambiente**: rappresenta un'illuminazione non specifica della scena che viene emanata in tutte le direzioni per garantire un'illuminazione minima, per calcolarla: $I = I_a k_a$.
- Componente **diffusiva**: rappresenta la quantità di luce riflessa equamente in ogni direzione da un punto sulla superficie. Questa dipende dall'orientamento della superficie (si usa la normale \mathbf{n}) e dalla direzione della luce (si usa il vettore che punta alla sorgente \mathbf{l}), per calcolarla: $I = I_d k_d = k_d I_l (\mathbf{l} \cdot \mathbf{n})$.
- Componente **speculare**: una superficie a specchio riflette la luce nella sola direzione di riflessione ideale \mathbf{r} che è ottenuta riflettendo \mathbf{l} (direzione della luce) rispetto ad \mathbf{n} (normale alla superficie). Questa dipende quindi anche dalla provenienza dell'osservatore (vettore \mathbf{v}), per calcolarla: $I = I_s k_s = k_s I_l (\mathbf{v} \cdot \mathbf{r})^{n_s}$. In cui n_s è il grado di brillantezza del materiale (più aumenta, più la luce riflessa viene concentrata in una piccola regione attorno al riflettore perfetto).

Calcolo del vettore riflesso \mathbf{r}

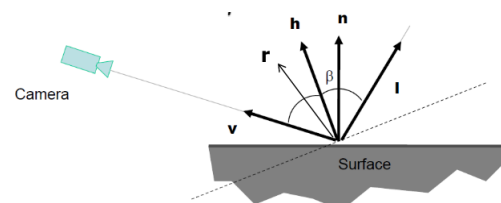
Il calcolo di \mathbf{r} tramite la normale \mathbf{n} e la sorgente luminosa \mathbf{l} avviene tramite: $\mathbf{r} = 2 (\mathbf{n} \cdot \mathbf{l}) \mathbf{n} - \mathbf{l}$.



Semplificazione: modello di illuminazione di Blinn-Torrance

Invece che calcolare il vettore riflesso \mathbf{r} per la componente speculare, si calcola il vettore \mathbf{h} che rappresenta l'esatta metà tra \mathbf{l} e \mathbf{v} e si utilizza al posto di \mathbf{r} per la componente speculare. È meno preciso ma più efficiente da calcolare:

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|}$$

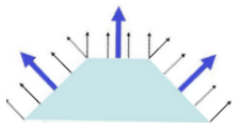


7.3 Shading

La fase di shading avviene nella **Rasterization Stage** in quanto effettuata sui frammenti, a differenza dell'illuminazione che avviene nella **Geometry Stage** in quanto effettuata sui vertici.

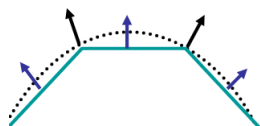
- Flat Shading

Si calcola l'illuminazione in un solo vertice per ogni poligono della mesh e si usa questo valore per ogni pixel appartenente al poligono. È ovviamente poco accurato per superfici lisce. Per far ciò si calcola la normale ad ogni triangolo e si spalma su ogni punto della primitiva. Si potrebbe pensare che se un oggetto è realmente formato da sole facce primitive (es. un cubo) questo metodo sia adeguato, invece no in quanto si suppone \mathbf{l} , \mathbf{n} e \mathbf{v} siano costanti sul poligono, invece anche solo \mathbf{l} (direzione della luce) in caso di luce puntiforme varia per ogni singolo punto della primitiva. Oppure per quanto riguarda il riflesso speculare, la direzione all'occhio \mathbf{v} varia all'interno di ogni singola primitiva.



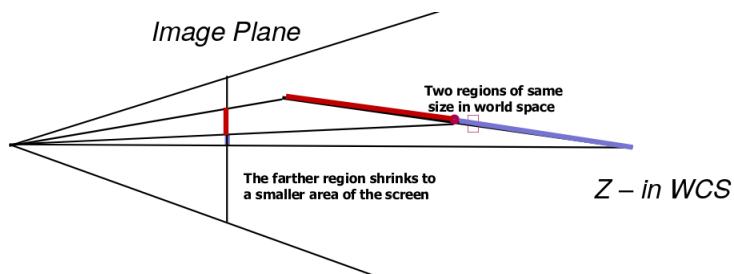
- Gouraud Shading

Il Gouraud shading è molto più accurato del precedente, ma presenta comunque problemi. Per prima cosa si determina il vettore normale medio in ogni vertice del poligono (facendo la media tra le normali dei poligoni che condividono quel vertice). Successivamente si applica un modello di illuminazione ad ogni vertice ottenendone l'intensità. Ed infine si effettua un'interpolazione lineare delle intensità dei vertici per determinare quelle dei frammenti interni al poligono.



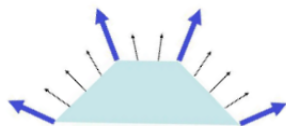
Questo modello è molto comune ed implementato anche da OpenGL: le normali sono calcolate in ogni vertice (in VCS), l'intensità in ogni vertice è calcolata usando la normale e un modello di illuminazione (Phong), nella fase di rasterization per ogni poligono i valori di intensità per i pixel interni sono calcolati per interpolazione lineare delle intensità nei vertici.

Un problema è che l'interpolazione dei colori è effettuata nello Screen Space e non corrisponde all'interpolazione lineare in WCS, la proiezione prospettica di fatto altera l'interpolazione lineare. La soluzione è calcolare lo shading nell'Object Space e non nello Screen Space, oppure applicare una correzione prospettica.



- Phong Shading

Vengono calcolate le normali in ogni vertice, e si interpolano per calcolare le normali di ogni pixel nella faccia per poi applicarvi il modello di illuminazione di Phong (prima venivano interpolate le intensità, qui vengono interpolate le normali).

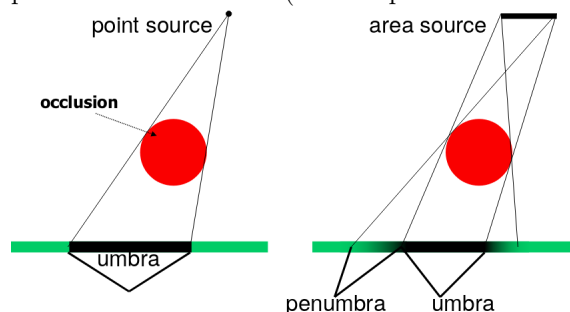


Un problema di questo modello di shading è che è molto costoso a livello computazionale e l'illuminazione avviene dopo la proiezione prospettica. Infine le silhouettes irregolari degli oggetti rimangono sia con Gouraud che con Phong.



7.4 Ombre

L'ombra di un oggetto dipende dalla sua forma e dalla posizione della sorgente luminosa. Le ombre si dividono in *soft* e *hard*; le ombre soft sono generate da sorgenti di luce estese, e sono composte da: umbra (la sorgente è completamente occlusa) e penumbra (la sorgente è parzialmente occlusa); le sorgenti luminose puntiformi producono ombre hard (in cui è presente solo umbra).



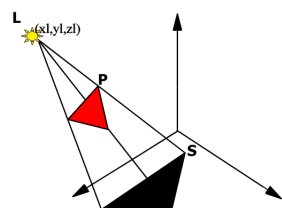
Un modo per produrre queste ombre consiste o nel simulare aree di luce con molti punti luce (molto costoso), oppure applicare un effetto blur all'immagine proiettata dell'oggetto (poco costoso ma inaccurato).

Per le ombre su *superfici piane* si disegnano le primitive dell'oggetto una seconda volta, e si proiettano sul piano. La sorgente puntiforme è il **Center of Projection** (COP).

1. Algoritmo di Blinn

Si disegna la proiezione dell'oggetto sul piano, il poligono ombra è la proiezione del poligono sulla superficie con COP nel punto luce **L**. Per ogni vertice **P** dell'oggetto un raggio ombra passa dal punto luce **L** nel piano, intersecando il poligono ombra **S**. Una limitazione è che non produce ombre sull'oggetto stesso, se è concavo.

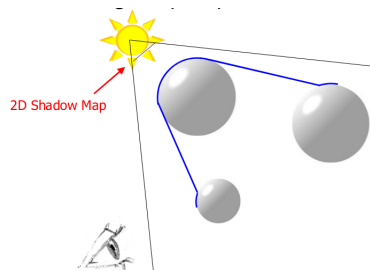
$$\begin{bmatrix} x_s \\ y_s \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\frac{x_l}{z_l} & 0 \\ 0 & 1 & -\frac{y_l}{z_l} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$



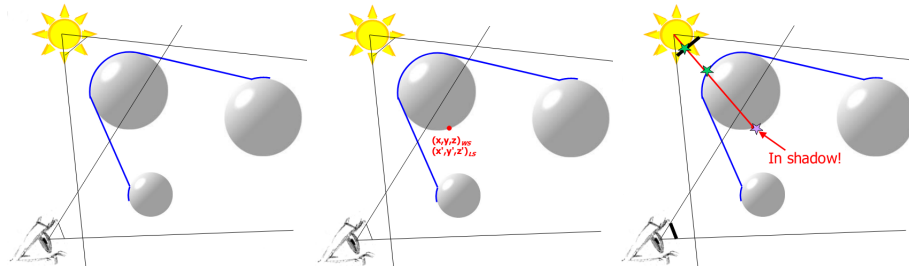
2. Algoritmo Shadow Map

Un punto è illuminato se è visibile dalla sorgente luminosa, il calcolo dell'ombra è simile al calcolo di vista (vengono rimosse le superfici nascoste). Per questo algoritmo è necessario un buffer (shadow z-buffer) per ogni sorgente luminosa, la procedura utilizza due passaggi lungo la pipeline:

a) calcola la **mappa dell'ombra**: si effettua il primo rendering della scena usando la sorgente luminosa come punto di riferimento della view, poi si salva il risultato nello shadow Z-buffer.



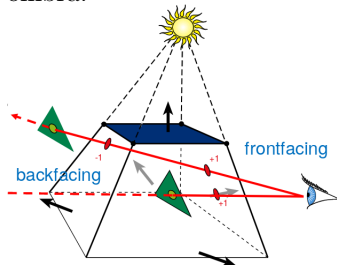
b) si effettua il **rendering dell'immagine finale**: si esegue il secondo rendering della scena usando l'algoritmo Z-buffer e controllando la mappa dell'ombra per vedere se i punti sono in ombra.



Le shadow-map generano una texture dell'ombra catturando le silhouettes degli oggetti, visti dalla sorgente luminosa, poi proietta la texture sulla scena. Lo svantaggio è che va ricalcolata l'ombra se la luce si muove.

3. Volumi Ombra

Si rappresenta esplicitamente il volume dello spazio in ombra, poi si esegue uno **shadow test**, simile al clipping. Si lancia un raggio dall'occhio al punto visibile e si incrementa e decrementa un contatore ogni volta che viene intersecato il volume dell'ombra. Se il contatore al termine è diverso da 0 allora il punto è in ombra.



7.5 Trasparenza

Quando si rasterizza un triangolo semi-trasparente si ha prima di tutto una quantità alpha per ogni pixel. Poi si ha il colore di ogni pixel fornito dall'interpolazione dei vertici con Gouraud, chiamato source color, questo è il colore dell'oggetto semi-trasparente in foreground. Si ha infine il colore del pixel in background, chiamato destination color. Il colore finale è dato banalmente dalla formula:

$$c_{final} = \alpha c_{src} + (1 - \alpha) c_{dst}$$

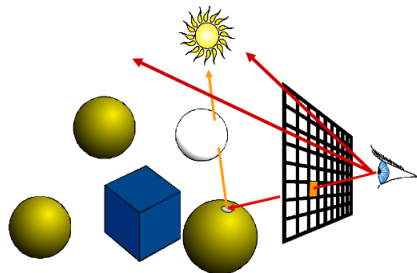
Il **rendering** avviene in 3 passaggi:

1. Si effettua il rendering degli oggetti opachi in scena (il depth buffer rigetta gli oggetti oscurati da altri).
2. Si effettua il rendering degli oggetti trasparenti in scena in base all'alpha e alla profondità.
3. Si ordinano gli elementi trasparenti in base alla loro posizione e si effettua il blending per ogni pixel.

8 Ray Tracing

Nel **ray tracing base** è sufficiente mappare solo i raggi che raggiungono l'osservatore, quindi invece che calcolare in avanti una quantità infinita di raggi dalla sorgente di luce all'oggetto e poi fino all'osservatore, si mappa al contrario un numero finito di raggi dall'osservatore verso ogni direzione che rimbalzeranno eventualmente su oggetti, fino a raggiungere la sorgente luminosa.

Si genera un raggio primario, ovvero un raggio che parte dall'occhio attraverso il centro di un pixel, e si cerca l'oggetto più vicino lungo il percorso del raggio: la prima intersezione tra il raggio e un oggetto in scena può essere una *sorgente di luce*, un *oggetto* o *niente*, in quest'ultimo caso si imposta il colore di background.

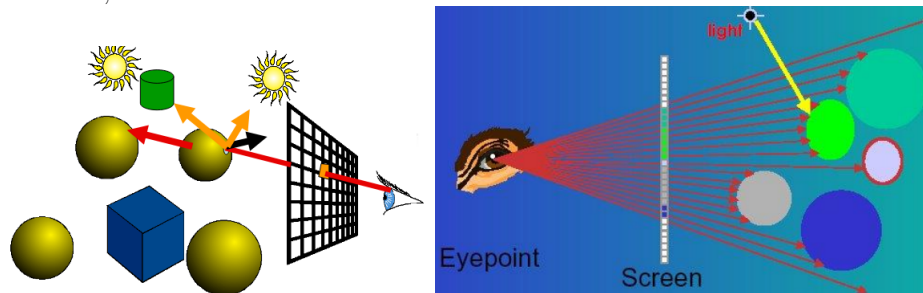


Una volta che si hanno le informazioni sull'intersezione, si applica un modello di illuminazione per determinare il contributo diretto da parte delle sorgenti luminose (**ray casting**), è fondamentalmente quello che fa la pipeline di rendering. Per una maggiore accuratezza, si generano ricorsivamente raggi secondari per catturare i contributi indiretti forniti dai riflessi di altri oggetti, i quali avranno anch'essi contributi indiretti (**ray tracing**).

8.1 Ray Casting

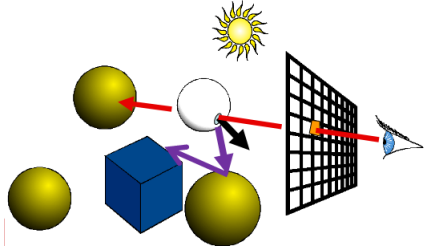
L'algoritmo di Ray Casting ha complessità $O(n \cdot m)$ con n numero di oggetti ed m numero dei pixel. Ogni luce in scena fornisce un contributo di colore e intensità a un elemento nella superficie. Si costruisce allora un raggio da una superficie verso ogni sorgente di luce, chiamato **shadow ray** e si controlla se il raggio intersezione altri oggetti prima che di raggiungere la luce. Se non ne intersezione nessuno (né altri, né se stesso), allora viene contato l'intero contributo della luce. Se un raggio invece intersezione altri oggetti prima di raggiungere la luce, non viene contato nessun contributo da parte della luce.

Se non sono presenti oggetti trasparenti in scena con il ray casting si ottiene la rimozione delle superfici nascoste, ombre e illuminazione locale.



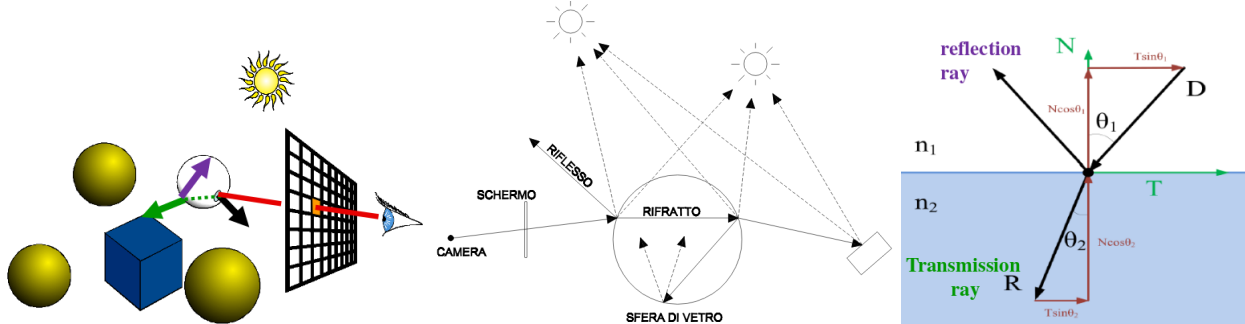
8.2 Riflessione

La luce può eseguire anche percorsi più complessi che colpire semplici superfici solide e rimbalzare, può ad esempio rimbalzare molte volte sulle superfici prima di raggiungere la luce o il vuoto. Si introduce quindi il **reflection ray**, si parte dal punto di intersezione e si invia un raggio verso una riflessione speculare, questo processo può essere ripetuto ricorsivamente. La direzione di riflessione avviene con lo stesso angolo alla normale del raggio in arrivo, ma nel verso opposto (il calcolo è stato visto nella sezione Illuminazione).



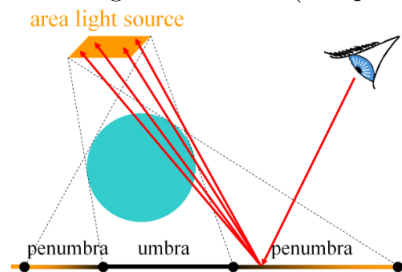
8.3 Rifrazione

Il ray tracing può anche essere usato per renderizzare la luce in superfici trasparenti, tenendo conto della rifrazione della luce quando passa attraverso un oggetto. Un **transmission ray** viene generato quando un raggio colpisce la superficie di un oggetto trasparente. La rifrazione causa il cambiamento della direzione del raggio trasmesso, la quantità di rifrazione è calcolata usando l'indice di rifrazione dell'oggetto. All'interno di un oggetto quando il raggio rifratto colpisce la superficie interna viene prodotto un altro refraction ray ma anche un reflection ray, questo processo continua ricorsivamente. L'indice di rifrazione è dato da: $\frac{\sin\theta_1}{\sin\theta_2}$, ovvero angolo di incidenza diviso angolo di rifrazione.



8.4 Ombre

Se si vogliono calcolare le ombre si lancia un raggio aggiuntivo per ogni sorgente luminosa. Una luce può contribuire al colore finale solo se non colpisce nulla tra il punto e la sorgente luminosa. Se si vogliono calcolare le soft shadows si lanciano raggi multipli (in posizioni random) che puntino all'interno dell'area della sorgente luminosa (non possibile per luci puntiformi).

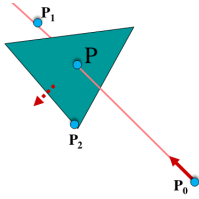


8.5 Ricorsione

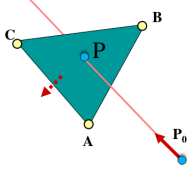
Il ray tracing può essere eseguito ricorsivamente, i criteri per terminare la ricorsione possono essere o la **profondità di ricorsione** (dopo un certo numero di rimbalzi il raggio si ferma), o in base al **contributo del raggio** (quando l'attenuazione della trasparenza/trasmissione è troppo bassa si ferma).

Il calcolo dell'**intersection ray**, ovvero il raggio che interseca un oggetto è la fase più costosa.

1. **Intersezione con un piano:** data l'equazione parametrica di un raggio: $P(t) = P_0 + t(P_1 - P_0) = (1 - t) P_0 + t P_1$ con $t \in [0, \infty)$ bisogna trovare $t = \frac{n \cdot (P_1 - P_0)}{n \cdot (P_1 - P_0)}$ per trovare il punto di intersezione $P = P(t)$ del raggio con il piano.



2. **Intersezione con un triangolo:** si calcola la coordinata baricentrica di P rispetto al triangolo ABC come $P(\alpha, \beta, \gamma) = \alpha A + \beta B + \gamma C$, se si hanno $0 < \alpha, \beta, \gamma < 1$ allora P appartiene al triangolo.



3. **Intersezione con una sfera:** qui si usa l'equazione implicita della sfera centrata nell'origine ($c = 0$), $(P(t) - c)^2 = r^2$ e l'equazione parametrica del raggio, $P(t) = P_0 + t(P_1 - P_0) = P_0 + t r_d$, si sostituisce e si risolve l'equazione quadratica per t . Si avranno due soluzioni se il raggio attraversa in mezzo la sfera, una se sfiora un bordo e soluzione immaginaria se il raggio non tocca la sfera.

9 Texturing

9.1 Texture mapping

Il texture mapping avviene basandosi sul Texture Coordinate System (TCS), da qui nella fase di *parametrizzazione* si passa dal Texture Space all'Object Space e infine nella fase di *proiezione* dall'Object Space allo Screen Space.

Parametrizzazione

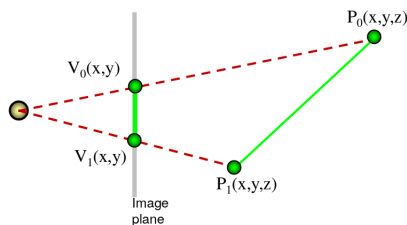
In questa fase si applica la texture ad un oggetto, definendo una funzione M che associa ogni punto (x, y, z) della superficie ad un punto 2D nella texture map (s, t) , ovvero: $M(x, y, z) = (s, t)$.

Questo può avvenire "automaticamente" come nel caso di superfici parametriche (spline), o fornito "manualmente" dall'utente assegnando una coordinata texture ad ogni vertice. La parametrizzazione ideale per le mesh dovrebbe essere *isometrica*, ovvero lunghezze e angoli preservati, una parametrizzazione *conforme* preserva solo gli angoli.

Il **genere** (genus) di una superficie indica il numero di "buchi" che ha la superficie se venisse aperta, es. una sfera ha $\text{genus} = 0$, un toro ha $\text{genus} = 1$, due tori fusi insieme ha $\text{genus} = 2$, ecc. Se una mesh ha genere > 0 la si suddivide in porzioni di texture separate che verranno poi affiancate per creare il risultato finale. Il texture mapping avviene dopo la rasterizzazione, questo è particolarmente efficiente perchè solo pochi poligoni superano la fase di clipping.

Proiezione

Nella fase di proiezione può nascere un problema chiamato **distorsione della texture**. Se si considera supponiamo un lato di un triangolo mappato con una texture in World Space e lo si proietta in Screen Space, si parte da una linea retta di punti regolarmente spazati tra loro ad una linea retta di punti spazati in modo *irregolare* tra loro (usando la proiezione prospettica), nell'immagine saranno più concentrati in basso e più radi in alto.



Perciò per ogni pixel si calcolano le coordinate baricentriche in Screen Space, poi si mappa da Screen Space a World Space e infine si interpolano gli indici texture (u, v) .

Two Step Mapping

Con questo algoritmo il mapping avviene in due fasi:

1. S Mapping: prima si *mappa* la texture su una superficie semplice intermedia (es. piano, cilindro, sfera, cubo).
2. O Mapping: poi si *proietta* la texture dalla superficie intermedia all'oggetto più complesso. Questa fase può avvenire in 4 modi: si applicano le normali dall'oggetto intermedio a quello reale, il contrario, si usano vettori dal centro dell'oggetto intermedio o si usano raggi riflessi (si traccia un raggio dalla camera all'oggetto e si calcola il punto dell'oggetto intermedio colpito dal raggio riflesso).

9.2 Bump mapping

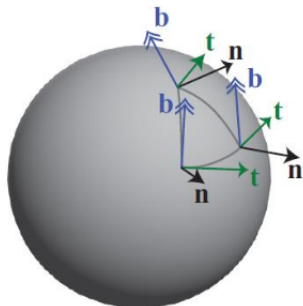
La differenza tra *bump mapping* e *displacement mapping* è che il primo va a modificare solo le normali, mentre il secondo cambia l'intera superficie (comprese le silhouette).

Una **bump map** è una **height map** su un singolo canale (scala di grigi) usata per modificare la direzione della normale della superficie di un oggetto. Si calcola la normale N in un punto della superficie, si moltiplica al valore del punto nella bump map e si somma il tutto al valore del punto già presente: $P'(u, v) = P(u, v) + B(u, v) N$.

Una **normal map** è un'immagine RGB in cui i valori RGB di ogni texel rappresentano i valori x, y e z di un vettore direzione usato per modificare il vettore normale della superficie. I valori espressi nell'intervallo $[0, 1]$ vengono convertiti nell'intervallo $[-1, 1]$.

Texture/Tangent Space (TBN)

TBN è un sistema di coordinate attaccato alla superficie locale e definito per ogni punto (vertice) della superficie dai vettori: **tangente** (T), tangente alla superficie, **bi-tangente** (B), tangente alla superficie ma sotto di essa (ortogonale a tangente e normale), **normale** (N), perpendicolare alla superficie.



Ci sono fondamentalmente due modi di usare una matrice TBN per il normal mapping:

- Trasformare le normali campionate da *Tangent Space* a *Object/World Space* usando la matrice TBN, in questo modo la normale viene spostata nello stesso spazio delle altre variabili di illuminazione (in cui è utilizzata).

- Prendere l'inversa della matrice TBN che trasforma ogni vettore dall'*Object/World Space* al *Tangent Space* ed usare questa matrice per trasformare le variabili di illuminazione rilevanti (luce e posizione di vista) in *Tangent Space*.

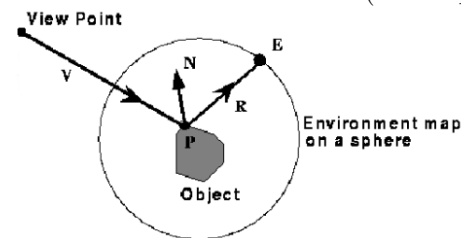
9.3 Procedural texturing

Una **texture map procedurale** è una funzione che prende in input le coordinate texture (s, t, r) e restituisce in output colore, opacità e shading. Un esempio è il legno, la seguente funzione produce un cilindro a "strati": $f(s, t, r) = s^2 + t^2$.

Per rendere le texture procedurali più interessanti si aggiunge *rumore*, si generano in modo casuale dei valori nel range $[-1, 1]$ e si interpolano tra loro (creando un effetto "sfumato" e non salti netti). Un buon modo per modellare il rumore è creare un array 3D di valori pseudo-random e interpolarli, un modo migliore è creare un array 3D di vettori in 3 dimensioni e usare spline cubiche per interpolarli. Modificando la *turbolenza* si amplifica o meno il rumore generato: $Turbulence(x) = \sum_{i=0}^k \frac{|noise(2^i x)|}{2^i}$

9.4 Environment mapping

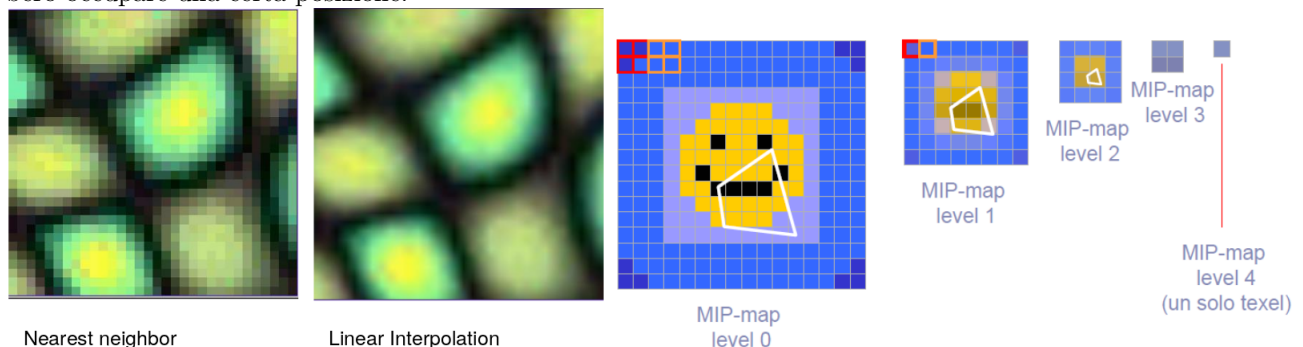
La creazione di un'**environment map** prevede due passaggi: il *preprocessing*, che prevede la creazione della mappa renderizzando la scena senza l'oggetto riflettente, con la camera al centro dell'oggetto; e il *rendering*, che avviene usando la texture (env-map) indicizzata dal vettore di riflessione.



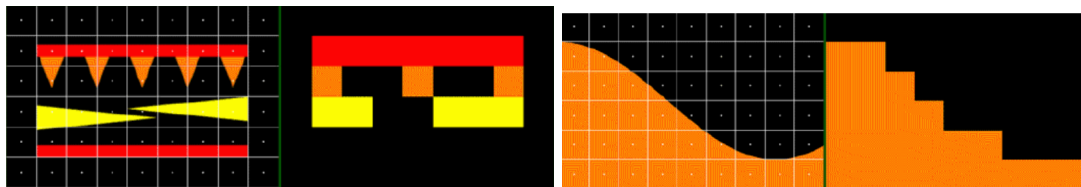
Cube mapping

Un'**environment map** cubica consiste in 6 mappe che coprono il cubo. Si usano 6 rendering con la camera al centro dell'oggetto che punta verso ognuna delle 6 direzioni. Quando si effettua il rendering, una coordinata indica la faccia guardata, mentre le altre due la posizione del pixel nella texture.

Possono sorgere dei problemi quando si parla di **rimpicciolimento** o **ingrandimento** della texture, in questi casi i pixel potrebbero risultare sfasati. Nel secondo caso per evitare di mostrare grossi pixel distinti tra loro si applica un'*interpolazione* lineare tra essi generando un effetto più "sfumato" e meno netto. Per il primo caso quando molti pixel dovrebbero sovrapporsi sullo stesso si applica una tecnica di MIP mapping (Multum In Parvo, molte cose in un piccolo spazio) ovvero un'*interpolazione* trilineare dei valori che dovrebbero occupare una certa posizione.

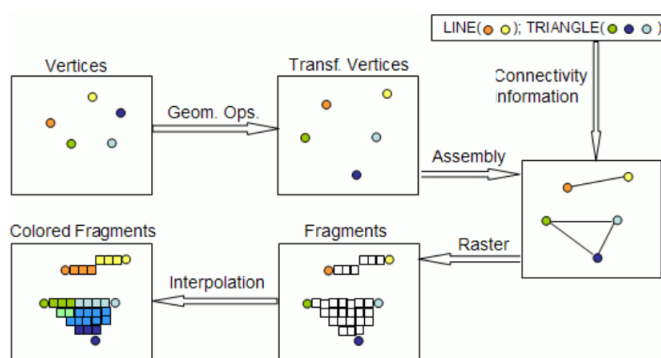


L'**aliasing** è la perdita di dettaglio nel rendering:



Come **antialiasing** è possibile: per il *ray tracing/rasterization* renderizzare ad una risoluzione più alta, applicare un *blur*, e renderizzare ad una risoluzione più bassa (effettuando il resampling). Anche per le textures è possibile applicare un blur prima del lookup.

10 Shaders



CPU - Application: specifica gli oggetti geometrici attraverso insiemi di vertici e attributi che vengono inviati alla GPU.

GPU - Vertex processor: le posizioni dei vertici e le normali vengono trasformate dalla matrice Model-View in coordinate eye, viene calcolata la luce per ogni vertice e generate/trasformate le coordinate texture. Viene restituito in output il colore del vertice e la sua posizione in 2D dopo a proiezione prima della viewport.

GPU - Rasterization: avviene in due fasi: *assemblamento delle primitive*, ovvero i vertici sono assemblati in oggetti, vengono proiettati ed applicato il clipping e trasformati secondo il viewport. La seconda fase, *fragment generation*, in cui entità geometriche sono rasterizzate in frammenti ognuno dei quali corrisponde a un pixel.

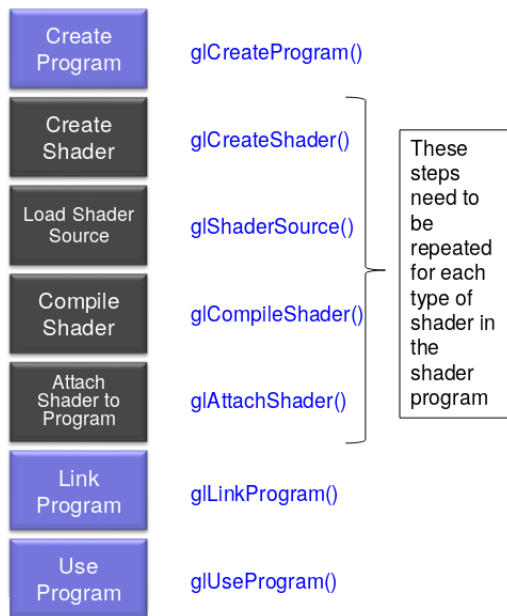
GPU - Fragment processor: prende in input l'insieme di frammenti prodotti, effettua delle operazioni sui frammenti come calcolo del colore, applicare texture, normale, illuminazione per pixel e in output restituisce il colore finale.

10.1 GLSL

- Variabili

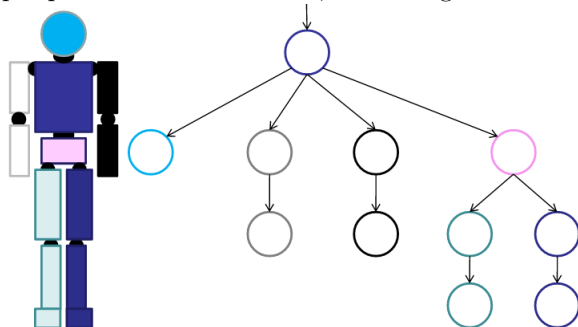
in e **out** copiano gli attributi del vertice e di altre variabili da e verso altri shaders.

uniform variabili globali in sola lettura che possono cambiare in base alle primitive e che vengono passate dall'applicazione OpenGL agli shaders, sono usate per condividere dati tra programma applicativo, vertex e fragment shaders.



11 Animazione digitale

Un tipo di animazione avviene tramite **figure articolate**, ovvero insiemi di giunzioni e ossa. Questo tipo di animazione può essere modellata attraverso la **modellazione gerarchica**, un modo di rappresentare le parti di un oggetto e le giunzioni connettive tra esse. Per rappresentare oggetti complessi attraverso parti più piccole si usa un albero, in cui i figli ereditano le *trasformazioni* applicate al padre.



Ogni arco dell'albero contiene due matrici: M_{pos} , ovvero la trasformazione per inserire un nodo correlato al nodo parent e M_{art} ovvero la trasformazione di articolazione del nodo.

Ogni nodo dell'albero contiene una funzione di disegno ed una matrice M_{loc} , ovvero la trasformazione del nodo in una posizione pronta per essere articolata.

Per disegnare una figura articolata è necessario attraversare l'albero, per far ciò ci sono due modi: esplicitamente (*stack-based*), in cui si usa lo stack per memorizzare attributi e matrici, o ricorsivamente (*tree-based*) usando una struttura dati che rappresenta la gerarchia ed attraversando l'albero ricorsivamente in modalità depth-first.

11.1 Animazioni basate sulla fisica

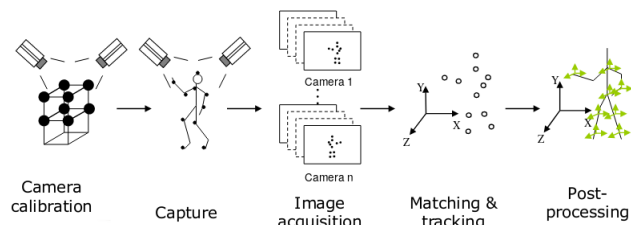
Questo tipo di animazioni sono adatte per oggetti di grande volume come vento, liquidi, fumo, tessuti. I meccanismi usati sono solitamente *sistemi particellari* e *sistemi mass-spring*.

Un movimento reale può essere modellato tramite: **cinematica**, ovvero lo studio del movimento senza considerare le forze che l'hanno causato o **dinamica**, ovvero la simulazione di corpi rigidi utilizzando forze e momenti per controllarne il movimento.

I tessuti ad esempio non sono altro che sistemi particellari in cui ogni particella è collegata ad altre 12 tramite legami elastici.

11.2 Motion Capture (MOCAP)

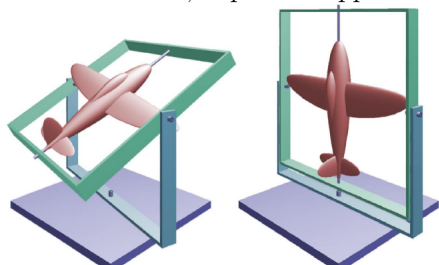
Questo tipo di rilevazione può essere di due tipi: elettromagnetica o ottica. La pipeline ha i seguenti passaggi: **calibrazione** dei reference points sul soggetto, **cattura**, ovvero il campionamento dei punti nel tempo, **ricostruzione** in 3D dei punti sottoforma di angoli articolari, **adattamento** allo scheletro, ovvero usare le congiunzioni per controllare il modello virtuale e infine avviene il **post-processing**.



11.3 Keyframes

Descrivere il movimento degli oggetti nel tempo a partire da una serie di posizioni chiave dell'oggetto. Prima si definiscono i **key frames**, assegnando valori ai parametri che caratterizzano il modello per ogni key frame. Poi si calcolano i frame **inbetween** per ottenere la continuità del movimento, tramite l'*interpolazione* dei valori per ottenere i frame intermedi.

Il **gimbal lock** è un fenomeno problematico dei giroscopi causato dall'allineamento di due assi rotanti verso la stessa direzione. Il blocco causa la perdita di un grado di libertà corrispondente all'asse bloccato. Nella foto sotto, dopo aver applicato la rotazione si perde un grado di libertà.



Cinematica diretta

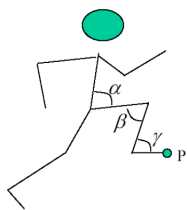
Descrive la posizione di ciascuna parte della struttura articolata assegnando valori ai parametri (angolo giunzione) per ogni posizione chiave e interpolando il giunto tra le posizioni chiave.

$$P = f(\alpha, \beta, \gamma)$$

Cinematica inversa

Specifica la posizione della parte terminale della struttura. L'animatore non indica come ogni parte separata della struttura deve muoversi, tutti gli angoli articolari vengono quindi calcolati in modo che l'effetto finale sia posizionato come richiesto.

$$\alpha, \beta, \gamma = f^{-1}(P)$$



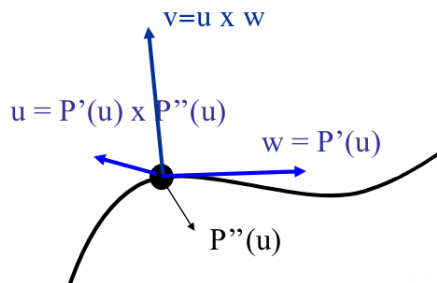
Lo **skinning** è il processo nel quale si collegano parti di una mesh a parti dello scheletro. Per un effetto più realistico è possibile che alcune parti della mesh sia collegate a più "ossa" contemporaneamente ma con pesi differenti.

12 Camera path

12.1 Frenet frame

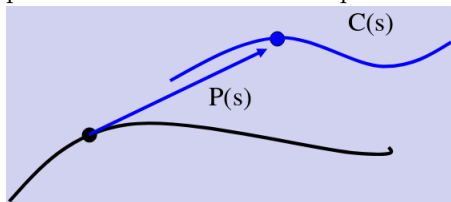
Le **formule di Frenet** descrivono le proprietà cinematiche di una particella che si muove lungo una curva continua e differenziabile nello spazio euclideo tridimensionale \mathbb{R}^3 , o le proprietà geometriche della curva stessa indipendentemente da qualsiasi movimento. Più specificamente, le formule descrivono le derivate dei cosiddetti vettori di unità tangente, normale e binormale in termini l'uno dell'altro.

- **w**: direzione di vista, è la direzione tangente alla curva.
- **u**: vettore ortogonale a w , ortogonale al piano definito da tangente e curvatura.
- **v**: up-vector ortogonale a w , ortogonale al piano definito da tangente e u .



12.2 Center of Interest

Si imposta un *centro di interesse* che verrà sempre "guardato" dalla camera durante il suo tragitto. È possibile inoltre costruire un percorso anche per il COI, oltre che per la camera, i due andranno di pari passo.



12.3 Look ahead

Semplicemente il COI è sul percorso che sta seguendo la camera, precisamente la posizione della camera più una breve distanza: $COI(s) = P(s + ds)$

