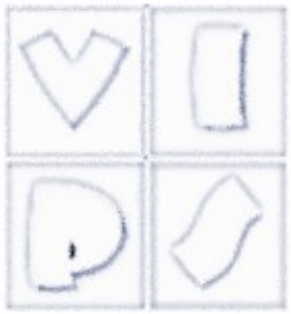
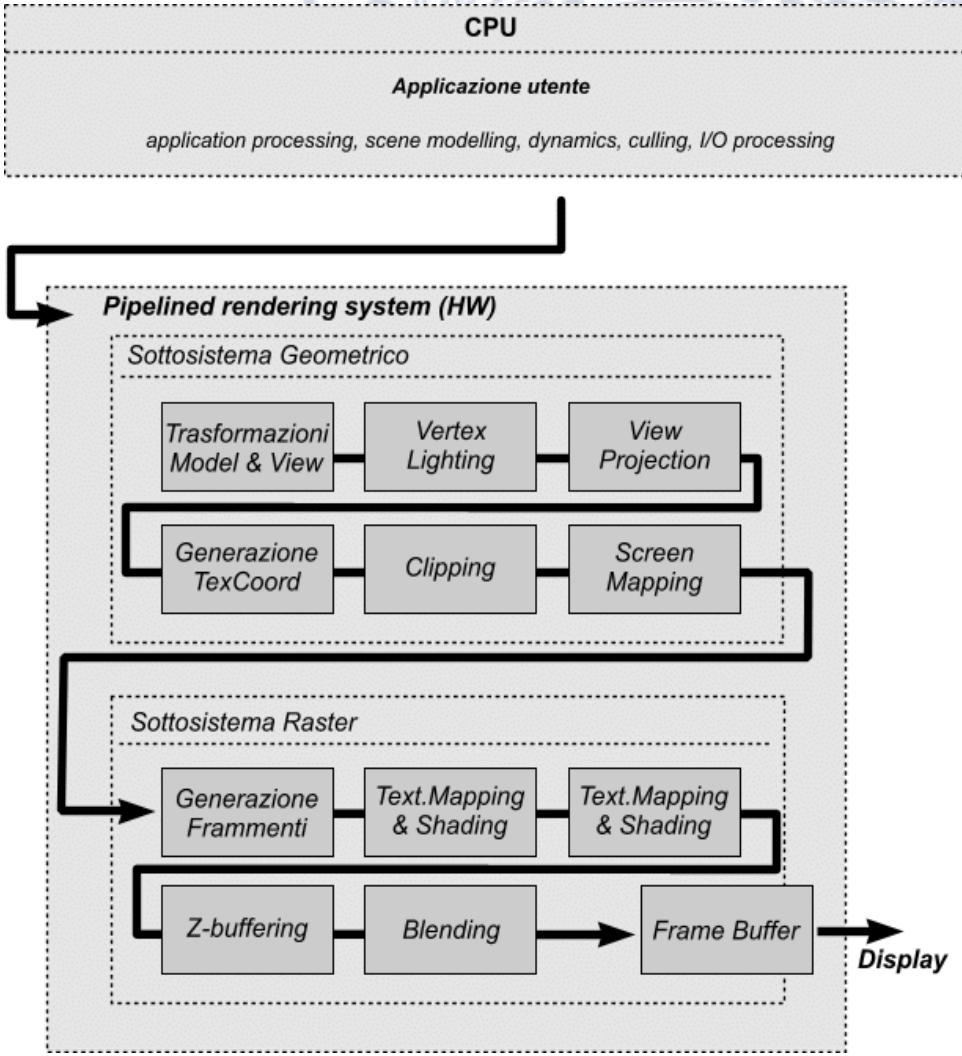


Grafica al calcolatore
-
Computer Graphics

8 – Pipeline di rasterizzazione - 2

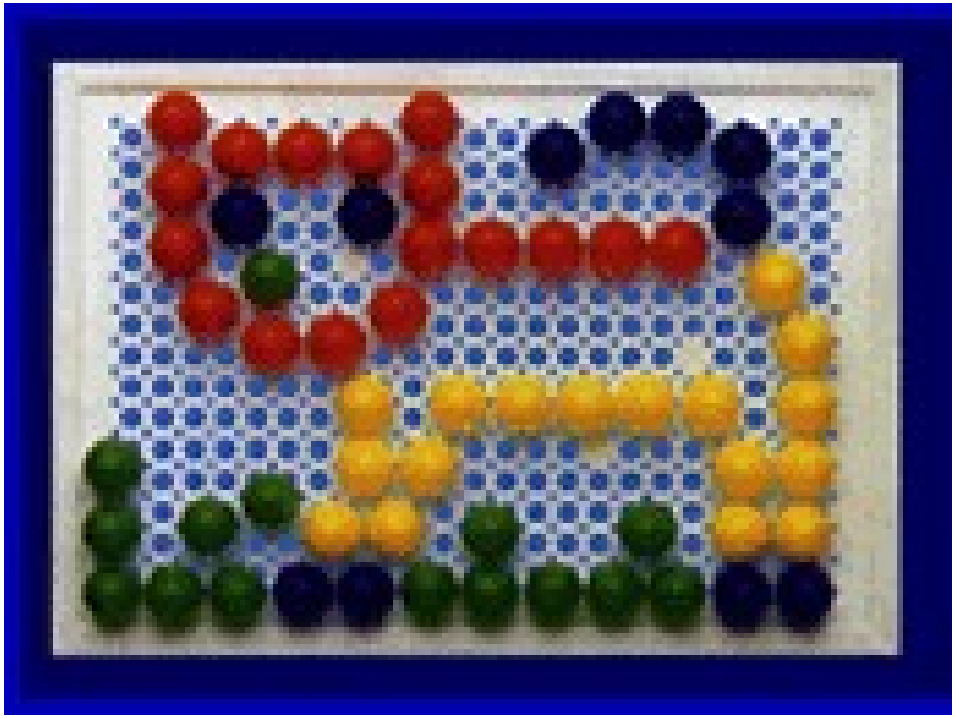
Pipeline grafica

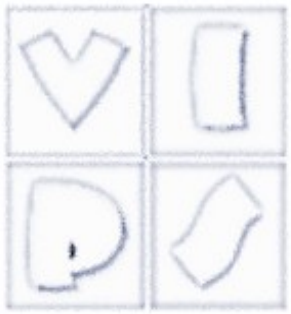
- Operazioni geometriche
- Pixel processing



Rasterizzazione/scan connversion

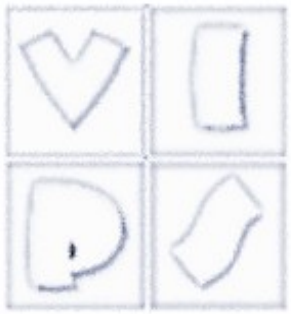
- Con il termine rasterizzazione si intende propriamente il processo di discretizzazione che consente di trasformare una primitiva geometrica definita in uno spazio continuo 2D nella sua rappresentazione discreta, composta da un insieme di pixel di un dispositivo di output. Spesso però si usa come sinonimo dell'intera pipeline





Scan conversion

- Il termine scan conversion è sinonimo di rasterizzazione in senso stretto
  - Nella pipeline dobbiamo stabilire per ciscun triangolo su che pixel esso si proietta
  - Viene generato un frammento (fragment) per ogni pixel interessato (anche solo parzialmente) da un triangolo.
- Un frammento è la più piccola unità in cui viene discretizzato un poligono. Normalmente viene generato un frammento per ciascun pixel, ma ci possono essere anche più frammenti
- La scan conversion assegna a ciascun frammento le proprietà posedute dai vertici (pseudo)profonfità normali, colore, texture, . . .) interpolandole.

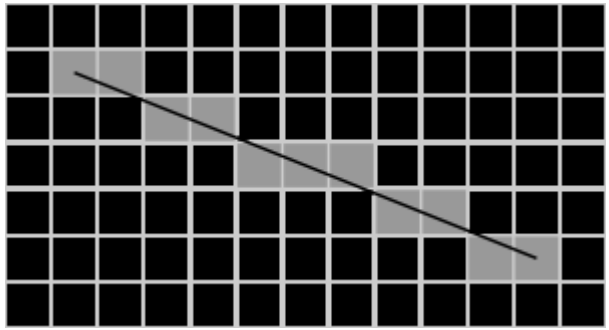


Shaders

- Da notare che nella pipeline standard (Fixed Functions) si usano funzioni limitate e fisse per calcolare il colore sui vertici e interpolare sui frammenti generati e combinare il tutto
- Oggi però schede programmabili e shader
  - Si possono creare funzioni più complesse per processare i vertici (vertex shaders) e i frammenti (fragment o pixel shaders)
  - Anche geometry shaders per modificare le geometrie
  - Oggi stesso hardware per i differenti shaders
  - GLSL linguaggio di programmazione ufficiale degli shaders in opengl

Mappare sui pixel

- Stabilire quali pixel fanno parte di un triangolo comporta la risoluzione dei seguenti due problemi:
  - Determinare i pixel dei lati (segmenti)
  - Determinare i pixel interni al poligono
- Algoritmo di Bresenham. Il classico algoritmo per disegnare un segmento è noto come algoritmo di Bresenham. Esso genera una sequenza connessa di pixel. Dopo avere disegnato un pixel l'algoritmo sceglie tra i suoi 8-vicini quale accendere in base all'equazione della retta, usando solo aritmetica intera.



Bresenham algorithm

- L'algoritmo più intuitivo per accendere i pixel sotto un segmento di retta è il Digital Differential Analyzer (DDA)
- Prendi endpoint (X0,Y0) (X1,Y1)
- Calcola differenze
  - dx = X1 - X0
  - dy = Y1 - Y0
- Se dx > dy incremento unitariamente dx, calcolo incrementi unitari su x e accendo il pixel più vicino a y=x0+(dy/dx)k
- Ma si può fare meglio

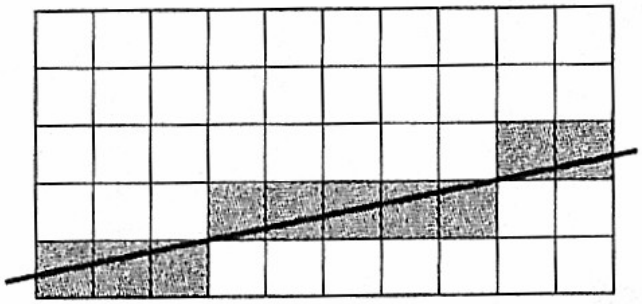


Figure 8.41 Pixels generated by DDA algorithm.

Bresenham algorithm

- Evita calcoli float ed agisce incrementalmente
- L'algoritmo di Bresenham (sempre supponendo m<1) calcola variabili di decisione per decidere quale dei due possibili pixel accendere alla x seguente
  - a e b distanze tra retta e centri (i+1, j) and (i+1, j+1)
  - Devo accendere il più vicino
  - $d = (x_2 - x_1)(a - b) = \Delta x(a - b)$,

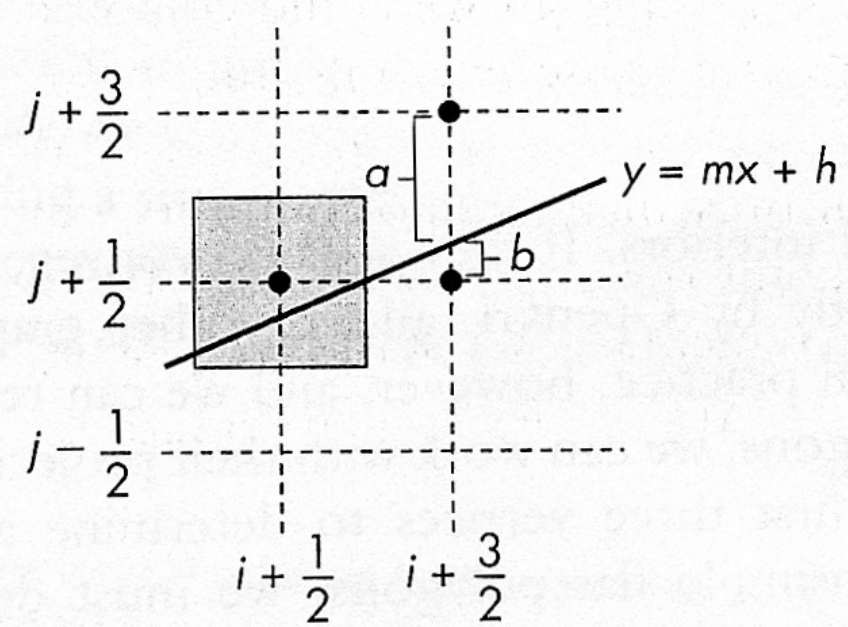


Figure 8.45 Decision variable for Bresenham's algorithm.

Algoritmo

BRESENHAM'S LINE DRAWING ALGORITHM
(for $|m| < 1.0$)

Input the two line end-points, storing the left end-point in (x_0, y_0)

Plot the point (x_0, y_0)

Calculate the constants Δx, Δy, $2\Delta y$, and $(2\Delta y - 2\Delta x)$ and get the first value for the decision parameter as: $d_0 = 2\Delta y - \Delta x$ *(intero)*

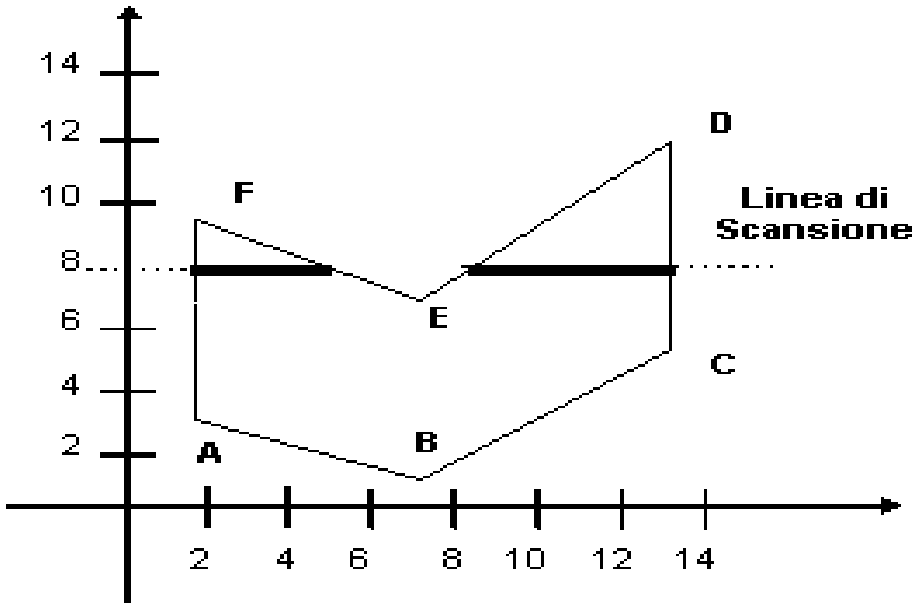
At each x_k along the line, starting at $k = 0$, perform the following test.

- If $d_k < 0$, plot (x_k+1, y_k) and: $d_{k+1} = d_k + 2\Delta y$
- Else plot (x_k+1, y_k+1) and: $d_{k+1} = d_k + 2\Delta y - 2\Delta x$

Repeat $(\Delta x - 1)$ times

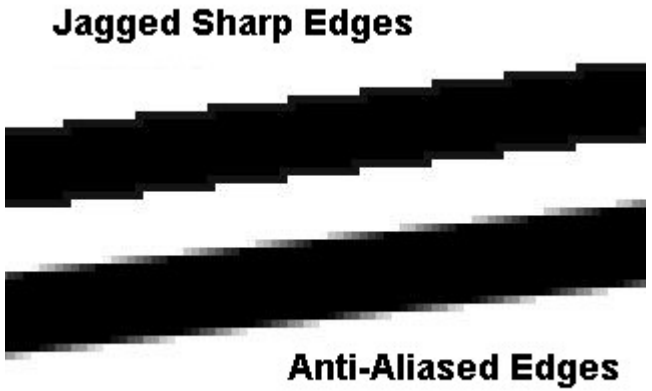
Riempire i poligoni

- L'algoritmo scan-line è l'algoritmo standard per riempire i poligoni. Il poligono viene riempito considerando una linea che lo scandisce riga dopo riga dal basso verso l'alto.
- Per ciascuna riga si effettua una scansione da sinistra a destra, e quando si incontra un edge del poligono si inizia a riempire, quando si incontra un altro edge si smette. Ci sono casi speciali da gestire con accortezza.

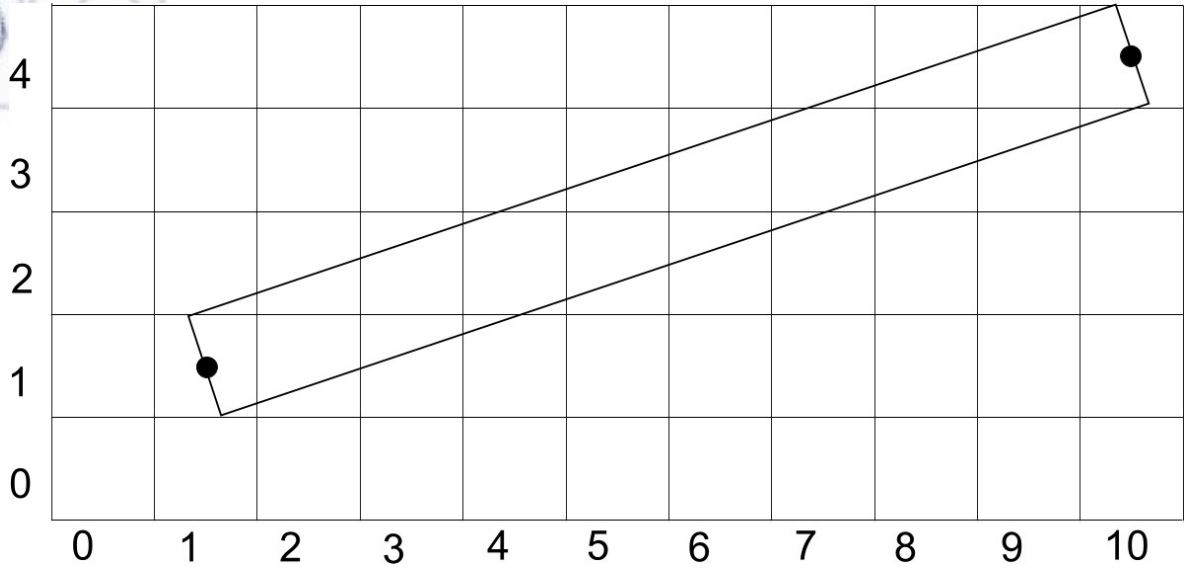


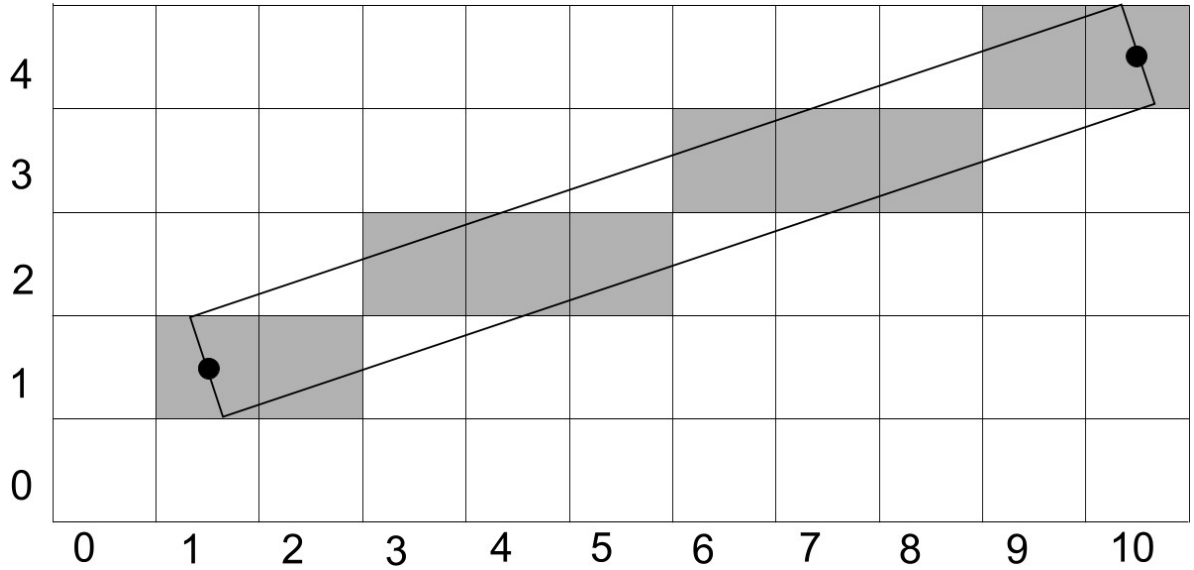
Tecniche di antialiasing

- Le tecniche di rasterizzazione producono l'effetto indesiderato di segmenti o bordi di poligoni non rettilinei ma con andamento a scaletta. L'effetto prende il nome di aliasing e deriva dal fatto che la frequenza di campionamento (dipendente dalla dimensione del pixel) è troppo piccola rispetto al segnale.
- Aliasing dipende da:
  - Numero di pixel finito;
  - Posizione dei pixel prefissata;
  - Forma e dimensione dei pixel prefissata



Tecniche di antialiasing - aliasing





Antialiasing

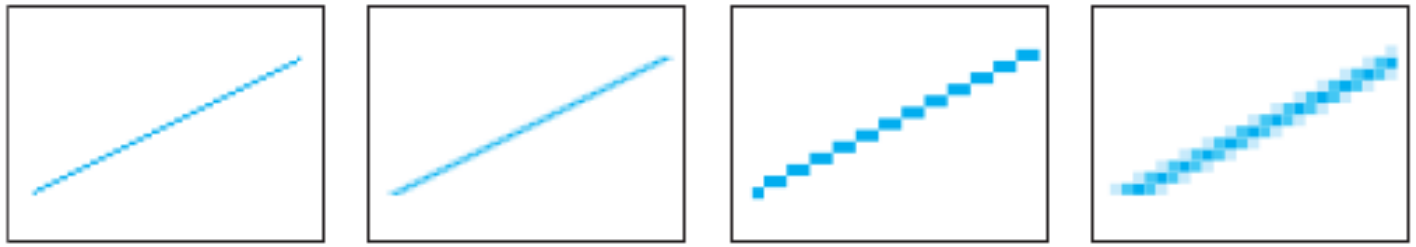
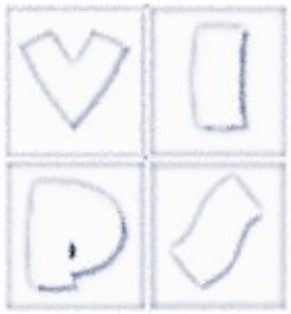


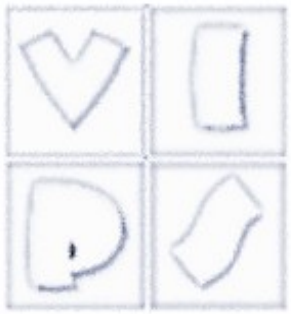
FIGURE 6.60 Aliased versus antialiased line segments. (a) Aliased line segment. (b) Antialiased line segment. (c) Magnified aliased line segment. (d) Magnified antialiased line segment.

- Idealmente si dovrebbe fare "area averaging" pesando il colore con l'intersezione della figura
- Si può fare con i fragment shaders (usando float)
  - Maggior costo computazionale



Supersampling

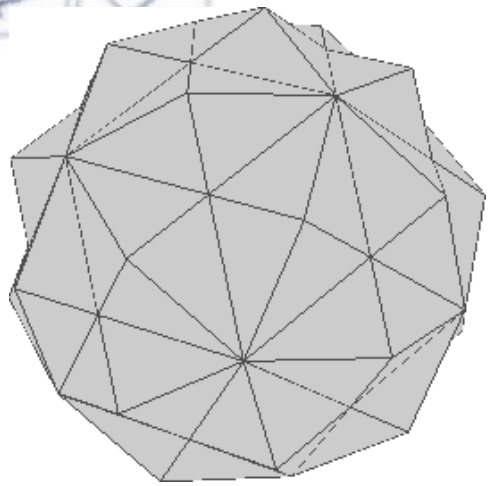
- Si ipotizza una risoluzione della memoria di quadro superiore a quella reale;
- Ogni pixel è idealmente suddiviso in 4 oppure 16 sub-pixel;
- Ad ogni pixel è assegnata una intensità proporzionale al numero di sub-pixel on.
- Approssima l'algoritmo Unweighted Area Sampling:
  - Si può fare anche weighted: cioè si pesa l'area per la distanza dal centro del pixel



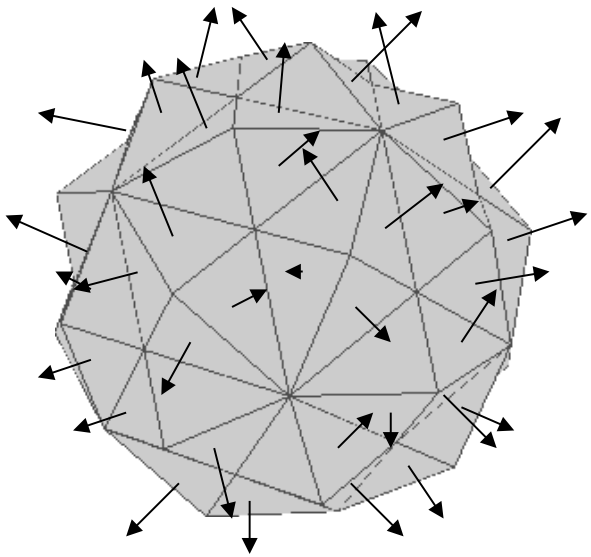
Shading interpolativo

- Supponiamo di aver mappato i vertici e i triangoli
- Dobbiamo calcolare il colore del frammento
  - Posso applicare un modello di illuminazione
  - Devo calcolare le normali
  - Posso calcolare un colore per triangolo (shading costante) e assegnare quello a tutti i pixel
  - In genere si utilizza un approccio interpolativo: il modello di illuminazione viene applicato ai vertici dei poligoni (vertex shading), ed i rimanenti pixel sono colorati per interpolazione (pixel shading).
  - Il vertex shading viene fatto nello stadio di geometric processing.

Shading costante

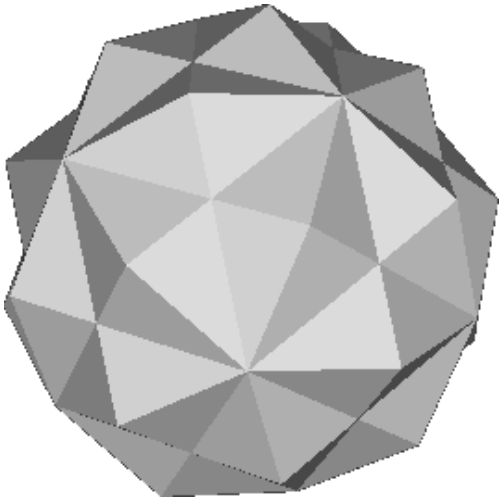


Dato l'oggetto per cui calcolare l'equazione di illuminazio I l …



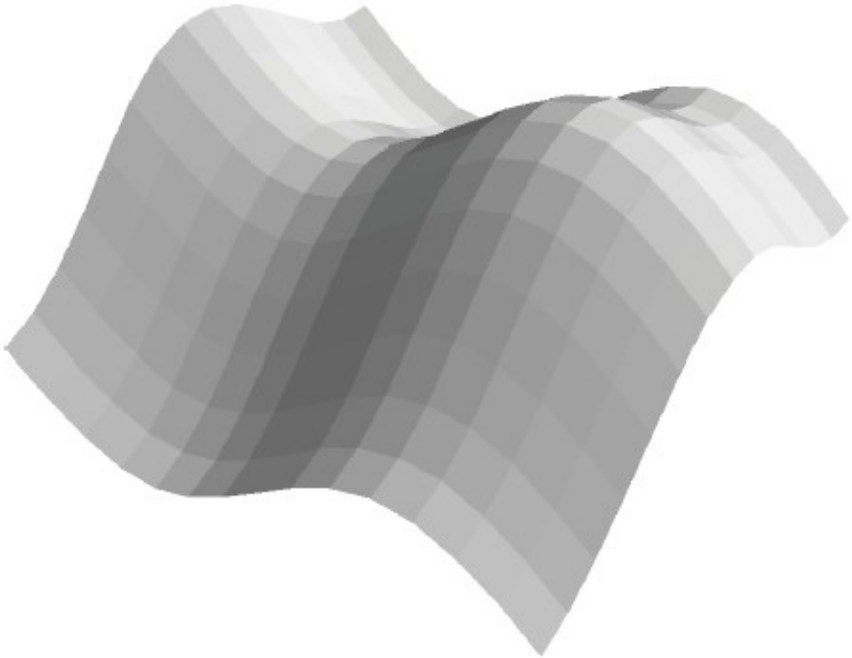
…calcolare le normali in ogni faccia…

…e calco I l una sola volta per faccia

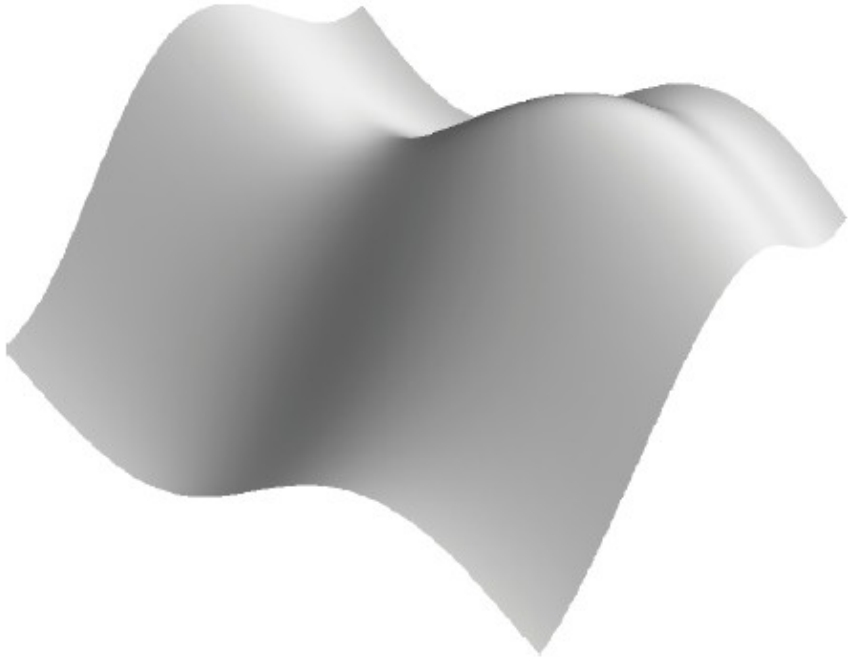


Shading costante

- Problema: il modello discreto rappresenta in modo approssimato una superficie curva e continua



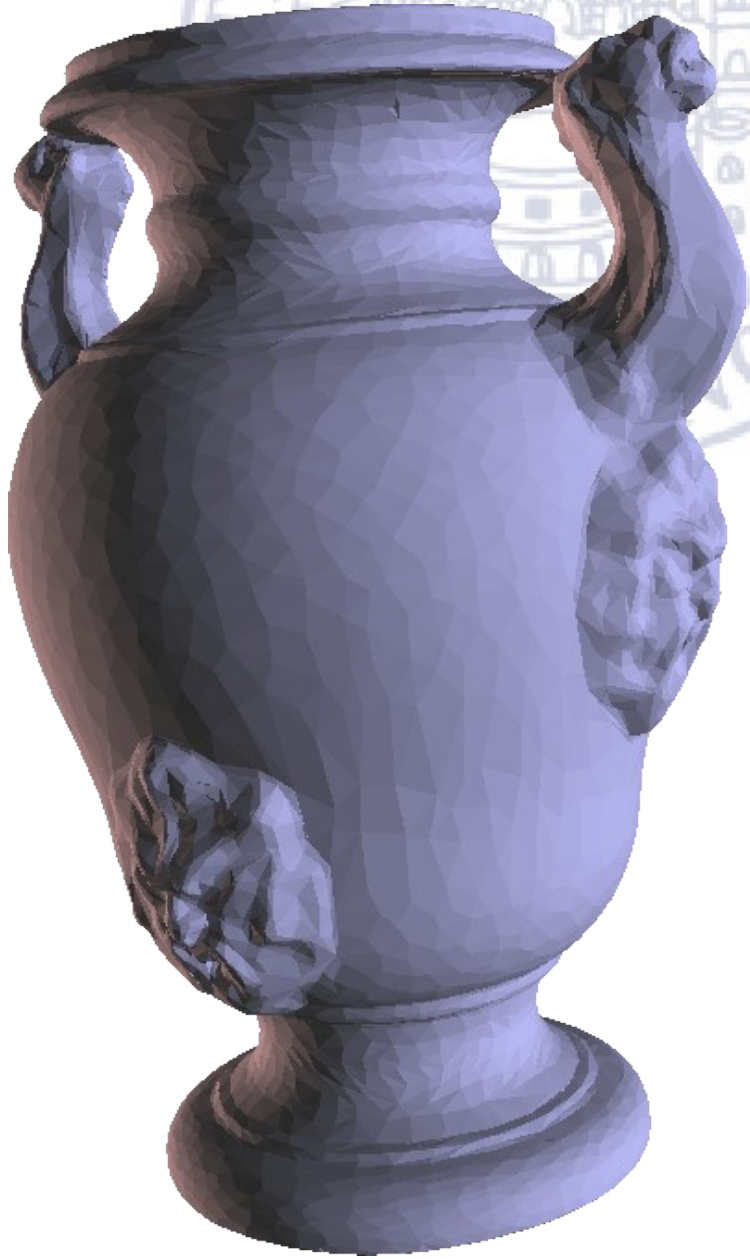
Com'è



Come dovrebbe essere

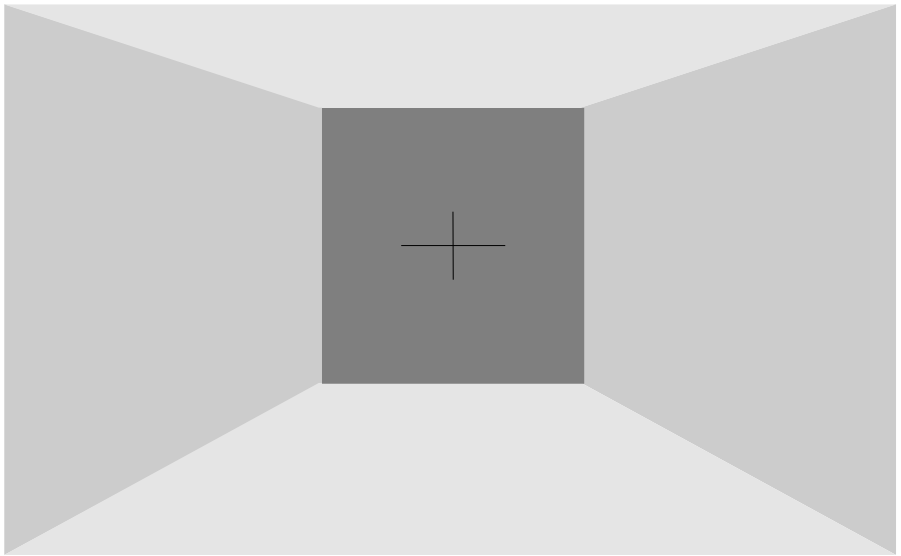
Shading costante

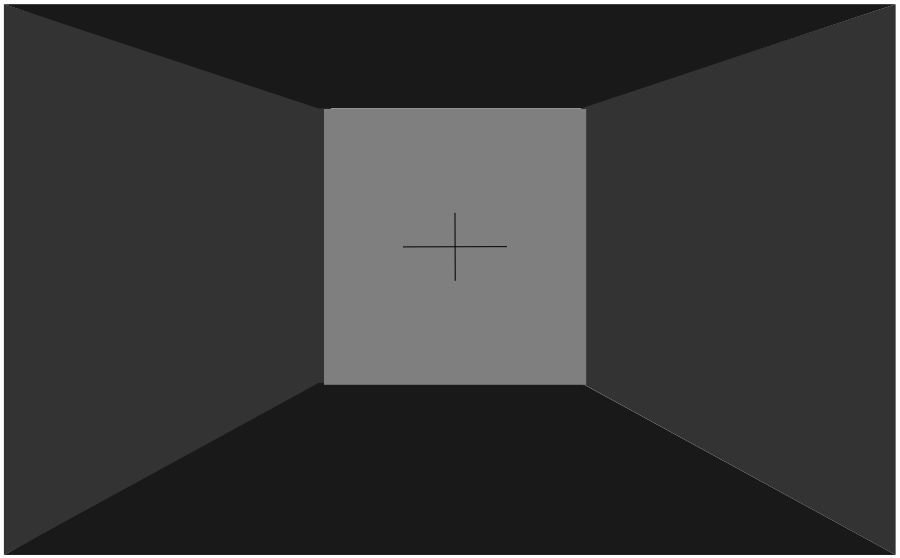
- Soluzione: uso un numero elevato di facce
- Non funziona, si vedono comunque le discontinuità tra una faccia e la vicina

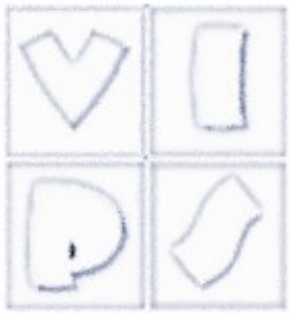


Mach banding

- Alterazione della percezione visiva di una zona in cui la luminanza varia rapidamente
- Un oggetto messo vicino ad uno più chiaro risulta più scuro e messo vicino ad uno più scuro risulta più chiaro





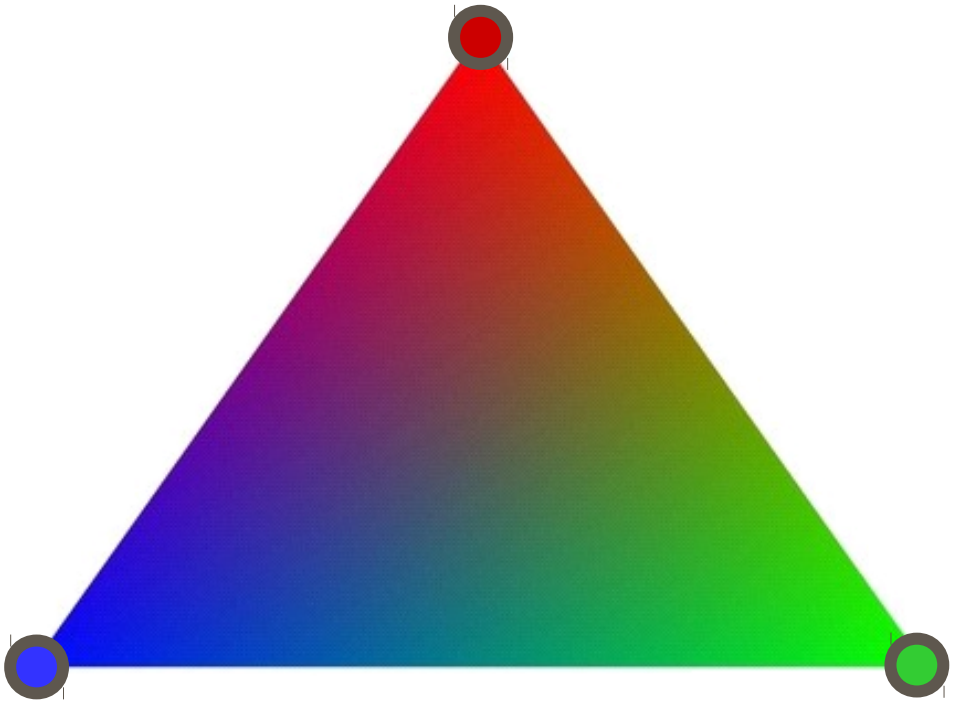


Gouraud shading

- Proprietà fondamentale dello spazio colore RGB: linearità
- Il valore colore intermedio tra due colori dati nello spazio RGB si calcola per interpolazione lineare
- Interpolazione separata sulle tre componenti R, G, e B

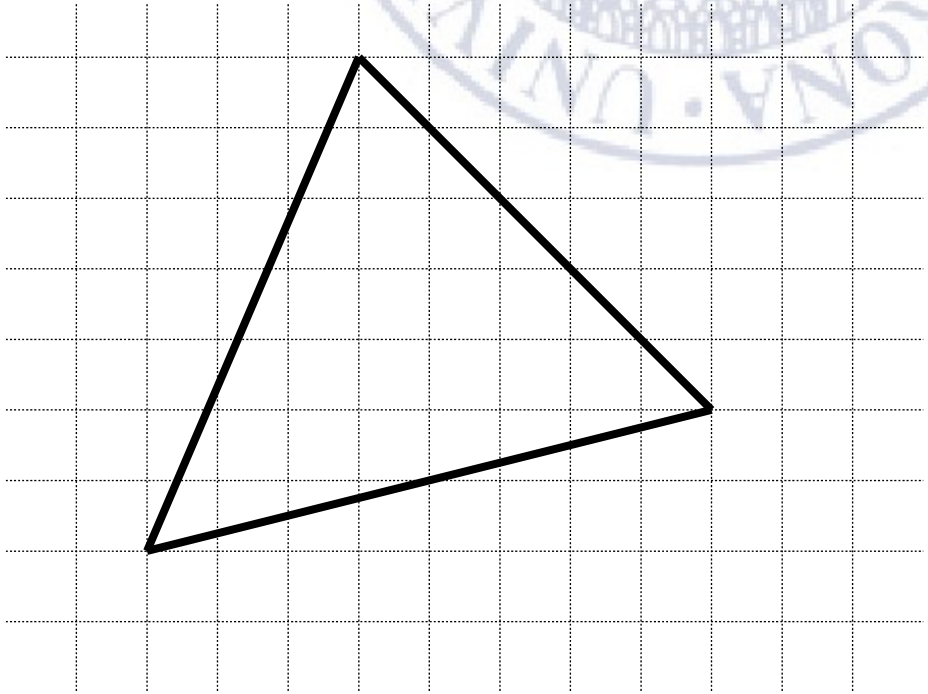
Gouraud shading

- Calcolare l'equazione di illuminazione solo in alcuni punti nodali
- Interpolare linearmente tra questi valori



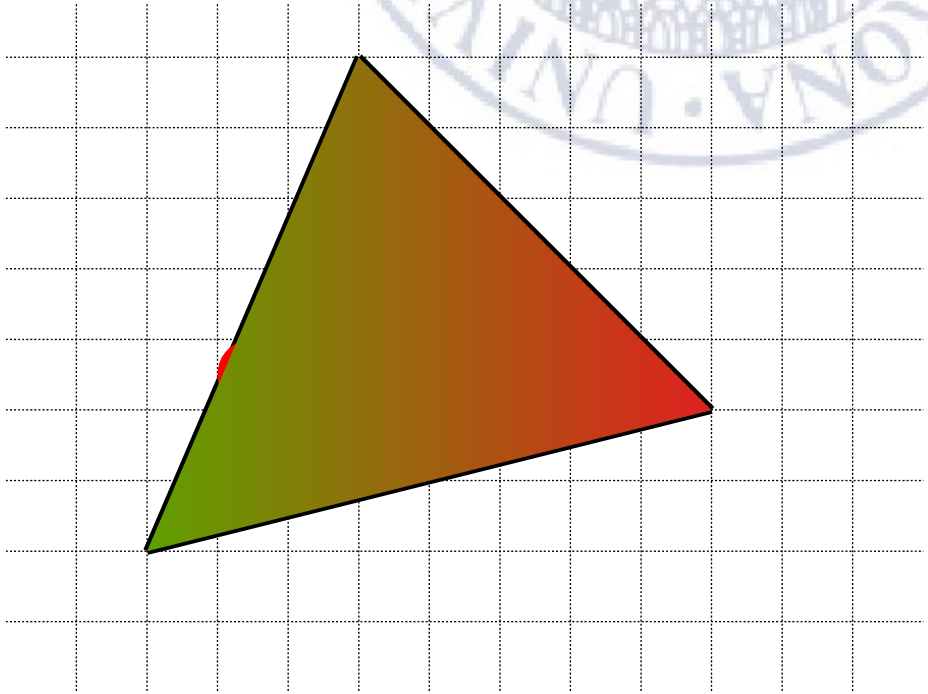
Gouraud shading

- Aggiungere all'algoritmo di rasterizzazione l'operazione di interpolazione nello spazio colore comporta uno sforzo minimo



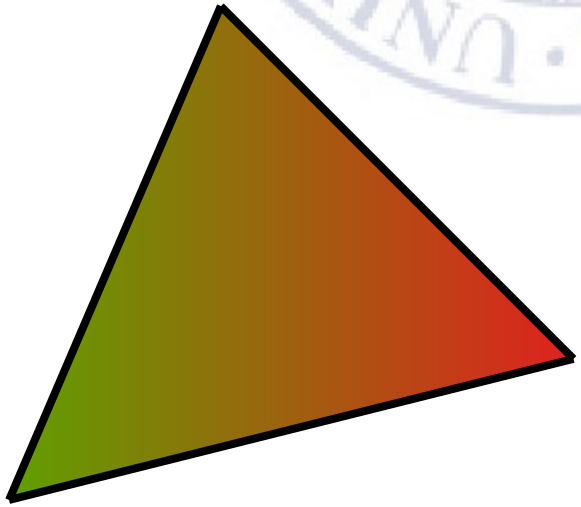
Gouraud shading

- Per ogni span si calcola il valore di I all'estremo con un algoritmo incrementale, e, sempre incrementalmente, si calcolano i valori all'interno della span



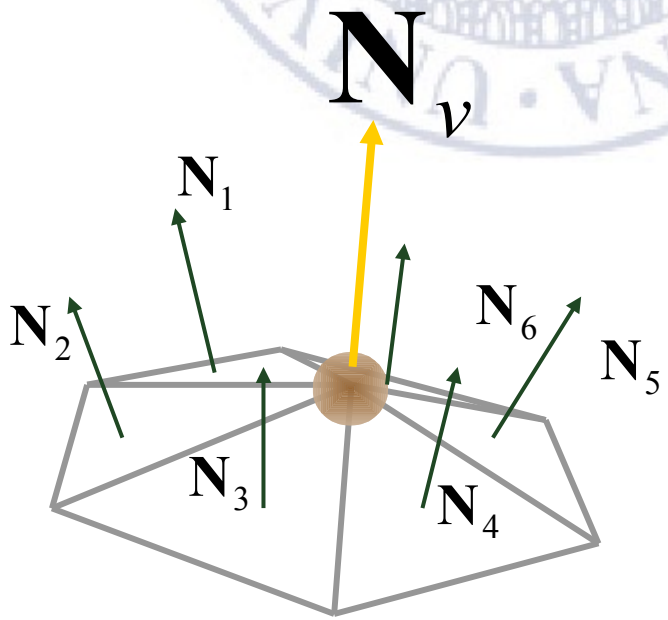
Gouraud shading

- Il risultato così ottenuto approssima molto il modello di Phong per superfici generiche rispetto allo shading costante



Gouraud shading

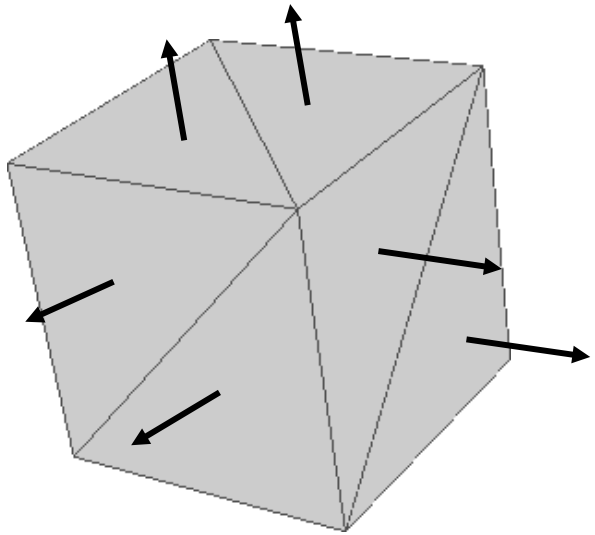
- Che normali utilizzo?
- La normale alla faccia è ben definita
- La normale al vertice la calcolo come media delle normali delle facce che insistono sul vertice

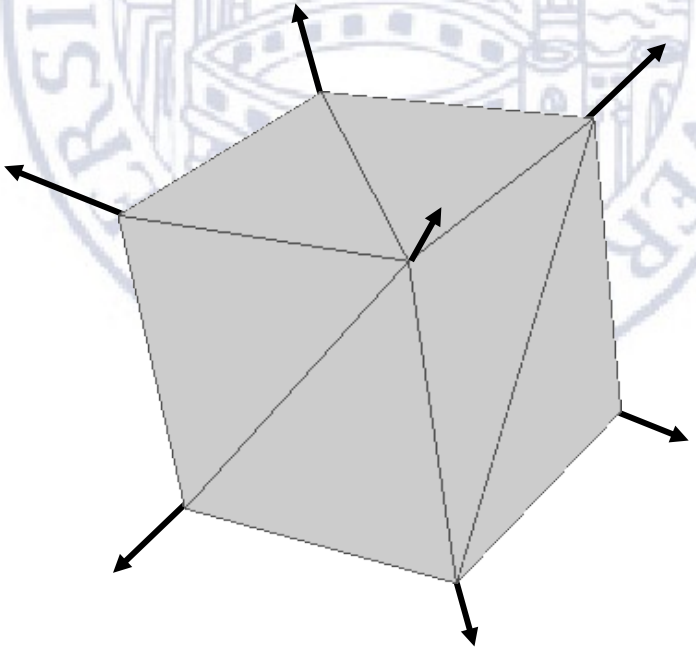


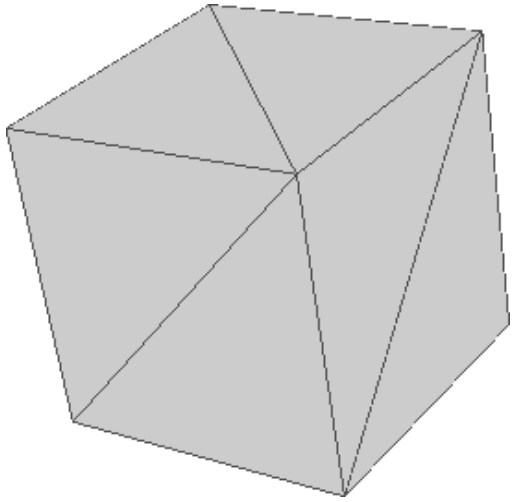
$$\vec{\mathbf{N}}_v = \frac{\sum_i \vec{\mathbf{N}}_i}{|\sum_i \vec{\mathbf{N}}_i|}$$

Gouraud shading

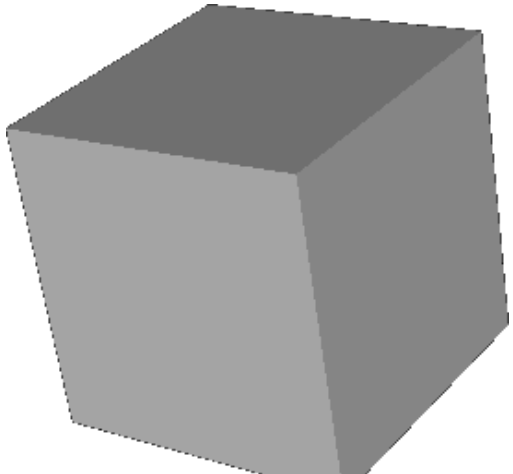
- Problema: gli spigoli "veri"?

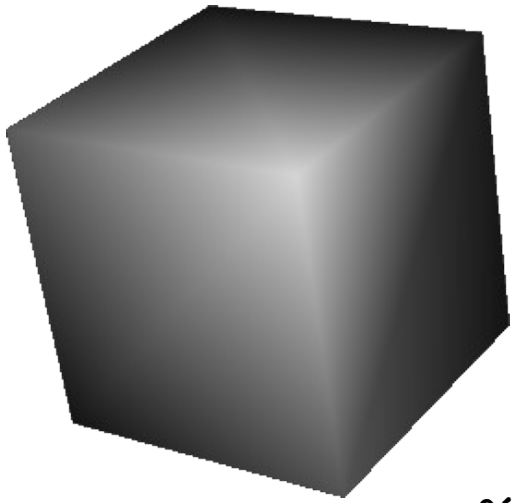






shading costante

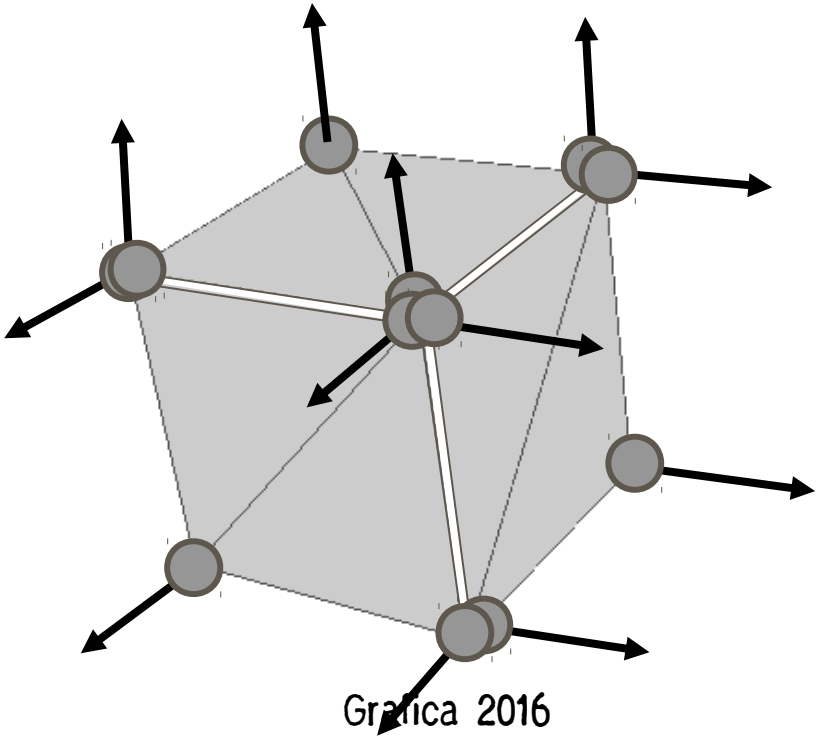




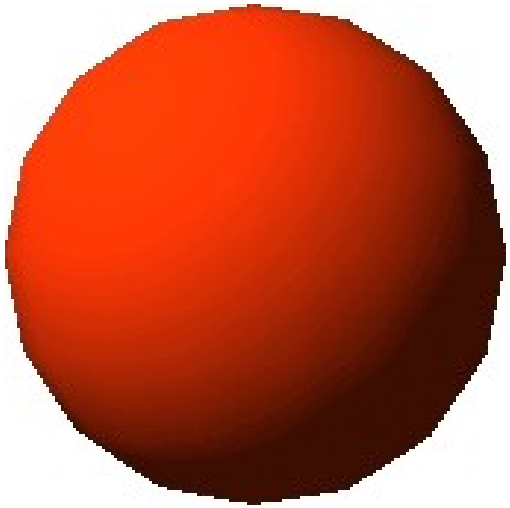
Gouraud shading

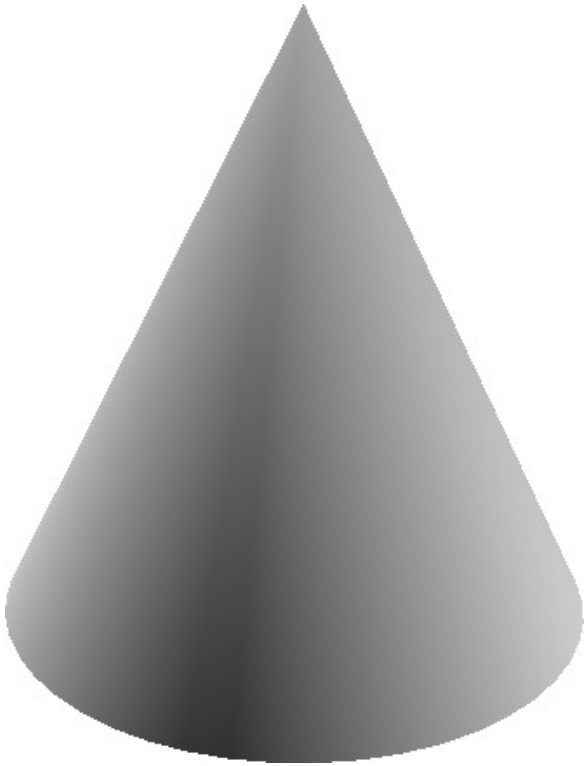
Gouraud shading

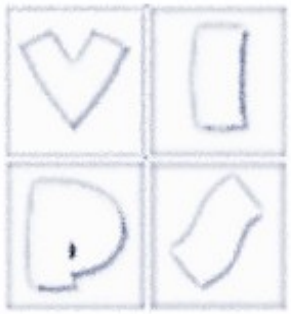
- Soluzione: si utilizzano normali diverse per i due lati dello spigolo
- La struttura dati deve memorizzare le adiacenze e le diverse tipologie



Paragone: costante e Gouraud





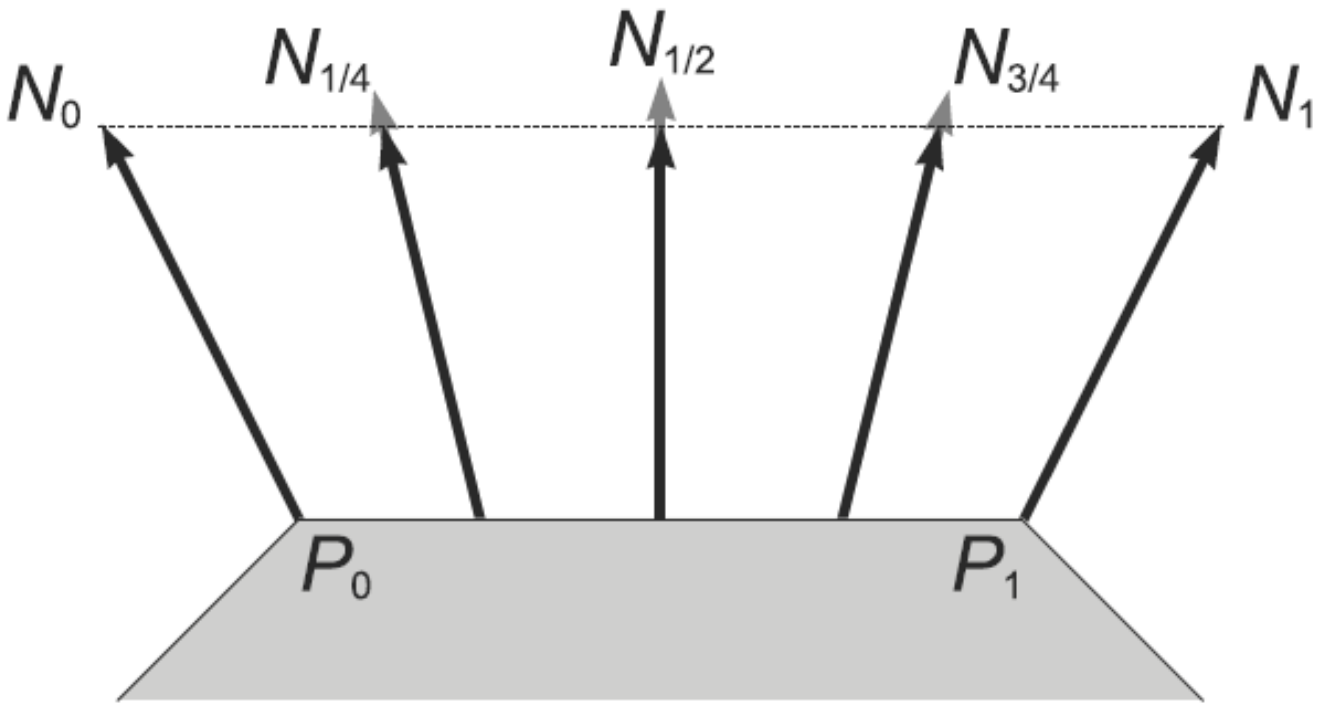


Phong shading

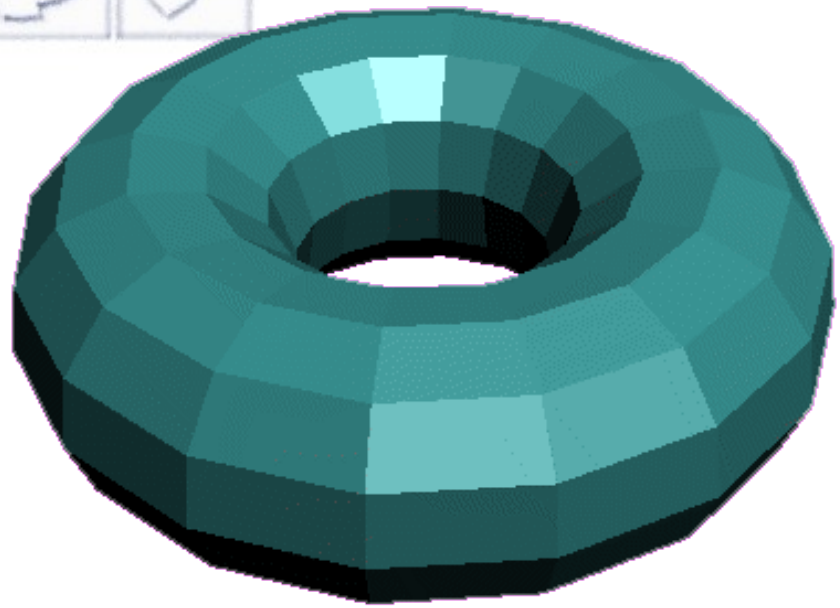
- Gouraud shading: ottimale rapporto qualità/prezzo
- Risultati non eccezionali per superfici dotate di un alto coefficiente n riflessione speculare
- Problema: per n alto lo *specular highlight* deve essere piccolo, invece si "propaga" per tutta la faccia (per interpolazione) se cade vicino a un vertice, si "perde" se è interno

Phong shading

- Soluzione: si interpola nello spazio delle normali e si calcola l'equazione di illuminazione in ogni pixel

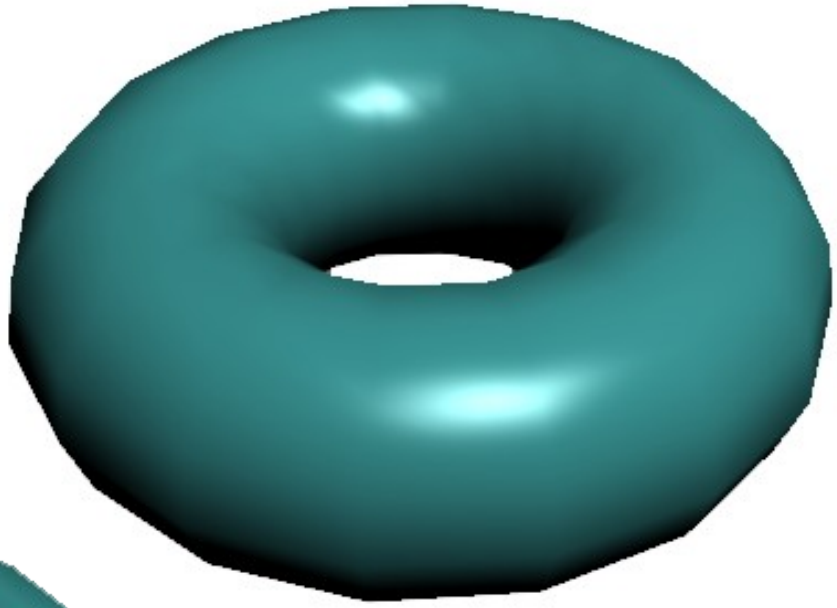


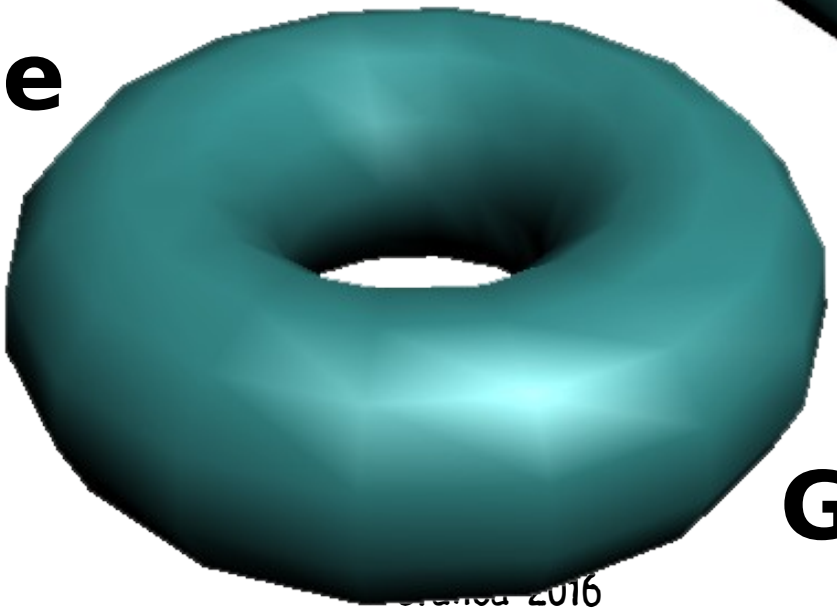
Paragone: costante, Gouraud e Phong



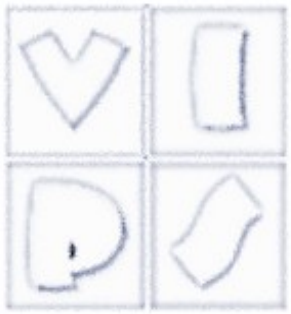
Costante

Phong





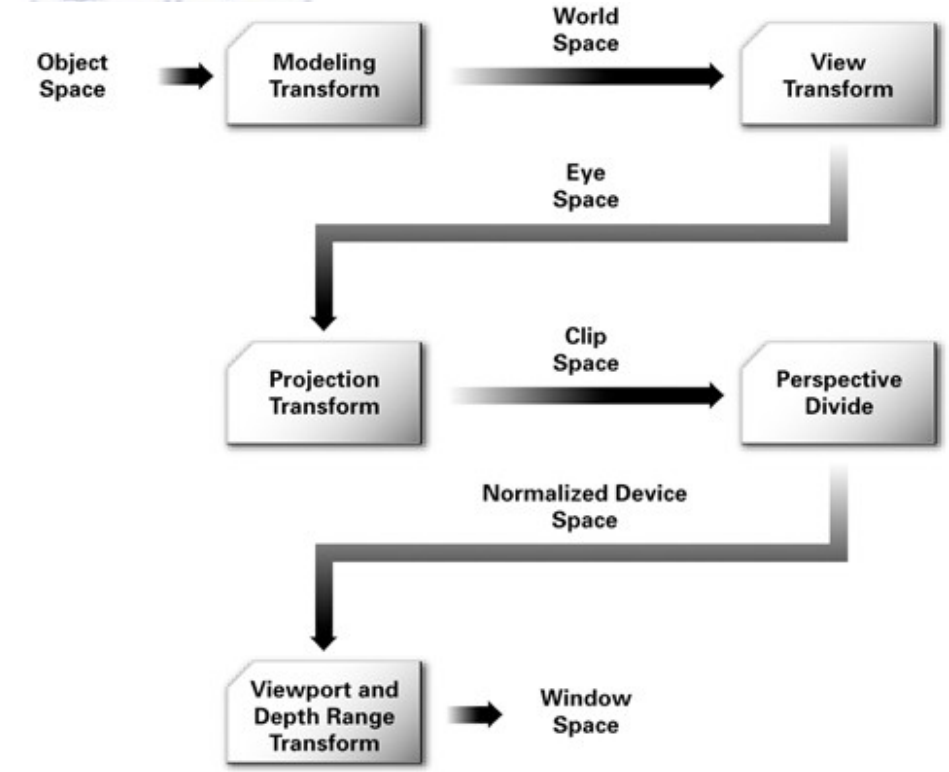
Gouraud

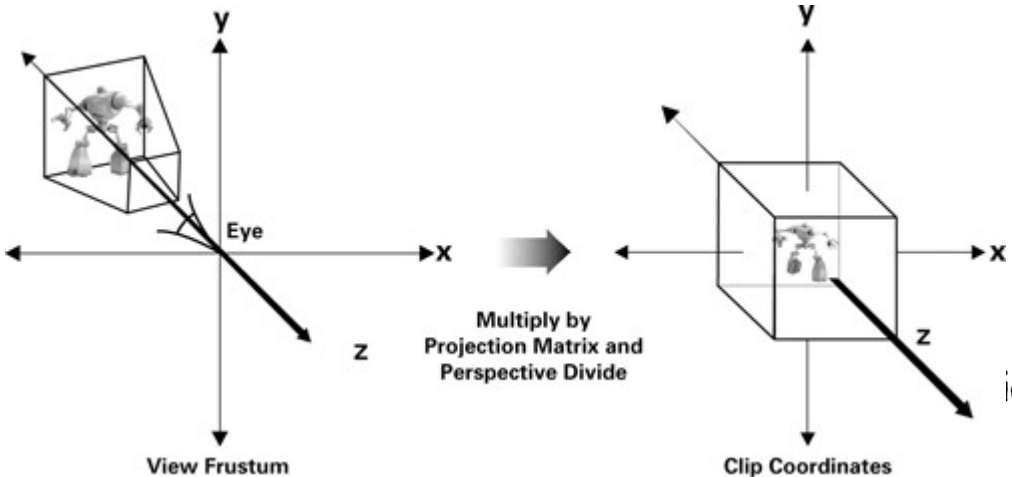


Shading

- Dato che ora la pipeline è completamente programmabile, si possono implementare il phong shading e shading più complessi dopo la rasterizzazione, sul fragment shader
- Occorre che tutti i parametri che servono per l'algoritmo siano propagati e interpolati tra vertex shader e fragment shader

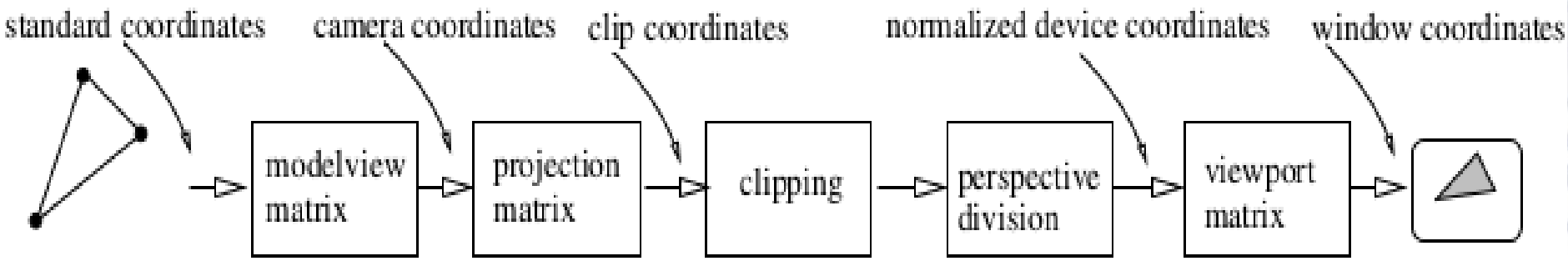
Ricapitoliamo: openGL





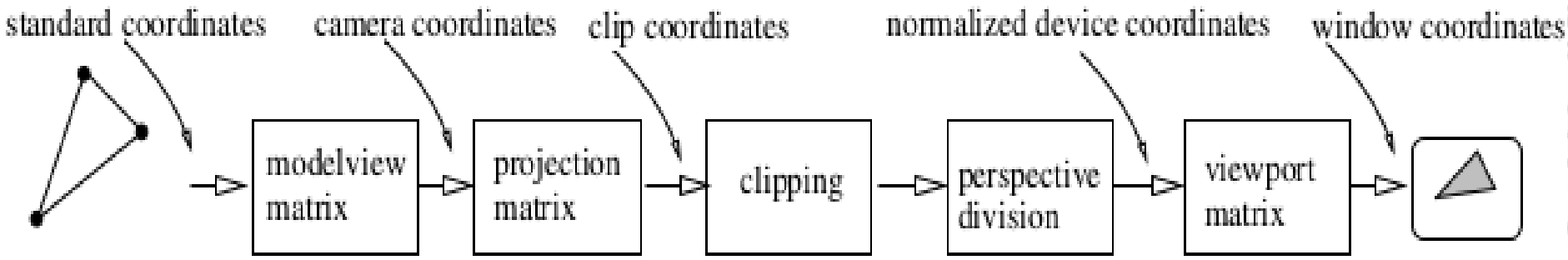
- In OpenGL non esiste lo spazio mondo. Attiene alla applicazione.
- Inoltre in openGL a differenza dello schema visto sopra tra le coordinate camera e le NDC ci sono le clip coordinates.
- Nello schema di prima le coordinate immagine sono 2D, mentre in OpenGL si distingue tra le window coordinates che conservano la pseudo-profondità e le screen coordinates, che sono 2D eliminando la pseudo-profondità

Storia di un poligono



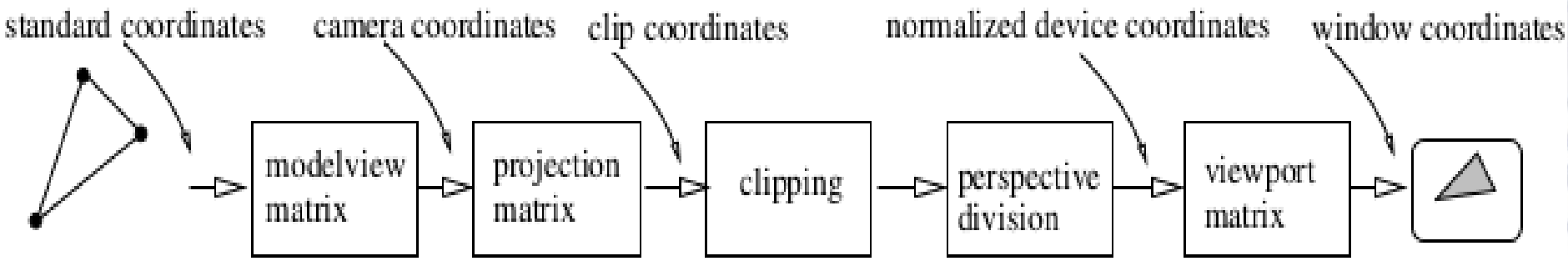
- Una maglia poligonale viene inserita nella pipeline grafica, assieme alle normali associate ai suoi vertici.
- I vertici vengono quindi trasformati dalla modelview matrix, che li trasforma dal sistema di riferimento mondo al sistema di riferimento telecamera (eye).
- Qui il modello di illuminazione locale viene applicato ai vertici, usando le normali, in modo da assegnare un colore a ciascun vertice.

Storia di un poligono

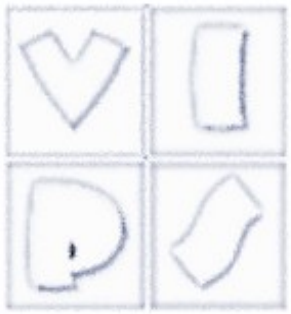


- Andando avanti nella pipeline viene applicata la projection matrix. Le coordinate risultanti prendono il nome di "clip coordinates".
- Si effettua il clipping dei poligoni.
- I vertici attraversano quindi la divisione prospettica, passando da una rappresentazione 3D ad una rappresentazione 2D + pseudo-profondità. Le normali si perdono, ma si mantiene il colore associato ai vertici.
- Siamo in normalized device coordinates che variano tra -1 ed 1.

Storia di un poligono



- Per mappare queste nella immagine finale (window coordinates) si applica la viewport matrix (la pesudodepth si mappa tra 0 ed 1).
- Si procede alla scan-conversion: i valori di pseudo-profondità e colore sui vertici vengono interpolati con interpolazione scan-line), per determinare la visibilità dei pixel interni (depth-buffer) e il loro colore (es. Gouraud shading)



Mapping e compositing

- Ci sono ancora diverse cose da fare per creare dei rendering di buona qualità utilizzando la pipeline grafica.
- Le tecniche che consentono di ottenere un buon risultato sono legate al combinare colori ottenuti da diverse procedure ed utilizzare anche mappe (texture) memorizzate nella memoria grafica.

Buffers

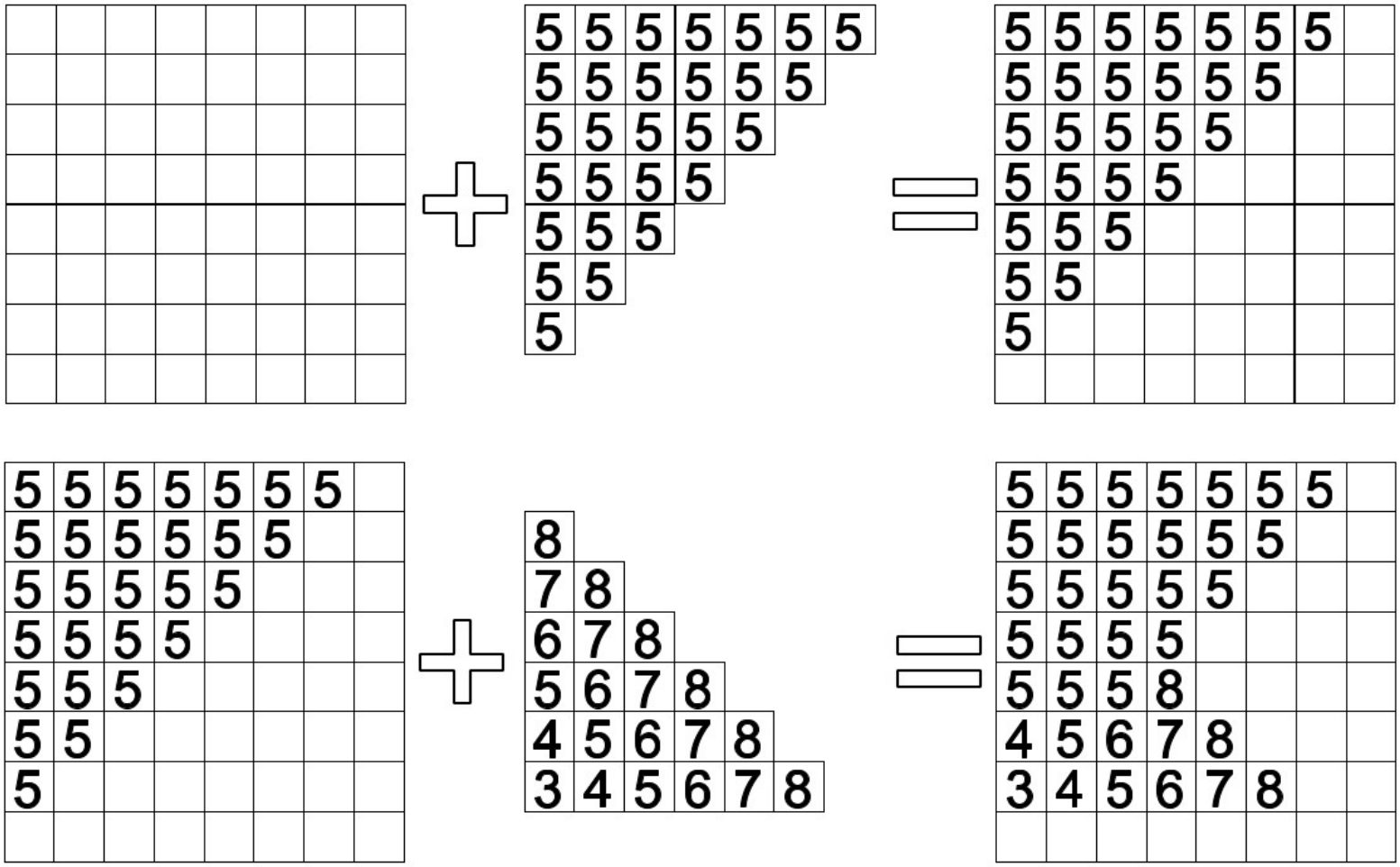
- Nella pipeline implementata in openGL, le immagini (digitali) vengono scritte in una delle due matrici front e back con le immagini in scrittura e quella da sostituire
- Abbiamo visto che si usa anche un altro buffer per memorizzare la profondità per l'algoritmo di rimozione superfici nascoste. Vediamo ora in dettaglio
- Ma esiste in OpenGl anche uno stencil buffer
  - Crea una maschera su cui è disabilitata la scrittura nelle corrispondenti posizioni
- La possibilità di utilizzare più flessibilmente la memoria poi consente di realizzare vari trucchi per ottenere un rendering migliore, combinando ad esempio diverse passate di rendering (tecniche multi-pass) o utilizzando il texture mapping.

Hidden surface removal in pixel space: z-buffer

- Durante la fase di rasterizzazione delle primitive si determina, per ogni pixel (x,y) su cui la primitiva viene mappata, la profondità della primitiva in quel punto. La rasterizzazione avviene dopo la proiezione sul piano di vista, nello spazio 3D NDC;
- Se la profondità z in (x,y) è inferiore alla profondità corrente memorizzata nello z-buffer allora (x,y) assume z come profondità corrente ed il pixel (x,y) nel frame buffer assume il valore colore della primitiva in esame.

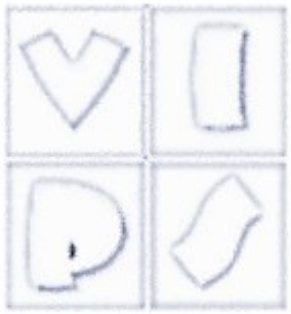
z-buffer

- Esempio



z-buffer

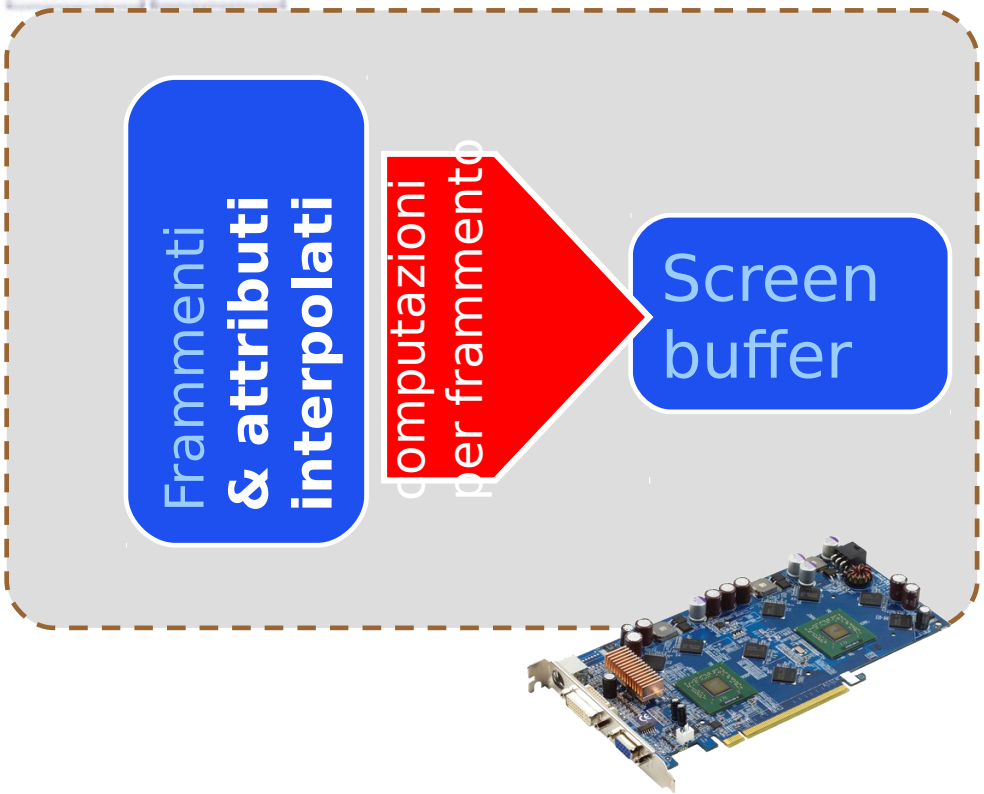
- Lo z-buffer ha la stessa risoluzione del frame buffer ed ogni cella ha dimensione sufficiente per memorizzare la profondità alla risoluzione richiesta (di solito 24 bit);
- Ogni elemento dello z-buffer è inizializzato al valore della distanza massima dal centro di proiezione;
- Non è richiesto alcun ordine preventivo tra le primitive geometriche;
- Generalmente implementato firmware;
- Complessità pressoché costante (ad un aumento delle primitive corrisponde in genere una diminuzione della superficie coperta).



z-buffer

- L'algoritmo z-buffer è un algoritmo di tipo image-space, basato su una logica molto semplice e facile da realizzare;
- Lavora in accoppiamento con l'algoritmo di scan conversion (rasterizzazione, disegno, delle primitive geometriche sulla memoria di quadro) ed ha bisogno, come struttura dati di supporto, di un'area di memoria, il depth buffer, delle stesse dimensioni del frame buffer
- Per ogni posizione (x,y) della vista che si genera, il frame buffer contiene il colore assegnato a quel pixel, il depth buffer la profondità del punto corrispondente sulla primitiva visibile.

Semitrasparenza in OpenGL



Alpha Blending

- I colori hanno 4 componenti:

  R,G,B, α

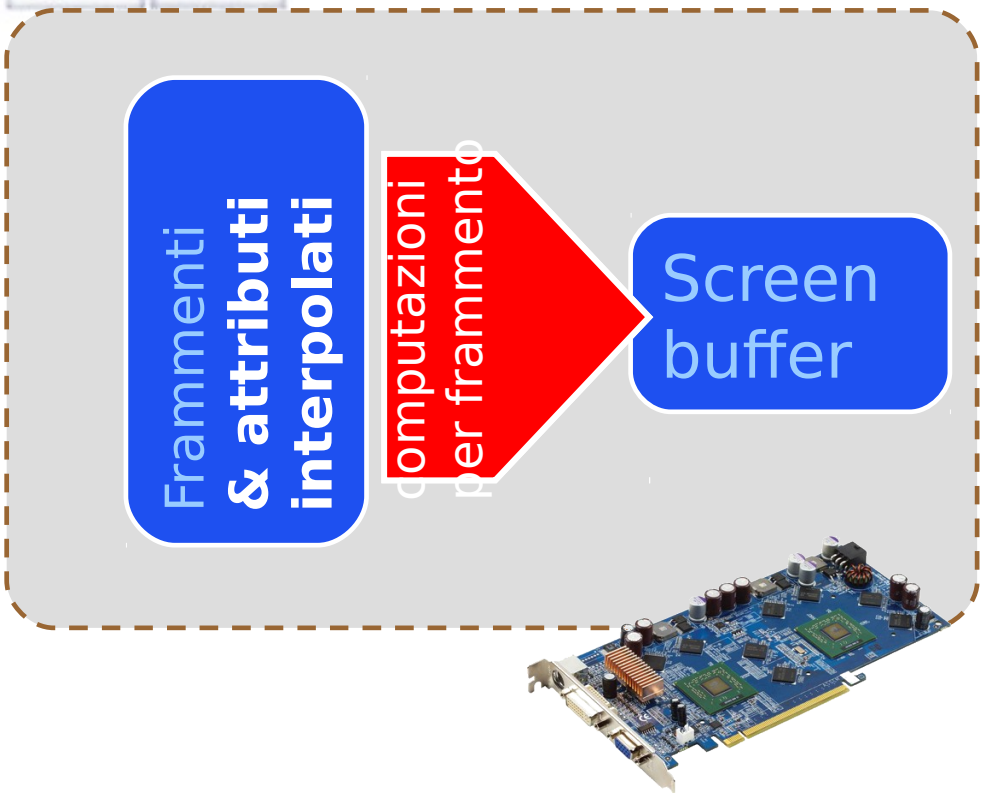
- Quando arriva un frammento (che sopravvive al depth test)

  invece di sovrascriverlo
  uso la formula

$$(r,g,b)_{finale} = (r,g,b)_{vecchio} \cdot (1-\alpha) + (r,g,b)_{nuovo} \cdot (\alpha)$$

"alpha blending"

Semitrasparenza in OpenGL



- Comodo avere la trasparenza come un canale del colore!
- Problema: Alpha blending e z-buffer
  - Per oggetti semi-trasparenti, l'ordine di rendering torna ad essere determinante
  - Necessario un passo di ordinamento in profondità degli elementi della scena (sorting da effettuare per ogni diverso viewpoint) ➔ overhead che riduce frame rate

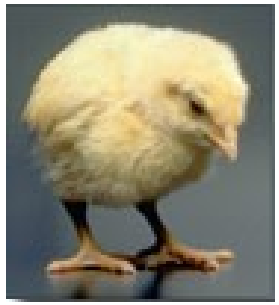
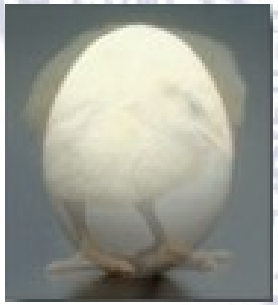
Semitrasparenza in OpenGL

vecchia:
(già sul buffer)

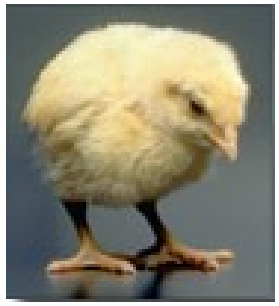
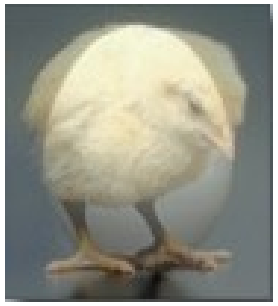
nuova:
che sovrascrivo

risultato

α = livello di trasparenza

$(1-\alpha)$ x  + α x  = 

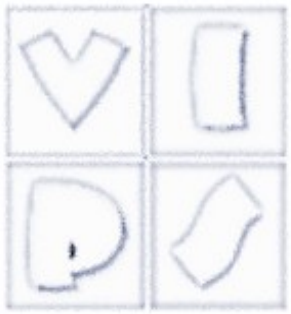
$\alpha = 0.25$

$(1-\alpha)$ x  + α x  = 

$\alpha = 0.5$

$(1-\alpha)$ x  + α x  = 

$\alpha = 0.75$



Altre applicazioni

- Nel caso delle **light map** viene eliminato il colore della immagine source e viene invece modulato il colore dell'immagine nel frame buffer con il colore della light map.
- Per quanto riguarda l'**anti-aliasing**, ci si riferisce qui alla rimozione dell'effetto scalettatura che si ottiene nella rasterizzazione di primitive geometriche (in particolare, linee). Il trucco consiste nell'assegnare valori di opacità α ad un pixel pari alla frazione del pixel che è coperta dalla linea. Si procede poi ad effettuare il rendering con opacità.

Depth cueing

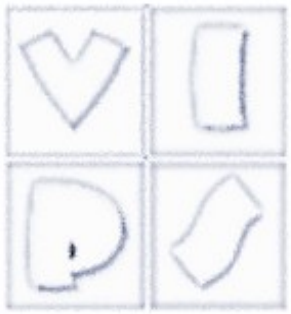
- Si stabilisce un colore per la nebbia pari a C_f
- Se lo shading fornisce il colore C_s per il pixel, gli si assegna il colore: $C_s = (1 - z_s)C_s + z_s C_f$
  - ovvero si interpola linearmente tra il colore dello shading ed il colore della nebbia . In tal modo oggetti vicini appaiono colorati normalmente (o quasi), oggetti lontani invece sfumano in C_f
- Se C_f è nero, per esempio, gli oggetti tendono a sparire man mano che si avvicinano al piano di far clipping $(z_s = 1)$
- Se C_f è bianco, invece, si ha un effetto "nebbia"
- La tecnica del depth cueing si usa per:
  - dare un senso di profondità all'immagine
  - modellare la vera nebbia o altri effetti atmosferici
  - mascherare il fenomeno di pop-in. Gli oggetti lontani entrano ed escono dal view frustum con un effetto di dissolvenza più gradevole.

Depth cueing

- Si può anche usare una funzione di z_s per avere effetti diversi, oppure una densità di nebbia indipendente da z_s .
- Si possono impiegare fog maps per definire una nebbia non uniforme. Una fog map è una immagine, in cui RGB rappresentano il colore della nebbia e α la sua densità.

Altre tecniche multipass

- **Motion blur**. Nelle macchine fotografiche reali, a causa del tempo di esposizione finito, gli oggetti in moto appaiono sfocati (in ragione della loro velocità apparente). Per simulare questo effetto si può fare il rendering dell'oggetto in posizioni diverse (lungo la traiettoria) e accumulare i risultati.
- **Depth of field**. Un altro effetto tipico delle telecamere reali è la profondità di campo, ovvero il fatto che gli oggetti appaiono perfettamente a fuoco solo ad una certa distanza z_f Quelli che si trovano più vicini o più lontani appaiono sfocati Per simulare questo effetto si accumulano diverse immagini prese muovendo la telecamera in modo che i punti sul piano $z = z_f$ siano fermi. In pratica si deve muovere il punto di vista ed aggiustare il view frustum.



Riferimenti

- Scateni cap. 5
- Ganovelli cap.5
- Angel cap. 7.1, 7.11