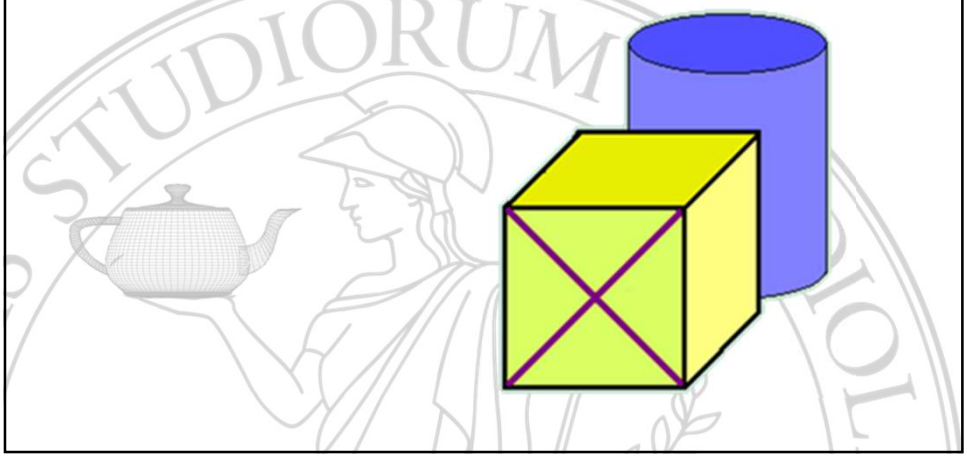
Marco Tarini - Computer Graphics 2022/2023
Università degli Studi di Milano

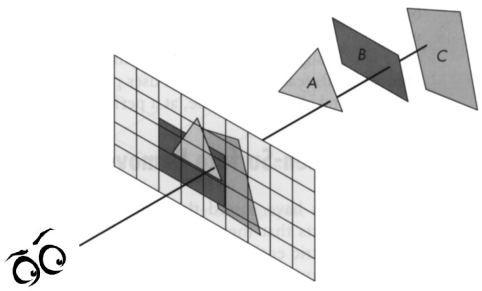
Rimozione Superfici Nascoste



1

Rimozione delle superfici nascoste (cioè occluse)

- ✓ Gli oggetti più vicini devono coprire ("occludere") quelli più lontani



Con raytracing era ovvio!
Con raserization?

2

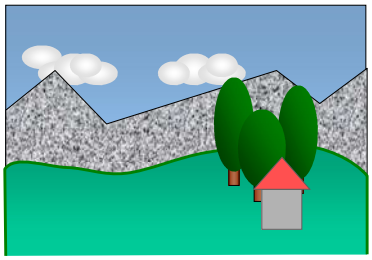
Rimozione delle superfici nascoste ("hidden surface removal")

- ✓ Negli approcci basati su ray-tracing, questo era, concettualemnte, banale da ottenere:
  - ⇒ basta prendere, per ogni raggio primario, la primitiva intersecata avente la distanza *k* minore
- ✓ Negli approcci basati sulla rasterizzazione, non è banale
  - ⇒ Se i pixel prodotti dalla rasterizzazione di una primitiva **sovrascrivono** sullo screen buffer quelli generati alla stessa posizione dalle primitive precedent…
  - ⇒ …allora il valore finale sullo screen buffer sarà semplicemente il valore dei pixel rasterizzati dall'ultima primitiva rasterizzata

3

Rimozioni delle superfici nascoste

- ✓ Soluzione 1: basata su ordinamento
  - ⇒ le primitive rasterizzate **sovrascrivono** nel frame buffer quelle rasterizzate in precedenza
  - ⇒ quindi, basta rasterizzare le primitive nell'*ordine giusto*
    - ordine "back-to-front"
  - ⇒ approccio noto come "**algoritmo del pittore**"



4

Algoritmo del pittore (o depth sorting)

✓Data una scena (composta da primitive)

⇒ordinare le primitive back-to-front

- dalle più lontane dalla camera, alle più vicine

⇒quindi, in ordine di coordianta Z…

- crescente, in spazio vista
  (dato che la Z del nostro spazio vista decresce col crescere della distanza dalla camera)
- decrescente, in spazio clip o schermo
  (dato che la Z – o depth – in spazio schermo decresce)

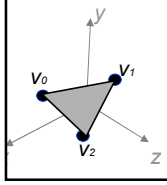
⇒rasterizzarle in successione

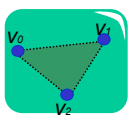
- Permettendo ai frammenti generate dalle primitive di sovrascrivere i pixel già presenti sul buffer (generati dalla primitive precedenti)

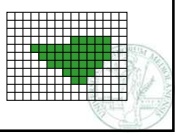
5

Algoritmo del pittore (depth sorting): non è adatto a soluzioni hardware

La coordianta z in spazio vista non è nota a priori.
Nessuna fase del pipeline HW si presta a questo task.



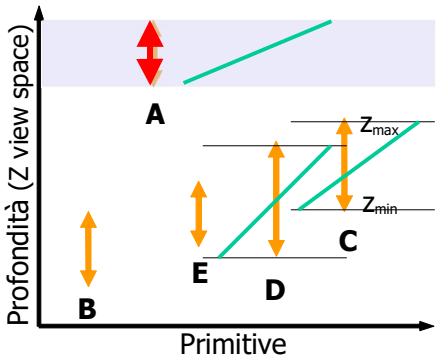




6

Algoritmo del pittore (depth sorting)

- ✓ In questo esempio, la primitiva A può essere disegnata tranquillamente per prima
- ✓ Ma va prima C o D?

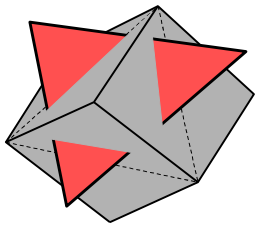


7

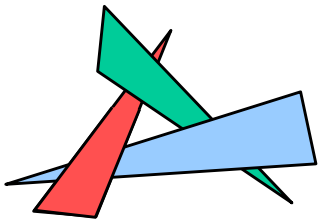
Algoritmo del pittore (depth sorting)

- ✓ Esiste sempre un ordine corretto?

⇒NO.
Controesempio: intersezioni



⇒NO.
Controesempio: cicli



8

Rimozioni delle superfici nascoste tramite ordinamento: sommario degli ostacoli

- ✓ Limiti delle soluzioni basate su oridnamento:
  - ⇒ Problema 1: ordinamento costa O (n log n) ("pseudolineare")
    - con n numero di primitive
    - n può molto grande. Es: se n = milioni → log(n) è un fattore 30
      In CG, qualsiasi cosa peggiore di lineare è mal tollerato
  - ⇒ Problema 2: quando effettuare l'ordinamento?
    - le coordinate in spazio vista (compreso la Z)
      sono note solo dopo la trasfromazione
  - ⇒ Problema 3: una primitiva non ha un'unica Z
    - ogni vertice che la compone ha una Z diversa
    - quale usare? (media, min, max…). Scelte diverse, ordini diversi.
    - a volte, non esiste alcun ordinamento che da i risultati corretti
  - ⇒ Problema 4: complica il rendering
    - prescrivendo un certo ordine di rendering delle primitive

9

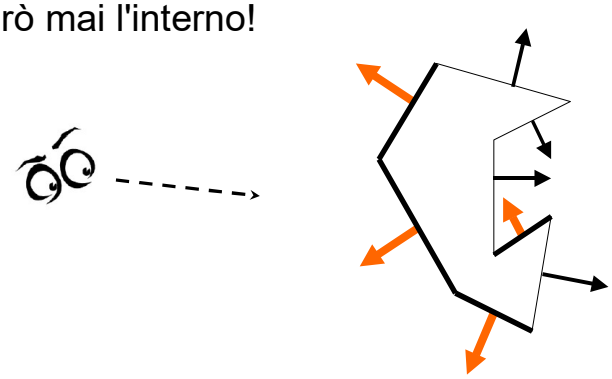
Algoritmo del pittore (depth sorting)

- ✓ Gli ostacoli sono superabili, ma l'algoritmo del pittore in pratica è usato raramente
  - ⇒ solo nelle occasioni in cui sia facile ordinare preventivamente le primitive back-to-front in fase di preprocessing
  - ⇒ esempio: terreno come campo di altezza
- ✓ Vediamo invece alternative per rimuovere le superfici nascoste che sono «order independent»
  - ⇒ Il redenring produce lo stesso risultato, indipendentemente dall'ordine di disegno delle primitive

10

Caso particolare per mesh chiuse e ben orientate. Backface Culling

- ✓ Un caso particolare di Occlusion Culling
- ✓ Idea:
  - ⇒ ipotesi: superficie **chiusa** e **opaca**…
  - → non vedrò mai l'interno!



11

Caso particolare: mesh chiuse e ben orientate

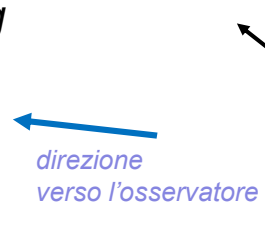
- ✓ Back-face Culling
  - ⇒ "to cull" = scartare.
    To cull a face = scartare la primitiva in fase di rendering
- ✓ Idea:
  - ⇒ ipotesi: mesh **ben orientata** e **chiusa**,
  - ⇒ ipotesi: posizione POV esterno alla superficie
  - ⇒ ipotesi: materiale opaco (nel senso di non trasparente)
  - → non vedrò mai l'interno!
  - → qualunque faccia si veda "da dietro"
    (back-facing triangle)
    finirà per forza *occlusa* alla vista da (almeno)
    una faccia "vista da davanti"
    (front-facing triangle)
  - → posso scartare tutti i triangles che sono back-facing, sapendo che saranno automaticamente occlusi

12

Back-face culling

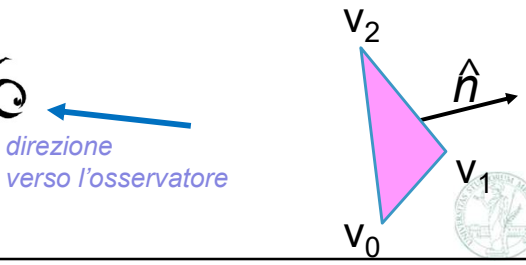
✓Traingolo *front-facing*

⇒"visto da davanti"



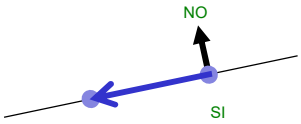
✓Traingolo *back-facing*

⇒"visto di spalle"

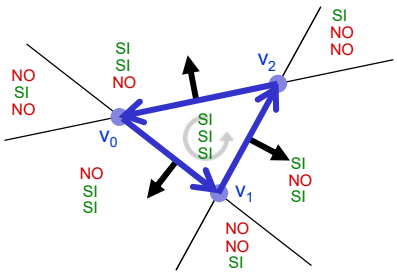


15

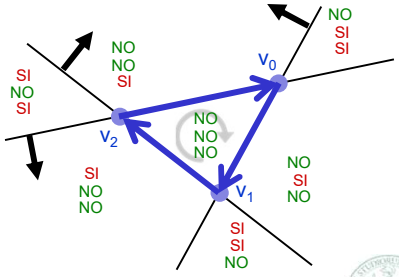
Back-face culling e test di appartenenza di frammento a triangolo



SCREEN SPACE:



Front- facing

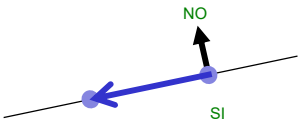


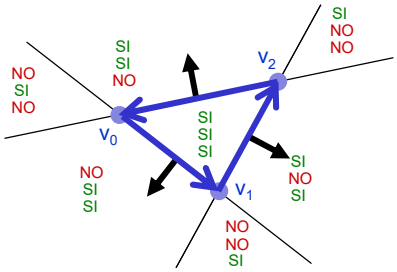
Back-facing

16

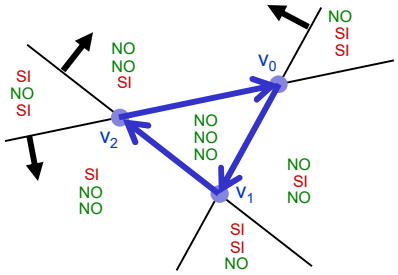
Back-face culling e test di appartenenza di frammento a triangolo

- ✓ Triangolo = interesezione di 3 semipiani
- ✓ Un punto è interno al triangolo *sse* appartiene a tutti e tre i semipiani





Triangolo 2D front facing
(in spazio Clip)



Triangolo 2D back facing
(in spazio Clip)

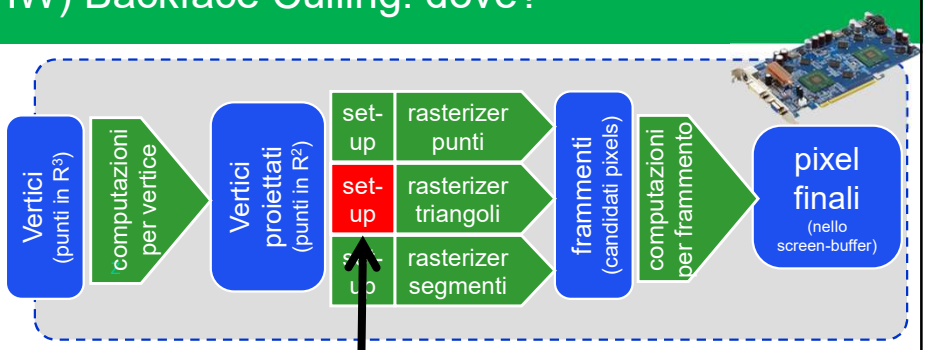
17

Back-face culling e test di appartenenza di frammento a triangolo

- ✓ Il *facing* (front o back) di una faccia è il segno del prodotto dot fra normale della faccia è direzione di vista V
  - ⇒ V = direzione VERSO l'osservatore,
  - ⇒ cioè V = (0,0,1) in spazio Vista
  - ⇒ Se prodotto dot positivo = back facing (assumendo normali verso esterno)
- ✓ Equivalentemente, il facing può essere determinato dall'orientamento orario oppure antiorario dei sui vertici in spazio clip (oppure schermo)
  - ⇒ Dipende da come è modellata la mesh ma per convenzione:
  - ⇒ Antiorario : front-facing
  - ⇒ Orario : back facing
- ✓ Questo può essere determinato nello stesso test di appartenenza del frammento al triangolo (vedi rasterizzazione):
  - ⇒ Se il frammento supera tutti e tre gli edge test: frammento interno a triangolo anti-orario
  - ⇒ Se il frammento non supera nessuno dei tre edge test: frammento interno a triangolo orario (scartare, se il back-face culling è attivato)
  - ⇒ Ogni altro caso: frammento esterno (scartare in ogni caso)

18

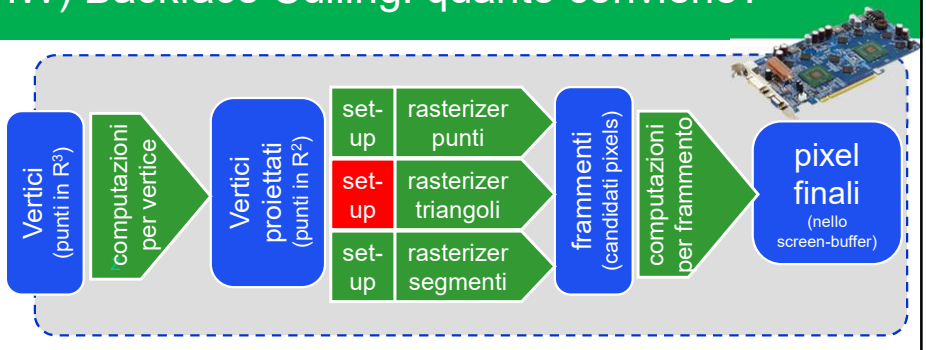
(HW) Backface Culling: dove?



QUI.
(in una fase di set-up che precede la rasterizzazione di ciascun triangolo)

19

(HW) Backface Culling: quanto conviene?



scarto circa il 50% delle facce...
Lo speed-up? **x2** ?

20

Quando usare il backface culling

- ✓ *Se valgono le ipotesi dette,* allora il back-face culling non cambia le immagini generate
  - ⇒ Altrimenti, genera artefatti (scompare il «retro» delle superfici)
- ✓ Quando è possibile usarlo, è molto conveniente perché…
  - ⇒ migliora le prestazioni delle applicazioni fill-limited, riducendo di circa il 50% il numero di frammenti prodotti
    - Non impatta però le applicazioni geometry-limited, dato che lo scarto della primitiva avviene a valle del processing per vertice
  - ⇒ contribuisce all'hidden-surface removal, dato rimuove solo facce auto-occluse
    - Non è però sufficiente, nel caso di mesh concave, perché non rimuove *tutte* le facce auto-occluse (vedi dopo)
- ✓ E' un test eseguito dall'HW della GPU.
  - ⇒ Quindi, programmando l'API, il programmatore si limita ad abilitarlo oppure disabilitarlo, a seconda delle condizioni
  - ⇒ (non: viene implementato dal programmatore della app)

22

Esempio di abilitazione del backface culling in API grafiche: in WebGL (JavaScript)

- ✓ Attivare e disattivare, prima di renderizzare una mesh:

```
gl.enable( gl.GL_CULL_FACE );
gl.disable( gl.GL_CULL_FACE );
```

- ✓ E decidere se scartare le *front* oppure le *back-facing* :

```
gl.cullFace( gl.GL_FRONT );
gl.cullFace( gl.GL_BACK );
```

Questi comandi cambiano solo lo stato dell pipeline di rendering, che influenza le prossime primitive che verranno rasterizzate (fino a contrordine)

23

Esempio di abilitazione del backface culling in librerie ad alto livello: in three.js (JavaScript)

✓ Abilito o disabilito il back-face culling settando un parametro del materiale associato alle mesh

⇒Back-Face Culling abilitato, scarta le face back-facing:

```
materiale.side = THREE.Front;
```
il default

⇒"Front-Face Culling" abilitato, scarta le face front-facing:

```
materiale.side = THREE.Back;
```

⇒Back-Face Culling disabilitato:

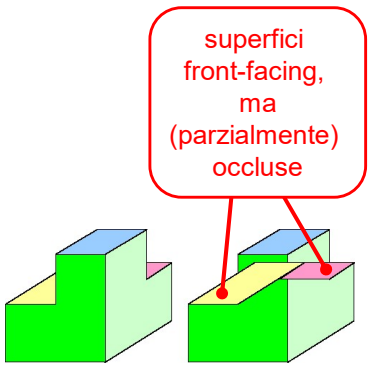
```
materiale.side = THREE.DoubleSide;
```

24

Rimozioni delle superfici nascoste

✓ Back-face culling non basta a rimuovere *tutte* le superfici nascoste

⇒Contro-esempio:
(succede solo con le superfici concave, cioè non convesse)



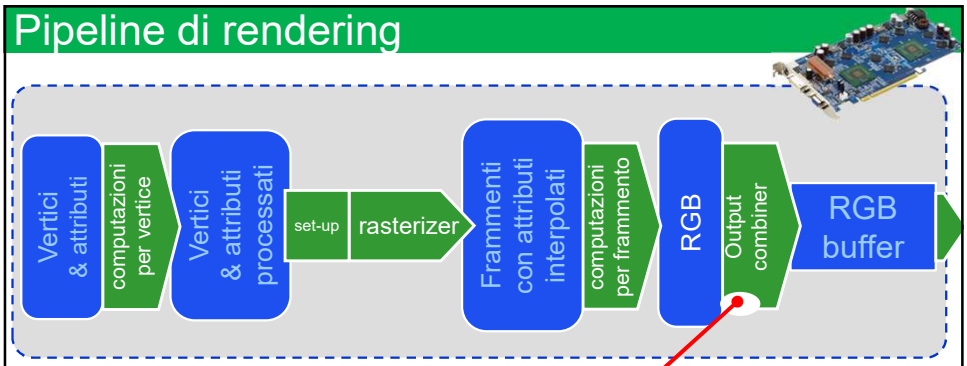
25

Rimozioni delle superfici nascoste: attraverso il Depth-buffer

- ✓ Idea base: eseguire il test a livello di frammento
  - ⇒ Di tutti i frammenti che insistono su un pixel, tengo solo quello di profondità (depth) minore
    - cioè quello più vicino all'osservatore
  - ⇒ Serve una struttura per memorizzare la profondità attuale di ogni pixel…
- ✓ «Depth-buffer»: (a volte: «Z-buffer»)
  - ⇒ un buffer 2D
  - ⇒ stessa risoluzione dello screen buffer
  - ⇒ per ogni posizione [x , y] memorizza il depth del frammento attualmente presente in quella locazione

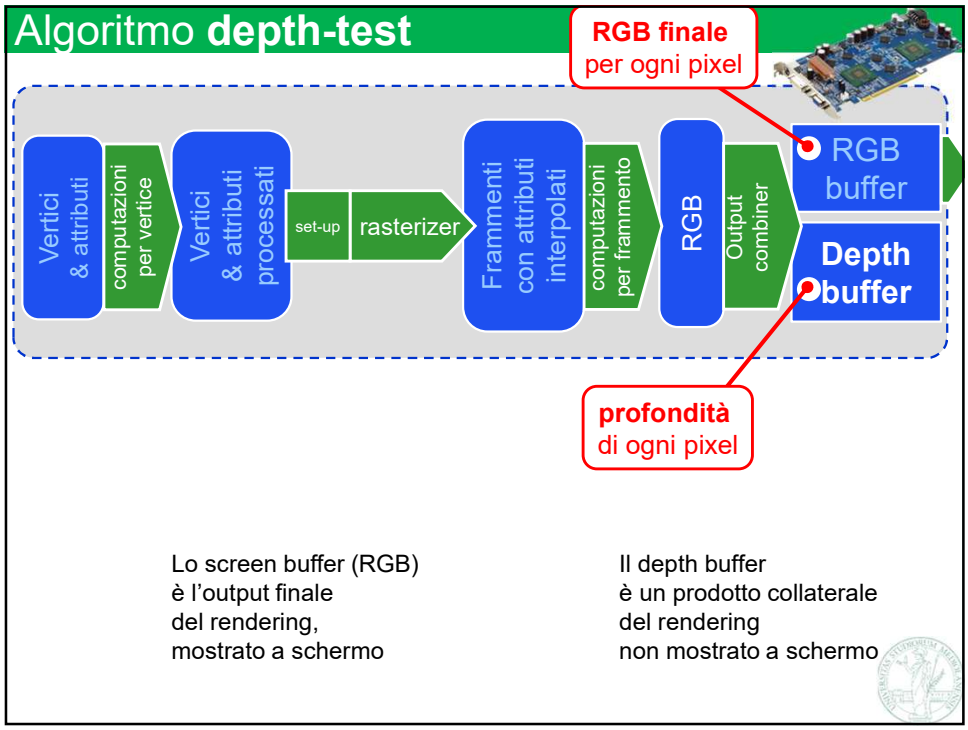
26

Pipeline di rendering

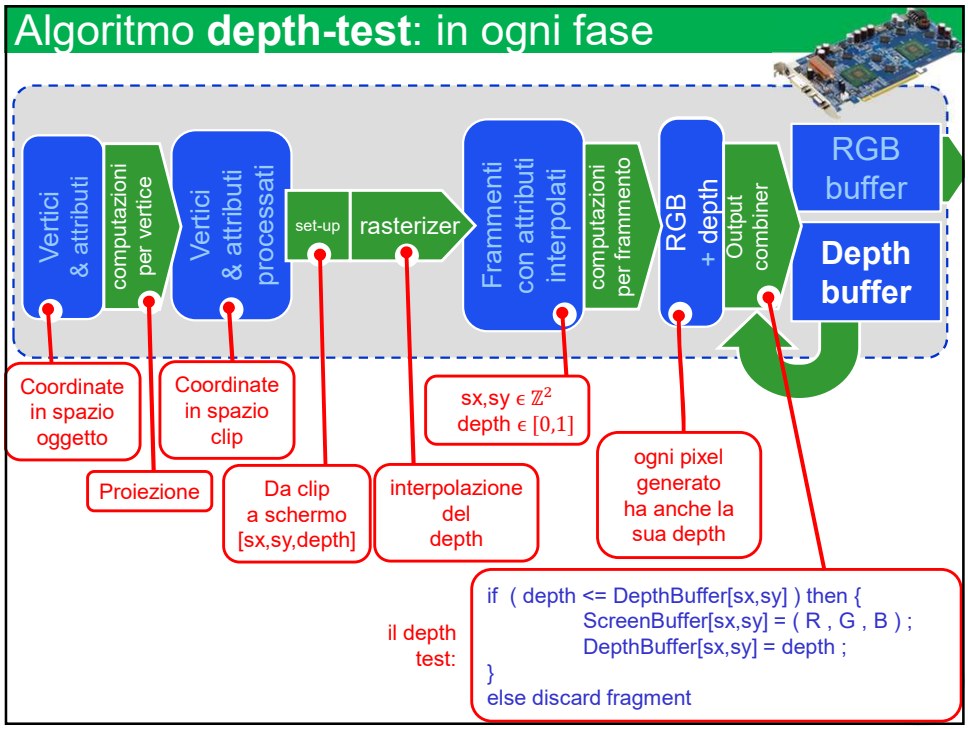


Una fase della pipeline di rendering, dal funzionamento fisso (e non programmabile), che si occupa di memorizzare i pixel generati da un frammento nei buffer di uscita, applicando varie regole

27

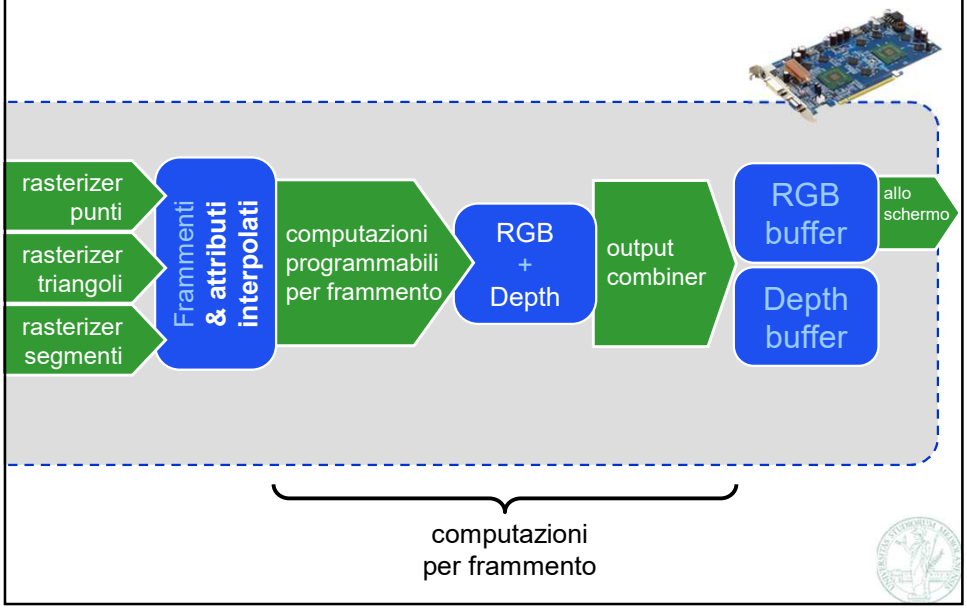

Algoritmo depth-test
RGB finale
per ogni pixel
Vertici & attributi
computazioni per vertice
Vertici & attributi processati
set-up
rasterizer
Frammenti con attributi interpolati
computazioni per frammento
RGB
Output combiner
RGB buffer
Depth buffer
profondità
di ogni pixel
Lo screen buffer (RGB)
è l'output finale
del rendering,
mostrato a schermo
Il depth buffer
è un prodotto collaterale
del rendering
non mostrato a schermo

28


Algoritmo depth-test: in ogni fase
Vertici & attributi
computazioni per vertice
Vertici & attributi processati
set-up
rasterizer
Frammenti con attributi interpolati
computazioni per frammento
RGB + depth
Output combiner
RGB buffer
Depth buffer
Coordinate in spazio oggetto
Proiezione
Coordinate in spazio clip
Da clip a schermo [sx,sy,depth]
interpolazione del depth
sx,sy ϵ ℤ²
depth ϵ [0,1]
ogni pixel generato ha anche la sua depth
il depth test:
if (depth <= DepthBuffer[sx,sy]) then {
ScreenBuffer[sx,sy] = (R , G , B) ;
DepthBuffer[sx,sy] = depth ;
}
else discard fragment

29

Il pipeline: da più vicino



30

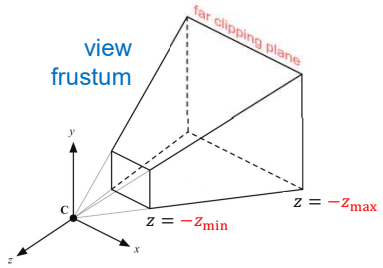
Valore di depth (profondità) di un frammento

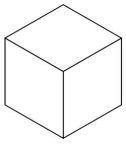
- ✓ Un valore da 0 a 1
  - ⇒ 0: pixel più vicino possible
  - ⇒ 1: pixel più lontano possible
  - ⇒ i frammenti con depth non inclusa fra 0 e 1 sono scartati automaticamente dal rasterizzatore
- ✓ Determinato (per vertice) a partire dalla Z dello spazio clip
  - ⇒ Se Z = -1 → depth = 0
  - ⇒ Se Z = +1 → depth = 1
  - ⇒ depth = (Z+1) / 2
- ✓ Interpolato, dentro ogni primitiva, per ottenere il valore dei frammenti
  - ⇒ Come qualsiasi altro valore attributo definito sui vertici
  - ⇒ Attraverso coordinate baricentriche

31

Limiti dello spazio visibile (cioè quello inquadrato)

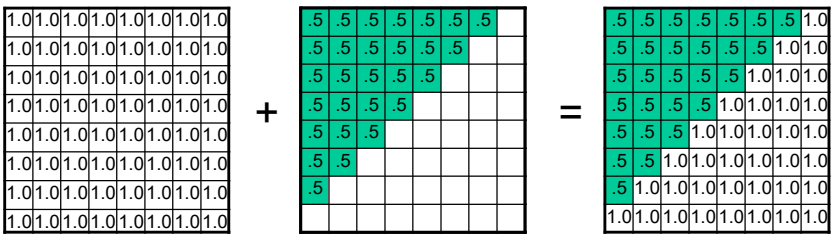
		Spazio Vista	Spazio Clip	Spazio Schermo	
x	da:	«left clipping plane»	-1	0	pixel coordinates
	a:	«right clipping plane»	$+1$	res_x	
y	da:	«bottom clipping plane»	-1	0	
	a:	«top clipping plane»	$+1$	res_y	
z	da:	$-z_{min}$	-1	0	pixel depth
	a:	$-z_{max}$	$+1$	1	

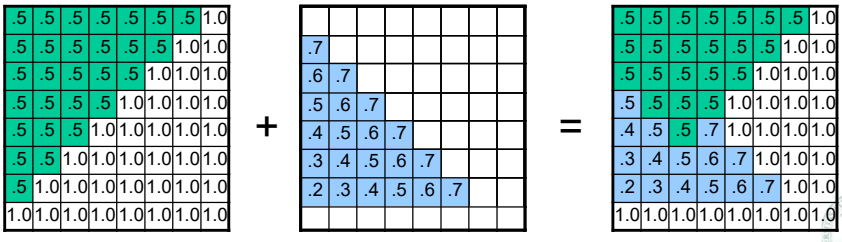




32

Algoritmo dello **depth-test**: esempio

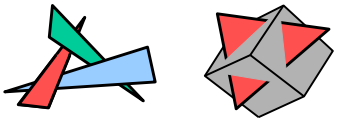




33

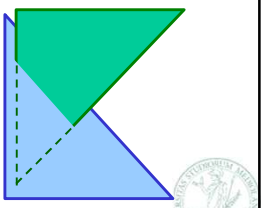
Algoritmo del **depth-test**: vantaggi

- ✓ il rendering diventa "order independent" ☺!!!
- ✓ Funziona su tutto ☺
  - ⇒ anche su:



.5	.5	.5	.5	.5	.5	.5	1.0
.5	.5	.5	.5	.5	.5	1.0	1.0
.5	.5	.5	.5	.5	1.0	1.0	1.0
.5	.5	.5	.5	1.0	1.0	1.0	1.0
.4	.5	.5	.7	1.0	1.0	1.0	1.0
.3	.4	.5	.6	.7	1.0	1.0	1.0
.2	.3	.4	.5	.6	.7	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

- ✓ Adatto all'implementazione parallela quindi HW ☺



34

Algoritmo del **depth-test**: costi e limiti

- ✓ Costa un po' di memoria (GPU)
  - ⇒ spesso: sizeof(depth buffer) = sizeof(screen buffer) / 2
  - ⇒ il buffer deve essere inizializzato (a profondità massima) prima di ogni rendering
- ✓ Problemi di *aliasing sulla z*
  - ⇒ detto: "z-fighting"
  - ⇒ quando la precisione dei valori di depth non è sufficiente
    es., se si renderizzano due superfici parallele molto vicine
- ✓ I frammenti vengono scartati dal test solo alla fine del pipeline
  - ⇒ tutta la computazione è già stata inutilmente effettuata
- ✓ Assume superfici del tutto opache
  - ⇒ problemi con le superfici semitrasparenti
- ✓ Il depth buffer è memoria condivisa in lettura e scrittura ☹
  - ⇒ complicazione per chi implementa HW
  - ⇒ efficienza ameno in parte impattata
    (anche se i produttori implementano molte ottimizzazioni)
  - ⇒ per questo, il depth test è gestito da una apposita fase non programmabile
    (non è parte di un fragment-shader)

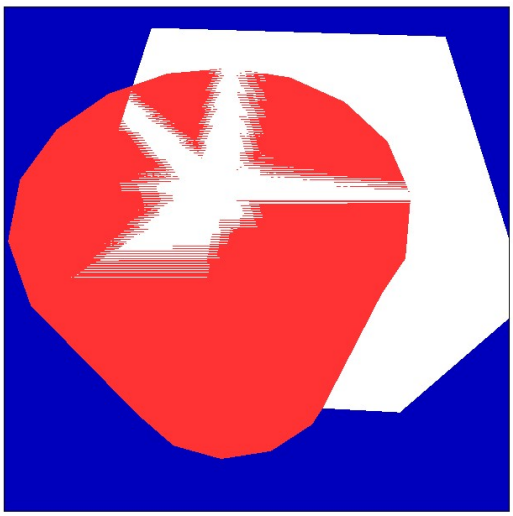
37

Depth test: problemi di precisione

- ✓ Il depth buffer memorizza valori scalari da 0 a 1 inclusi
- ✓ Possono essere rappresentati in virgola mobile, ma più spesso sono in virgola fissa, senza segno
- ✓ Es: se usassi 8 bit, allora, memorizzo l'intero x (da 0 a 255) per rappresentare il razionale $x / (255)$
- ✓ La precisione necessaria al test impone di usare più di soli 8 bit ma almeno 16
  - ⇒ Servono ben più di 2^8 = 256 livelli di profondità distinti, per evitare z-fighting
- ✓ Il problema di z-fighting non viene mai completamente evitato, e si manifesta se vengono renderizzate superfici parallele sufficientemente vicine fra loro (oppure concidenti)

38

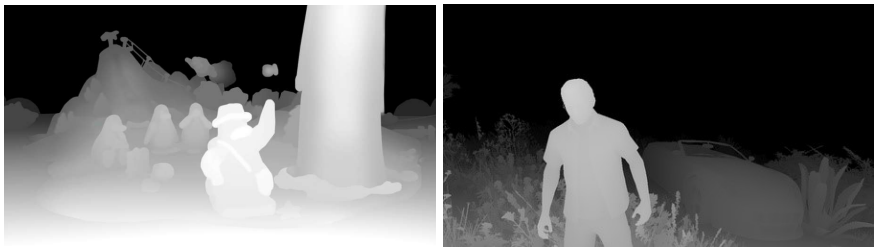
Z-figthing



39

Il rendering produce anche una depth image

- ✓ Effetto collaterale dell'algoritmo di depth test: dopo il rendering, ho prodotto non solo un'immagine (RGB per pixel) ma anche un depth-buffer (profondità per pixel)
  - ⇒ una depth image con un «bassorilievo» della scena renderizzata
  - ⇒ Si tartta quindi di una range scan



  - ⇒ il color-buffer (RGB) viene mandato a schermo,
  - ⇒ il depth-buffer è solo di uso interno e normalmente viene scartato
  - ⇒ alcuni algoritmi di rendering lo sfruttano invece per gli scopi più vari

41

Depth test, vantaggi

- ✓ Il rendering è reso *order independent*:
  - ⇒ se disegno prima la primitiva davanti: i frammenti di quella dietro verranno scartati
  - ⇒ se disegno prima la primitiva dietro: i frammenti della primitiva davanti li sovrascriveranno
  - ⇒ risultato finale: lo stesso! (eccetto i rarissimi casi di pareggio)
  - ⇒ L'efficienza può essere però diversa: scartare è più efficiente di scrivere-per-poi-sovrascrivere
    - Quindi, ordine ottimale: front-to-back

42

Depth test: considerazioni pratiche sull'efficeinza

- ✓ scartare è più efficiente di scrivere-per-poi-sovrascrivere
  - ⇒ quindi: meglio disegnare *prima* le cose davanti *poi* quelle dietro: rendering front-to-back
  - ⇒ l'impatto è accentuato dalle ottimizzazioni HW
  - ⇒ quindi: il depth-sorting è ancora utile, come ottimizzazione
    - effettuato al contrario, che non nell'algoritmo del pittore
  - ⇒ ma, non è necessario (il risultato è sempre corretto)
    - es: può agevolmente essere fatto a livello di oggetto, non di primitive

43

Abilitare il depth test

- ✓ Il depth test è implementato ad HW nella GPU
- ✓ Quindi sia nelle API grafiche a basso livello che (a maggior ragione) nelle librerie ad alto livello, il programmatore si limita ad abilitarlo oppure disabilitarlo
  - ⇒ Ma, a basso livello, è responsabilità del programmatore cancellare il depth buffer (settarlo al valore massimo) prima di ogni rendering
- ✓ In three.js, questa scelta è effettuata settando un apposito flag nel materiale della mesh renderizzata
  - ⇒ `mioMateriale1.depthTest = true;`
  - ⇒ Di default, è abilitata
- ✓ In molti contesti (come video-gaming) il depth test si assume, di norma, sempre abilitato

46